

PRINCIPLES OF OPERATING SYSTEMS

Edited By
Mandeep Kaur

ISBN: 978-93-87034-84-6

Printed by
EXCEL BOOKS PRIVATE LIMITED
Regd. Office: E-77, South Ext. Part-I, Delhi-110049
Corporate Office: 1E/14, Jhandewalan Extension, New Delhi-110055


+91-8800697053, +91-011-47520129


info@excelbooks.com/projects@excelbooks.com
internationalalliance@excelbooks.com


www.excelbooks.com



for
Lovely Professional University
Phagwara

CONTENTS

Unit 1:	Operating System *Pooja Gupta, Lovely Professional University*	1
Unit 2:	Process Management-I *Sahil Rampal, Lovely Professional University*	41
Unit 3:	Process Management-II *Sahil Rampal, Lovely Professional University*	60
Unit 4:	Process Management-III *Manpreet Kaur, Lovely Professional University*	108
Unit 5:	Memory Management *Deepak Mehta, Lovely Professional University*	147
Unit 6:	File Management *Manmohan Sharma, Lovely Professional University*	198
Unit 7:	Secondary Storage Structure *Sarabjit Kumar, Lovely Professional University*	237
Unit 8:	System Protection *Deepak Mehta, Lovely Professional University*	265
Unit 9:	System Security *Anil Sharma, Lovely Professional University*	284
Unit 10:	Introduction of Windows and its Programming *Deepak Mehta, Lovely Professional University*	311
Unit 11:	Operating System Structure *Sarabjit Kumar, Lovely Professional University*	326
Unit 12:	Process and Threads in Windows *Pawan Kumar, Lovely Professional University*	345
Unit 13:	Input/Output and Security of Windows *Mithilesh Kumar Dubey, Lovely Professional University*	371
Unit 14:	Case Study of Linux Operating System *Mandeep Kaur, Lovely Professional University*	397

SYLLABUS

Principles of Operating Systems

Objectives: The objective of this course is to introduce the essential elements of operating systems. The organizati on andstructure of modern operati ng systems and concurrent programming concepts will be discussed in the course. The knowledgeof operati ng system plays signifi cant role in the program since it manages computer system resources and is essenti al to theireffi cient use.

Sr. No.	Topics
1.	**Introduction:** Operating system Meaning, Supervisor & User mode, Meaning of System Calls & Kernel, Operating system operations & Functions, Types of OS: Single-processor system, multiprogramming, Multiprocessing, Multitasking, Parallel, Distributed, RTOS etc.
2.	**Process management**: Process Concept, PCB, Process Scheduling, Cooperating Processes, Overview of Inter process Communication.
3.	**Process Management:** Concept of Thread, Multithreading, Context Switching, scheduling criteria, Type of Scheduling: Long term, Short term & Medium term scheduling, scheduling algorithms, Overview of thread scheduling
4.	**Process Management:** Concept of critical section, Ways to handle critical section problem, semaphores, Deadlock concept & handling
5.	**Memory Management**: Logical & Physical Address space, Swapping, Contiguous memory allocation, paging, segmentation, Virtual memory, demand paging, Overview of Page replacement, Thrashing
6.	**File Management:** File concepts, access methods, directory structure, file sharing, protection, Allocation methods, Free space Mgt., Directory Implementation.
7.	**Secondary Storage Structure:** disk structure, Disk Scheduling, disk management, swapspace management, Overview of RAID structure.
8.	**System Protection:** Goals of protection, Access matrix and its implementation, Access control and revocation of access rights, capability-based systems
9.	**System Security:** Security problem, program threats, system and network threats, cryptography as a security tools, user authentication, implementing security defenses, firewalling to protect systems and networks.
10.	Case study of Windows OS or Linux or any other OS

DCAP403 Operating System

Sr. No.	Topics
1.	**Introduction:** Operating system Meaning, Supervisor & User mode, operating system operations & Functions, Types of OS: Single-processor system, multiprogramming, Multiprocessing, Multitasking, Parallel, Distributed, RTOS etc.
2.	**Operating System Structure:** OS Services, System Calls, System Programs, OS Structures, layered structure Virtual machines.
3.	**Processes**: Process Concept, PCB, Operation on Processes, Cooperating Processes, Inter process Communication, Process Communication in Client Server Environment. **Threads:** Concept of Thread, Kernel level & User level threads, Multithreading, Thread Libraries, Threading Issues.
4.	**Scheduling:** scheduling criteria, scheduling algorithms, Type of Scheduling: Long term, Short term & Medium term scheduling, multi-processor scheduling algorithm, thread scheduling.
5.	**Process Synchronization:** Critical Section problem, semaphores, monitors, Deadlock characterization, Handling of deadlocks -deadlock prevention, avoidance, detection, recovery from deadlock.
6.	**Memory Management**: Logical & Physical Address space, Swapping, Contiguous memory allocation, paging, segmentation, Virtual memory, demand paging, Page replacement & Page Allocation algorithms, thrashing, Performance issues.
7.	**File Management:** File concepts, access methods, directory structure, file system mounting, file sharing, protection, Allocation methods, Free space Mgt., Directory Implementation.
8.	**I/O & Secondary Storage Structure:** I/O H/W, Application I/O Interface, Kernel I/O subsystem, Disk Scheduling, disk management, swap-space management, RAID structure.
9.	**System Protection:** Goals of protection, Access matrix and its implementation, Access control and revocation of access rights, capability-based systems.
10.	**System Security:** Security problem, program threats, system and network threats, cryptography as a security tools, user authentication, implementing security defenses, firewalling to protect systems and networks. Case studies Windows OS, Linux or any other OS.

Unit 1: Operating System

Notes

CONTENTS

Objectives
Introduction
1.1 History of Operating Systems
1.1.1 Today
1.1.2 Unix-like Family
1.1.3 Microsoft Windows
1.1.4 Other Systems
1.2 History of Personal Computers
1.2.1 Uses
1.3 Operating System Meaning
1.4 Supervisor and User Mode
1.4.1 Some Examples from the PC World
1.5 System Calls
1.5.1 The Library as an Intermediary
1.5.2 Examples and Tools
1.5.3 Typical Implementations
1.5.4 Types of System Call
1.6 Kernel
1.6.1 Categories of Kernels
1.6.2 The Monolithic versus Micro Controversy
1.7 Operating System Functions
1.7.1 What is an Operating System?
1.7.2 What does a Driver do?
1.7.3 Other Operating System Functions
1.7.4 Operating System Concerns
1.7.5 Types of Operating System
1.8 Hardware ASMP
1.8.1 Overview
1.8.2 Differences between Hardware ASMP and SMP
1.9 Software ASMP
1.9.1 Overview
1.9.2 Differences between Software ASMP and SMP
1.9.3 Modern Applications of ASMP
1.9.4 Graphical Representation of Asymmetric Multiprocessing
1.10 Multitasking
1.11 Distributed Systems
1.11.1 Client-Server Systems
1.11.2 Peer-to-Peer Systems

Notes

1.12 Summary
1.13 Keywords
1.14 Review Questions
1.15 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss history of operating systems
- Explain Unix-like structures in operating systems
- Understand microsoft windows family of operating systems
- Explain history of personal computers
- Discuss meaning of operating systems
- Explain supervisor and user mode
- Understand meaning of system calls and kernel
- Explain operating system operations and functions
- Understand types of operating system: RTOSs, multiprogramming, multitasking, distributed systems, etc

Introduction

Modern general-purpose computers, including personal computers and mainframes, have an operating system to run other programs, such as application software. Examples of operating systems for personal computers include Microsoft Windows, Mac OS (and Darwin), Unix, and Linux. The lowest level of any operating system is its kernel. This is the first layer of software loaded into memory when a system boots or starts up. The kernel provides access to various common core services to all other system and application programs. These services include, but are not limited to: disk access, memory management, task scheduling, and access to other hardware devices.

As the kernel, an operating system is often distributed with tools for programs to display and manage a graphical user interface (although Windows and the Macintosh have these tools built into the operating system), as well as utility programs for tasks such as managing files and configuring the operating system. They are also often distributed with application software that does not relate directly to the operating system's core function, but which the operating system distributor finds advantageous to supply with the operating system.

The delineation between the operating system and application software is not precise, and is occasionally subject to controversy. From commercial or legal points of view, the delineation can depend on the contexts of the interests involved. For example, one of the key questions in the United States v. Microsoft antitrust trial was whether Microsoft's web browser was part of its operating system or whether it was a separable piece of application software.

Like the term "operating system" itself, the question of what exactly should form the "kernel" is subject to some controversy, with debates over whether things like file systems should be included in the kernel. Various camps advocate microkernels, monolithic kernels, and so on. Operating

Notes

systems are used on most, but not all, computer systems. The simplest computers, including the smallest embedded systems and many of the first computers, did not have operating systems. Instead, they relied on the application programs to manage the minimal hardware themselves, perhaps with the aid of libraries developed for the purpose. Commercially-supplied operating systems are present on virtually all modern devices described as computers, from personal computers to mainframes, as well as mobile computers such as PDAs and mobile phones.

1.1 History of Operating Systems

An operating system (OS) is a software program that manages the hardware and software resources of a computer. The OS performs basic tasks, such as controlling and allocating memory, prioritizing the processing of instructions, controlling input and output devices, facilitating networking, and managing files.

The first computers did not have operating systems. However, software tools for managing the system and simplifying the use of hardware appeared very quickly afterwards, and gradually expanded in scope. By the early 1960s, commercial computer vendors were supplying quite extensive tools for streamlining the development, scheduling, and execution of jobs on batch processing systems. Examples were produced by UNIVAC and Control Data Corporation, amongst others.

Through the 1960s, several major concepts were developed, driving the development of operating systems. The development of the IBM System/360 produced a family of mainframe computers available in widely differing capacities and price points, for which a single operating system OS/360 was planned (rather than developing ad hoc programs for every individual model). This concept of a single OS spanning an entire product line was crucial for the success of System/360 and, in fact, IBM's current mainframe operating systems are distant descendants of this original system; applications written for the OS/360 can still be run on modern machines. OS/360 also contained another important advance: the development of the hard disk permanent storage device (which IBM called DASD). Another key development was the concept of time-sharing: the idea of sharing the resources of expensive computers amongst multiple computer users interacting in real time with the system. Time sharing allowed all of the users to have the illusion of having exclusive access to the machine; the Multics timesharing system was the most famous of a number of new operating systems developed to take advantage of the concept. Multics, particularly, was an inspiration to a number of operating systems developed in the 1970s, notably Unix. Another commercially popular minicomputer operating system was VMS. The first microcomputers did not have the capacity or need for the elaborate operating systems that had been developed for mainframes and minis; minimalistic operating systems were developed. One notable early operating system was CP/M, which was supported on many early microcomputers and was largely cloned in creating MS-DOS, which became wildly popular as the operating system chosen for the IBM PC (IBM's version of it was called IBM-DOS or PC-DOS), its successors making Microsoft one of the world's most profitable companies. The major alternative throughout the 1980s in the microcomputer market was Mac OS, tied intimately to the Apple Macintosh computer.

By the 1990s, the microcomputer had evolved to the point where, as well as extensive GUI facilities, the robustness and flexibility of operating systems of larger computers became increasingly desirable. Microsoft's response to this change was the development of Windows NT, which served as the basis for Microsoft's entire operating system line starting in 1999. Apple rebuilt their operating system on top of a Unix core as Mac OS X, released in 2001. Hobbyist-developed reimplementations of Unix, assembled with the tools from the GNU project, also became popular; versions based on the Linux kernel are by far the most popular, with the BSD derived UNIXes holding a small portion of the server market. The growing complexity of embedded devices has a growing trend to use embedded operating systems on them.

Notes

1.1.1 Today

Command line interface (or CLI) operating systems can operate using only the keyboard for input. Modern OS's use a mouse for input with a graphical user interface (GUI) sometimes implemented as a shell. The appropriate OS may depend on the hardware architecture, specifically the CPU, with only Linux and BSD running on almost any CPU. Windows NT has been ported to other CPUs, most notably the Alpha, but not many. Since the early 1990s the choice for personal computers has been largely limited to the Microsoft Windows family and the Unix-like family, of which Linux and Mac OS X are becoming the major choices. Mainframe computers and embedded systems use a variety of different operating systems, many with no direct connection to Windows or Unix, but typically more similar to Unix than Windows.

- Personal computers
- IBM PC compatible — Microsoft Windows and smaller Unix variants (like Linux and BSD)
- Apple Macintosh — Mac OS X, Windows, Linux and BSD
- Mainframes — A number of unique OS's, sometimes Linux and other Unix variants
- Embedded systems — A variety of dedicated OS's, and limited versions of Linux or other OS's

1.1.2 Unix-like Family

The Unix-like family is a diverse group of operating systems, with several major subcategories including System V, BSD, and Linux. The name "Unix" is a trademark of the Open Group which licenses it for use to any operating system that has been shown to conform to the definitions that they have cooperatively developed. The name is commonly used to refer to the large set of operating systems which resemble the original Unix. Unix systems run on a wide variety of machine architectures. They are used heavily as server systems in business as well as workstations in academic and engineering environments. Free software Unix variants, such as Linux and BSD, are increasingly popular. They are used in the desktop market as well, for example Ubuntu, but mostly by hobbyists. Some Unix variants like HP's HP-UX and IBM's AIX are designed to run only on that vendor's proprietary hardware. Others, such as Solaris, can run on both proprietary hardware and on commodity x86 PCs. Apple's Mac OS X, a microkernel BSD variant derived from NeXTSTEP, Mach, and FreeBSD, has replaced Apple's earlier (non-Unix) Mac OS. Over the past several years, free Unix systems have supplanted proprietary ones in most instances. For instance, scientific modelling and computer animation were once the province of SGI's IRIX. Today, they are dominated by Linux-based.

The team at Bell Labs who designed and developed Unix went on to develop and Inferno, which were designed for modern distributed environments. They had graphics built-in, unlike Unix counterparts that added it to the design later did not become popular because, unlike many Unix distributions.

1.1.3 Microsoft Windows

The Microsoft Windows family of operating systems originated as a graphical layer on top of the older MS-DOS environment for the IBM PC. Modern versions are based on the newer Windows NT core that first took shape in OS/2 and borrowed from OpenVMS. Windows runs on 32-bit and 64-bit Intel and AMD computers, although earlier versions also ran on the DEC Alpha, MIPS, and PowerPC architectures (some work was done to port it to the SPARC architecture). As of 2004, Windows held a near-monopoly of around 90% of the worldwide desktop market share, although this is thought to be dwindling due to the increase of interest focused on open source operating systems. It is also used on low-end and mid-range servers, supporting applications such as web servers and database servers. In recent years, Microsoft has spent significant marketing and R&D money to demonstrate that Windows is capable of running any enterprise application (see the TPC article).

Notes

1.1.4 Other Systems

Mainframe operating systems, such as IBM's z/OS, and embedded operating systems such as VxWorks, eCos, and Palm OS, are usually unrelated to Unix and Windows, except for Windows CE, Windows NT Embedded 4.0 and Windows XP Embedded which are descendants of Windows, and several *BSDs, and Linux distributions tailored for embedded systems. OpenVMS from Hewlett-Packard (formerly DEC) is still under active development. Older operating systems which are still used in niche markets include the Windows like OS/2 from IBM; Mac OS, the non-Unix precursor to Apple's Mac OS X; BeOS; RISC OS; and AmigaOS. Research and development of new operating systems continues. GNU Hurd is designed to be backwards compatible with Unix, but with enhanced functionality and a microkernel architecture. Microsoft Singularity is a research project to develop an operating system with better memory protection.

1.2 History of Personal Computers

A personal computer (PC) is usually a microcomputer whose price, size, and capabilities make it suitable for personal usage. The term was popularized by IBM marketing.

Figure 1.1: Personal Computer

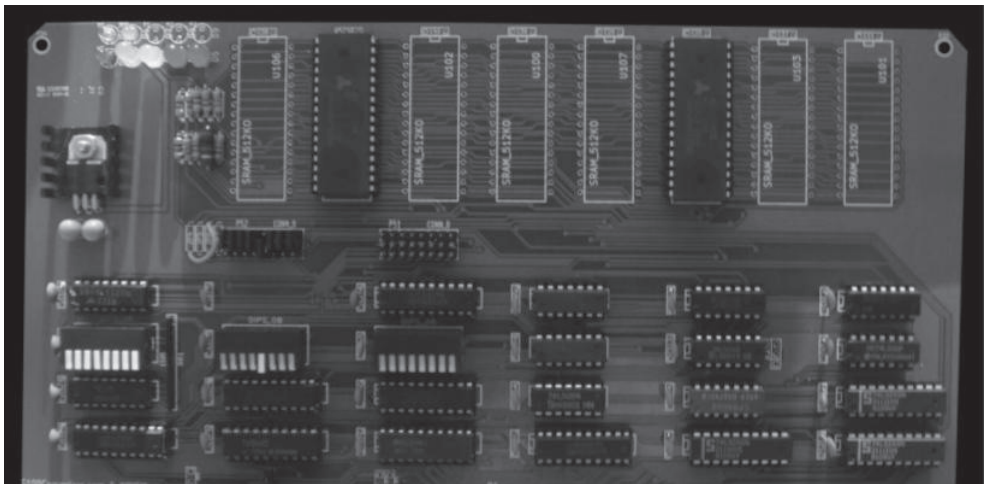


Time share "terminals" to central computers were sometimes used before the advent of the PC. (A smart terminal — televideo ASCII character mode terminal made around 1982.) Before their advent in the late 1970s to the early 1980s, the only computers one might have used if one were privileged were "computer-terminal based" architectures owned by large institutions. In these, the technology was called "computer time share systems", and used minicomputers and main-frame computers. These central computer systems frequently required large rooms — roughly, a handball-court-sized room could hold two to three small minicomputers and its associated peripherals, each housed in cabinets much the size of three refrigerators side by side (with blinking lights and tape drives). In that era, mainframe computers occupied whole floors; a big hard disk was a mere 10–20 megabytes mounted on a cabinet the size of a small chest-type freezer. Earlier PCs were generally called desktop computers, and the slower Pentium-based

Notes

personal computer of the late 1990s could easily outperform the advanced minicomputers of that era. Since the terms "personal computer" and "PC" have been introduced to vernacular language, their meanings and scope have changed somewhat. The first generations of personal microcomputers were usually sold as kits or merely instructions, and required a somewhat skilled person to assemble and operate them. These were usually called microcomputers, but personal computer was also used. Later generations were sometimes interchangeably called by the names "home computer" and "personal computer". By the mid-1980s, "home computer" was becoming a less common label in favour of "personal computer". These computers were pre-assembled and required little to no technical knowledge to operate. In today's common usage, personal computer and PC usually indicate an IBM PC compatible. Because of this association, some manufacturers of personal computers that are not IBM PCs avoid explicitly using the terms to describe their products. Mostly, the term PC is used to describe personal computers that use Microsoft Windows operating systems.

Figure 1.2: Random-Access Memory



A four-megabyte RAM card measuring about 22 by 15 inches; made for the VAX 8600 minicomputer (circa 1986). Dual in-line package (DIP) Integrated circuits populate nearly the whole board; the RAM chips are in the majority located in the rectangular areas to the left and right. One early use of "personal computer" appeared in a 3 November 1962, New York Times article reporting John W. Mauchly's vision of future computing as detailed at a recent meeting of the American Institute of Industrial Engineers. Mauchly stated, "There is no reason to suppose the average boy or girl cannot be master of a personal computer." Some of the first computers that might be called "personal" were early minicomputers such as the LINC and PDP-8. By today's standards they were very large (about the size of a refrigerator) and cost-prohibitive (typically tens of thousands of US dollars), and thus were rarely purchased by an individual. However, they were much smaller, less expensive, and generally simpler to operate than many of the mainframe computers of the time. Therefore, they were accessible for individual laboratories and research projects. Minicomputers largely freed these organizations from the batch processing and bureaucracy of a commercial or university computing centre. In addition, minicomputers were relatively interactive and soon had their own operating systems. Eventually, the minicomputer included VAX and larger minicomputers from Data General, Prime, and others. The minicomputer era largely was a precursor to personal computer usage and an intermediary step from mainframes.

Notes

Development of the single-chip microprocessor was an enormous catalyst to the popularization of cheap, easy to use, and truly personal computers. Arguably the first true "personal computer" was the Altair 8800, which brought affordable computing to an admittedly select market in the 1970s. However, it was arguably this computer that spawned the development of both Apple Computer as well as Microsoft, spawning the Altair BASIC programming language interpreter, Microsoft's first product. The first generation of microcomputers (computers based on a microprocessor) that appeared in the mid-1970s, due to success of the Steve Wozniak-designed Apple Computer release, the Apple II, were usually known as home computers. These were less capable and in some ways less versatile than large business computers of the day. They were generally used by computer enthusiasts for learning to program, running simple office/productivity applications, electronics interfacing, and general hobbyist pursuits. It was the launch of the VisiCalc spreadsheet, initially for the Apple II (and later for the Atari 8-bit family, Commodore PET, and IBM PC) that became the "killer app" that turned the microcomputer into a business tool. This was followed by the August 1981 release of the IBM PC which would revolutionize the computer market. The Lotus 1-2-3, a combined spreadsheet (partly based on VisiCalc), presentation graphics, and simple database application, would become the PC's own killer app. Good word processor programs would also appear for many home computers, in particular the introduction of Microsoft Word for the Apple Macintosh in 1985 (while earlier versions of Word had been created for the PC, it became popular initially through the Macintosh).

In the January 3, 1983 issue of Times magazine the personal computer was named the "Machine of the Year" or its Person of the Year for 1982. During the 1990s, the power of personal computers increased radically, blurring the formerly sharp distinction between personal computers and multiuser computers such as mainframes. Today higher-end computers often distinguish themselves from personal computers by greater reliability or greater ability to multitask, rather than by brute CPU ability.

1.2.1 Uses

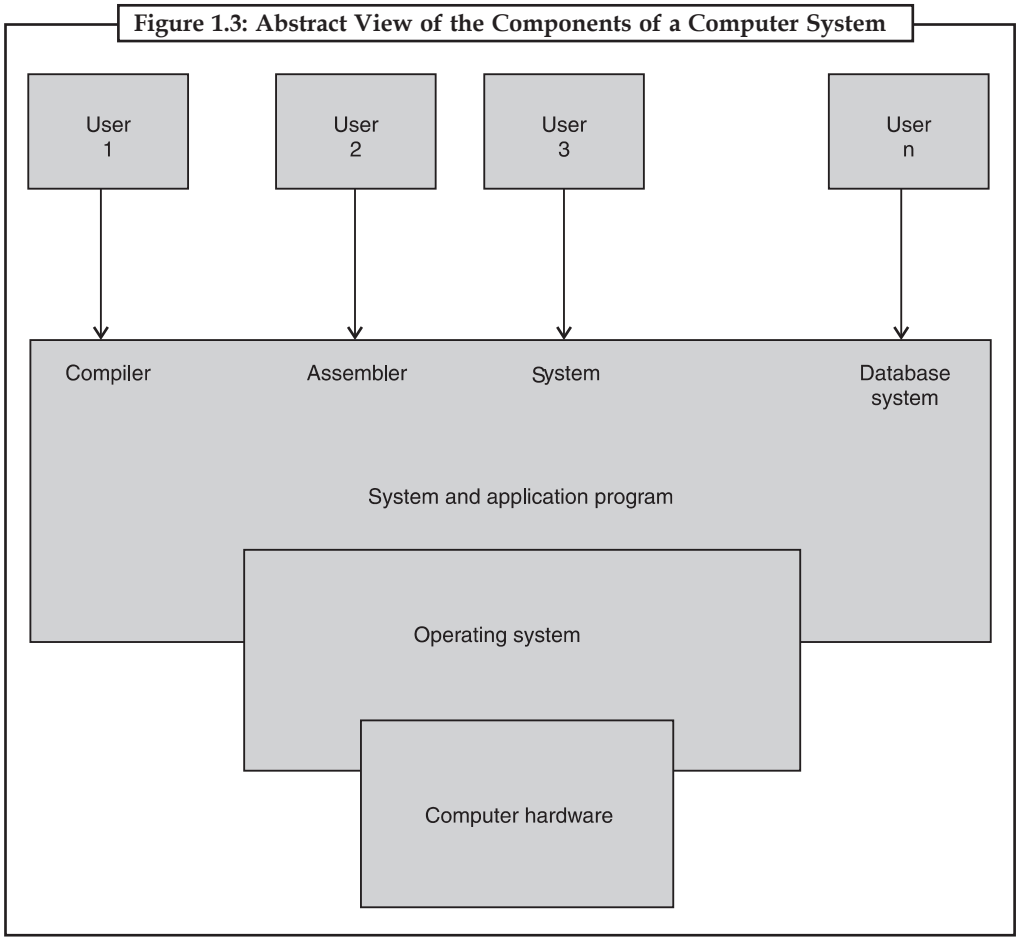
Personal computers are normally operated by one user at a time to perform such general purpose tasks as word processing, internet browsing, e-mail and other digital messaging, multimedia playback, video game play, computer programming, etc. Other more specific functions usually performed with the help of a PC include working, teleworking, learning, researching, printing, online banking, online shopping and dealing online with public sector institutions and services. The user of a modern personal computer may have significant knowledge of the operating environment and application programs, but is not necessarily interested in programming or even able to write programs for the computer. Therefore, most software written primarily for personal computers tends to be designed with simplicity of use, or "user-friendliness" in mind. However, the software industry continuously provide a wide range of new products for use in personal computers, targeted at both the expert and the non-expert user.

1.3 Operating System Meaning

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users (Figure 1.3).

Notes

Figure 1.3: Abstract View of the Components of a Computer System



The hardware-the central processing unit (CPU), the memory, and the input/output (110) devices-provides the basic computing resources. The application programs-such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. The components of a computer system are its hardware, software, and data.

The operating system provides the means for the proper use of these resources in the operation of the computer system. Operating systems can be explored from two viewpoints: the user and the system.


Did u know?

An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

1.4 Supervisor and User Mode

In computer terms, **supervisor mode** is a hardware-mediated flag which can be changed by code running in system-level software. System-level tasks or threads will have this flag set while they are running, whereas user-space applications will not. This flag determines whether

Notes

it would be possible to execute machine code operations such as modifying registers for various descriptor tables, or performing operations such as disabling interrupts. The idea of having two different modes to operate in comes from "with more control comes more responsibility" a program in supervisor mode is trusted to never fail, because if it does, the whole computer system may crash.

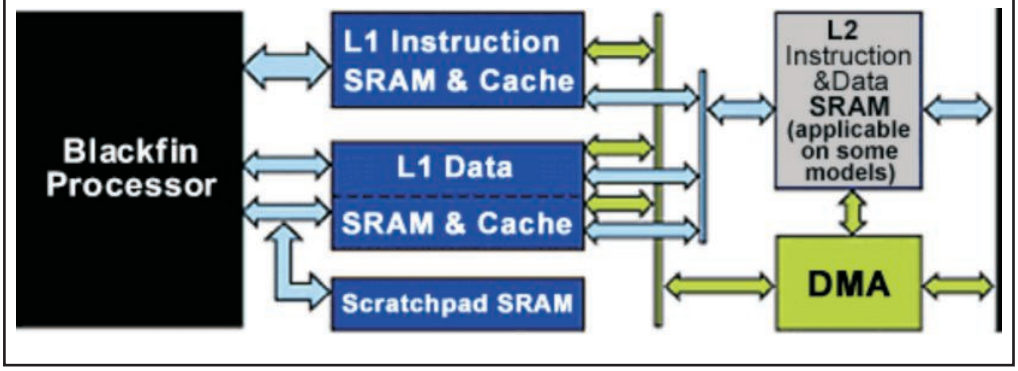
In a monolithic kernel, the kernel runs in supervisor mode and the applications run in user mode. Other types of operating systems, like those with an exokernel or microkernel do not necessarily share this behaviour.

1.4.1 Some Examples from the PC World

Linux and Windows are two operating systems that use supervisor/user-mode. DOS and other simple operating systems run in supervisor mode permanently, meaning that drivers can be written directly into software. In user-mode it would be necessary to utilise a system call into kernel-space (running in supervisor mode) where trusted code in the operation system will perform the needed task.

Most processors have at least two different modes. The x86-processors have four different modes divided into four different "rings". Programs that run in ring0 can do anything with the system and code that runs in ring3 should be able to fail at any time without any impact at the rest of the computer system. Ring1 and ring2 is mostly never used, but could be configured with different levels of access.

Figure 1.4: User and Supervisor Modes



Did u know?

A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.5 System Calls

In computing, a **system call** is the mechanism used by an application program to request service from the operating system based on the monolithic kernel or to system servers on operating systems based on the microkernel-structure. Timings of requested service have to be strictly predictable for application in real time systems — those are most advanced and secure. So far, the only thing we have done was to use well defined kernel mechanisms to register /proc files and device handlers. This is fine if you want to do something the kernel programmers thought you had want, such as write a device driver. But what if you want to do something unusual, to change the behaviour of the system in some way? Then, you are mostly on your own.

Notes


Caution

This is where kernel programming gets dangerous. While writing the example below, I killed the open() system call. This meant I couldn't open any files, I couldn't run any programs, and I couldn't shutdown the computer. I had to pull the power switch. Luckily, no files died. To ensure you won't lose any files either, please run sync right before you do the **insmod** and the **rmmod**.

In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that is the reason why it is called 'protected mode').

System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them)? Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel and, therefore, you are allowed to do whatever you want.

The location in the kernel a process can jump to is called system call. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (sys_call_table) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it is at the source file arch/$<$architecture$>$/kernel/entry.S, after the line ENTRY(system_call). So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at sys_call_table to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it is important for cleanup_module to restore the table to its original state.

The source code here is an example of such a kernel module. We want to 'spy' on a certain user, and to printk( ) a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called our_sys_open. This function checks the uid (user's id) of the current process, and if it is equal to the uid we spy on, it calls printk( ) to display the name of the file to be opened. Then, either way, it calls the original open( ) function with the same parameters, to actually open the file.

The init_module function replaces the appropriate location in sys_call_table and keeps the original pointer in a variable. The cleanup_module function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's open system call will be A open and B's will be B_open. Now, when A is inserted into the kernel, the system call is replaced with A open, which will call the original sys_open when it is done. Next, B is inserted into the kernel, which replaces the system call with B_open, which will call what it thinks is the original system call, A open, when it is done.

Now, if B is removed first, everything will be well it will simply restore the system call to A open, which calls the original. However, if A is removed and then B is removed, the system will crash. A's removal will restore the system call to the original, sys_open, cutting B out of the loop. Then, when B is removed, it will restore the system call to what it thinks is the original, A open, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won't change the system call when it is removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to B_open so that

Notes

it is no longer pointing to A open, so it won't restore it to sys_open before it is removed from memory. Unfortunately, B_open will still try to call A open which is no longer there, so that even without removing B the system would crash.

There are two ways to prevent this problem. The first is to restore the call to the original value, sys_open. Unfortunately, sys_open is not part of the kernel system table in /proc/ksyms, so we can't access it. The other solution is to use the reference count to prevent root from rmmod'ing the module once it is loaded. This is good for production modules, but bad for an educational sample which is why I didn't do it here.

1.5.1 The Library as an Intermediary

Generally, systems provide a library that sits between normal programs and the operating system, usually an implementation of the C library (libc), such as glibc. This library exists between the OS and the application, and increases portability.

On exokernel based systems, the library is especially important as an intermediary. On exokernels, libraries shield user applications from the very low level kernel API, and provide abstractions and resource management.

1.5.2 Examples and Tools

On Unix, Unix-like and other POSIX-compatible Operating Systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill. Many of today's operating systems have hundreds of system calls. For example, Linux has 319 different system calls. Similarly, FreeBSD has almost 330.

Tools such as strace and truss allow a process to execute from start and report all system calls the process invokes, or can attach to an already running process and intercept any system call made by said process if the operation does not violate the permissions of the user. This special ability of the program is usually also implemented with a system call, e.g. the GNU's strace is implemented with ptrace( ).

1.5.3 Typical Implementations

Implementing system calls requires a control transfer which involves some sort of architecture-specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the OS so software simply needs to set up some register with the system call number they want and execute the software interrupt.

For many RISC processors this is the only feasible implementation, but CISC architectures such as x86support additional techniques. One example is SYSCALL/SYSENTER, SYSRET/SYSEXIT (the two mechanisms were independently created by AMD and Intel, respectively, but in essence do the same thing). These are "fast" control transfer instructions that are designed to quickly transfer control to the OS for a system call without the overhead of an interrupt. Linux 2.5 began using this on the x86, where available; formerly it used the INT instruction, where the system call number was placed in the EAX register before interrupt 0x80 was executed.

An older x86 mechanism is called a call gate and is a way for a program to literally call a kernel function directly using a safe control transfer mechanism the OS sets up in advance. This approach has been unpopular, presumably due to the requirement of a far call which uses x86 memory segmentation and the resulting lack of portability it causes, and existence of the faster instructions mentioned above.

For IA64 architecture, EPC (Enter Privileged Mode) instruction is used. The first eight system call arguments are passed in registers, and the rest are passed on the stack.

Notes



A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable.

1.5.4 Types of System Call

System calls can be roughly grouped into five major categories:

1. Process Control
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - wait event, signal event
    - allocate and free memory
2. File Management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes
3. Device Management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
4. Information Maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
5. Communication
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices

Notes



Example:

```
/* syscall.c
*
* System call "stealing" sample.
*/

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h> /* The list of system calls */

/* For the current (process) structure, we need
* this to know who the current user is. */
#include <linux/sched.h>

/* In 2.2.3 /usr/include/linux/version.h includes a
* macro for this, but 2.0.35 doesn't - so I add it

* here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif

/* The system call table (a table of functions). We
* just define this as external, and the kernel will
* fill it up for us when we are insmod'ed
*/

extern void *sys_call_table[];

/* UID we want to spy on - will be filled from the
* command line
*/int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif/

* A pointer to the original system call. The reason
* we keep this, rather than call the original function
* (sys_open), is because somebody else might have
* replaced the system call before us. Note that this
* is not 100% safe, because if another module
* replaced sys_open before us, then when we're inserted
* we'll call the function in that module - and it
* might be removed before we are.
```

Notes

```
*
* Another reason for this is that we can't get sys_open.
* It's a static variable, so it is not exported.
asmlinkage int (*original_call)(const char *, int, int);

/* For some reason, in 2.2.3 current->uid gave me
* zero, not the real user ID. I tried to find what went
* wrong, but I couldn't do it in a short time, and
* I'm lazy - so I'll just use the system call to get the
* uid, the way a process would.
*
* For some reason, after I recompiled the kernel this
* problem went away.
*/
asmlinkage int (*getuid_call)();

/* The function we'll replace sys_open (the function
* called when you call the open system call) with. To
* find the exact prototype, with the number and type
* of arguments, we find the original function first
* (it's at fs/open.c).
*
* In theory, this means that we're tied to the
* current version of the kernel. In practice, the
* system calls almost never change (it would wreck havoc
* and require programs to be recompiled, since the system
* calls are the interface between the kernel and the
* processes).
*/

asmlinkage int our_sys_open(const char *filename,
        int flags,
        int mode)
{
        int i = 0;
        char ch;
/* Check if this is the user we're spying on */
if (uid == getuid_call()) {
/* getuid_call is the getuid system call,
* which gives the uid of the user who
* ran the process which called the system
* call we got */

/* Report the file, if relevant */
printk("Opened file by %d: ", uid);
do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
get_user(ch, filename+i);
#else
        ch = get_user(filename+i);
#endif
        i++;
        printk("%c", ch);
```

Notes

```
} while (ch != 0);
printk("\n");
}

/* Call the original sys_open - otherwise, we lose
* the ability to open files */
return original_call(filename, flags, mode);
}

/* Initialize the module - replace the system call */
int init_module()
{
/* Warning - too late for it now, but maybe for
* next time... */
printk("I'm dangerous. I hope you did a ");
printk("sync before you insmod'ed me.\n");
printk("My counterpart, cleanup_module(), is even");
printk("more dangerous. If\n");
printk("you value your file system, it will ");
printk("be \"sync; rmmod\" \n");
printk("when you remove this module.\n");

/* Keep a pointer to the original function in
* original_call, and then replace the system call
* in the system call table with our_sys_open */
original_call = sys_call_table[__NR_open];
sys_call_table[__NR_open] = our_sys_open;

/* To get the address of the function for system
* call foo, go to sys_call_table[__NR_foo]. */

printk("Spying on UID:%d\n", uid);

/* Get the system call for getuid */
getuid_call = sys_call_table[__NR_getuid];

return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{

/* Return the system call back to normal */
if (sys_call_table[__NR_open] != our_sys_open) {
printk("Somebody else also played with the ");
printk("open system call\n");
printk("The system may be left in ");
printk("an unstable state.\n");
}

sys_call_table[__NR_open] = original_call;
}
```

Notes



A system call is also known as a monitor call.

1.6 Kernel

The **kernel** is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system.

A kernel can be contrasted with a shell (such as bash, csh or ksh in Unix-like operating systems), which is the outermost part of an operating system and a program that interacts with user commands. The kernel itself does not interact directly with the user, but rather interacts with the shell and other programs as well as with the hardware devices on the system, including the processor (also called the central processing unit or CPU), memory and disk drives.

The kernel is the first part of the operating system to load into memory during booting (i.e., system startup), and it remains there for the entire duration of the computer session because its services are required continuously. Thus it is important for it to be as small as possible while still providing all the essential services needed by the other parts of the operating system and by the various application programs.

Because of its critical nature, the kernel code is usually loaded into a protected area of memory, which prevents it from being overwritten by other, less frequently used parts of the operating system or by application programs. The kernel performs its tasks, such as executing processes and handling interrupts, in kernel space, whereas everything a user normally does, such as writing text in a text editor or running programs in a GUI (graphical user interface), is done in user space. This separation is made in order to prevent user data and kernel data from interfering with each other and thereby diminishing performance or causing the system to become unstable (and possibly crashing).

When a computer crashes, it actually means the kernel has crashed. If only a single program has crashed but the rest of the system remains in operation, then the kernel itself has not crashed. A crash is the situation in which a program, either a user application or a part of the operating system, stops performing its expected function(s) and responding to other parts of the system. The program might appear to the user to freeze. If such program is a critical to the operation of the kernel, the entire computer could stall or shut down.

The kernel provides basic services for all other parts of the operating system, typically including memory management, process management, file management and I/O (input/output) management (i.e., accessing the peripheral devices). These services are requested by other parts of the operating system or by application programs through a specified set of program interfaces referred to as system calls.

Process management, possibly the most obvious aspect of a kernel to the user, is the part of the kernel that ensures that each process obtains its turn to run on the processor and that the individual processes do not interfere with each other by writing to their areas of memory. A process, also referred to as a task, can be defined as an executing (i.e., running) instance of a program.

The contents of a kernel vary considerably according to the operating system, but they typically include (1) a scheduler, which determines how the various processes share the kernel's processing time (including in what order), (2) a supervisor, which grants use of the computer to each process when it is scheduled, (3) an interrupt handler, which handles

Notes

all requests from the various hardware devices (such as disk drives and the keyboard) that compete for the kernel's services and (4) a memory manager, which allocates the system's address spaces (i.e., locations in memory) among all users of the kernel's services.

The kernel should not be confused with the BIOS (Basic Input/Output System). The BIOS is an independent program stored in a chip on the motherboard (the main circuit board of a computer) that is used during the booting process for such tasks as initializing the hardware and loading the kernel into memory. Whereas the BIOS always remains in the computer and is specific to its particular hardware, the kernel can be easily replaced or upgraded by changing or upgrading the operating system or, in the case of Linux, by adding a newer kernel or modifying an existing kernel.

Most kernels have been developed for a specific operating system, and there is usually only one version available for each operating system. For example, the Microsoft Windows 2000 kernel is the only kernel for Microsoft Windows 2000 and the Microsoft Windows 98 kernel is the only kernel for Microsoft Windows 98. Linux is far more flexible in that there are numerous versions of the Linux kernel, and each of these can be modified in innumerable ways by an informed user.

A few kernels have been designed with the goal of being suitable for use with any operating system. The best known of these is the Mach kernel, which was developed at Carnegie-Mellon University and is used in the Macintosh OS X operating system.

It is not necessary for a computer to have a kernel in order for it to be usable, the reason being that it is not necessary for it to have an operating system. That is, it is possible to load and run programs directly on bare metal machines (i.e., computers without any operating system installed), although this is usually not very practical.

In fact, the first generations of computers used bare metal operation. However, it was eventually realized that convenience and efficiency could be increased by retaining small utility programs, such as program loaders and debuggers, in memory between applications. These programs gradually evolved into operating system kernels.

The term kernel is frequently used in books and discussions about Linux, whereas it is used less often when discussing some other operating systems, such as the Microsoft Windows systems. The reasons are that the kernel is highly configurable in the case of Linux and users are encouraged to learn about and modify it and to download and install updated versions. With the Microsoft Windows operating systems, in contrast, there is relatively little point in discussing kernels because they cannot be modified or replaced.

1.6.1 Categories of Kernels

Kernels can be classified into four broad categories: **monolithic kernels, microkernels, hybrid kernels** and **exokernels**. Each has its own advocates and detractors.

Monolithic kernels, which have traditionally been used by Unix-like operating systems, contain all the operating system core functions and the device drivers (small programs that allow the operating system to interact with hardware devices, such as disk drives, video cards and printers). Modern monolithic kernels, such as those of Linux and FreeBSD, both of which fall into the category of Unix-like operating systems, feature the ability to load module at runtime, thereby allowing easy extension of the kernel's capabilities as required, while helping to minimize the amount of code running in kernel space.

A microkernel usually provides only minimal services, such as defining memory address spaces, interprocess communication (IPC) and process management. All other functions, such as hardware management, are implemented as processes running independently of the kernel. Examples of microkernel operating systems are AIX, BeOS, Hurd, Mach, Mac OS X, MINIX and QNX.

Notes

Hybrid kernels are similar to microkernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space. These kernels represent a compromise that was implemented by some developers before it was demonstrated that pure microkernels can provide high performance. Hybrid kernels should not be confused with monolithic kernels that can load modules after booting (such as Linux).

Most modern operating systems use hybrid kernels, including Microsoft Windows NT, 2000 and XP. DragonFly BSD, a recent fork (i.e., variant) of Free BSD, is the first non-Mach based BSD operating system to employ a hybrid kernel architecture.

Exokernels are a still experimental approach to operating system design. They differ from the other types of kernels in that their functionality is limited to the protection and multiplexing of the raw hardware, and they provide no hardware abstractions on top of which applications can be constructed. This separation of hardware protection from hardware management enables application developers to determine how to make the most efficient use of the available hardware for each specific program.

Exokernels in themselves they are extremely small. However, they are accompanied by library operating systems, which provide application developers with the conventional functionalities of a complete operating system. A major advantage of exokernel-based systems is that they can incorporate multiple library operating systems, each exporting a different API (application programming interface), such as one for Linux and one for Microsoft Windows, thus making it possible to simultaneously run both Linux and Windows applications.

1.6.2 The Monolithic versus Micro Controversy

In the early 1990s, many computer scientists considered monolithic kernels to be obsolete, and they predicted that microkernels would revolutionize operating system design. In fact, the development of Linux as a monolithic kernel rather than a microkernel led to a famous flame war (i.e., a war of words on the Internet) between Andrew Tanenbaum, the developer of the MINIX operating system, and Linus Torvalds, who originally developed Linux based largely on MINIX.

Proponents of microkernels point out that monolithic kernels have the disadvantage that an error in the kernel can cause the entire system to crash. However, with a microkernel, if a kernel process crashes, it is still possible to prevent a crash of the system as a whole by merely restarting the service that caused the error. Although this sounds sensible, it is questionable how important it is in reality, because operating systems with monolithic kernels such as Linux have become extremely stable and can run for years without crashing.

Another disadvantage cited for monolithic kernels is that they are not portable; that is, they must be rewritten for each new architecture (i.e., processor type) that the operating system is to be used on. However, in practice, this has not appeared to be a major disadvantage, and it has not prevented Linux from being ported to numerous processors.

Monolithic kernels also appear to have the disadvantage that their source code can become extremely large. Source code is the version of software as it is originally written(i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters) and before it is converted by a compiler into object code that a computer's processor can directly read and execute.

For example, the source code for the Linux kernel version 2.4.0 is approximately 100MB and contains nearly 3.38 million lines, and that for version 2.6.0 is 212MB and contains 5.93 million lines. This adds to the complexity of maintaining the kernel, and it also makes it difficult for new

Notes

generations of computer science students to study and comprehend the kernel. However, the advocates of monolithic kernels claim that in spite of their size such kernels are easier to design correctly, and thus they can be improved more quickly than can microkernel-based systems.

Moreover, the size of the compiled kernel is only a tiny fraction of that of the source code, for example roughly 1.1MB in the case of Linux version 2.4 on a typical Red Hat Linux 9 desktop installation. Contributing to the small size of the compiled Linux kernel is its ability to dynamically load modules at runtime, so that the basic kernel contains only those components that are necessary for the system to start itself and to load modules.

The monolithic Linux kernel can be made extremely small not only because of its ability to dynamically load modules but also because of its ease of customization. In fact, there are some versions that are small enough to fit together with a large number of utilities and other programs on a single floppy disk and still provide a fully functional operating system (one of the most popular of which is muLinux). This ability to miniaturize its kernel has also led to a rapid growth in the use of Linux in embedded systems (i.e., computer circuitry built into other products).

Although microkernels are very small by themselves, in combination with all their required auxiliary code they are, in fact, often larger than monolithic kernels. Advocates of monolithic kernels also point out that the two-tiered structure of microkernel systems, in which most of the operating system does not interact directly with the hardware, creates a not-insignificant cost in terms of system efficiency.



Did u know?

File-system management is one of the most visible components of an operating system.

Self Assessment

Fill in the blanks:

1. An ..... is a software program that manages the hardware and software resources of a computer.
2. Operating systems can be explored from two viewpoints: the ..... and the system.
3. A ..... manages the execution of user programs.
4. A ..... is the mechanism used by an application program to request service from the operating system.

1.7 Operating System Functions

1.7.1 What is an Operating System?

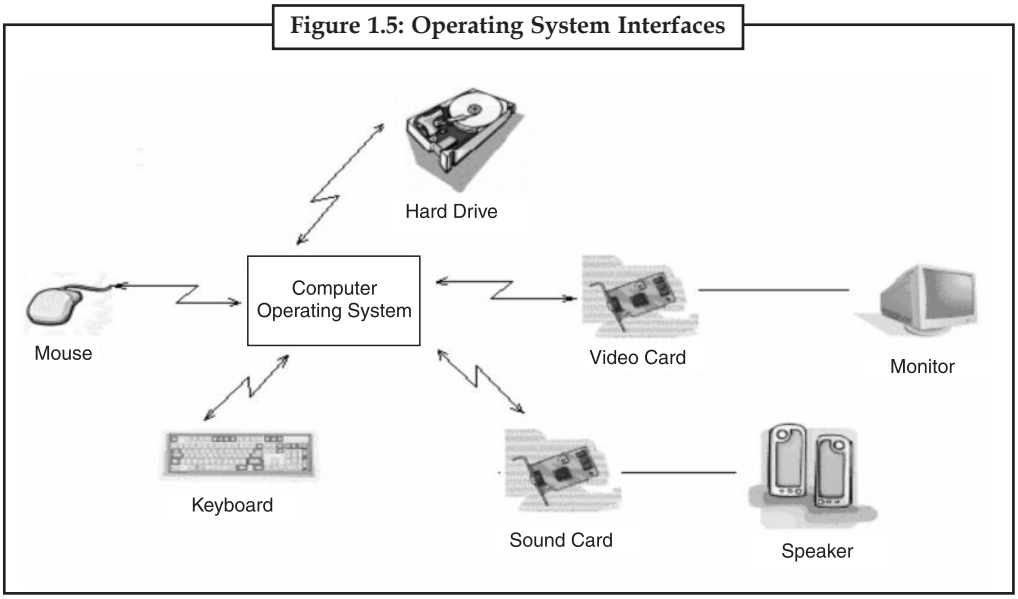
The operating system is the core software component of your computer. It performs many functions and is, in very basic terms, an interface between your computer and the outside world. In the section about hardware, a computer is described as consisting of several component parts including your monitor, keyboard, mouse, and other parts. The operating system provides an interface to these parts using what is referred to as “drivers”. This is why sometimes when you install a new printer or other piece of hardware, your system will ask you to install more software called a driver.

Notes

1.7.2 What does a Driver do?

A driver is a specially written program which understands the operation of the device it interfaces to, such as a printer, video card, sound card or CD ROM drive. It translates commands from the operating system or user into commands understood by the component computer part it interfaces with. It also translates responses from the component computer part back to responses that can be understood by the operating system, application program, or user. The below diagram gives a graphical depiction of the interfaces between the operating system and the computer component.

Figure 1.5: Operating System Interfaces



1.7.3 Other Operating System Functions

The operating system provides for several other functions including:

- System tools (programs) used to monitor computer performance, debug problems, or maintain parts of the system.
- A set of libraries or functions which programs may use to perform specific tasks especially relating to interfacing with computer system components.

The operating system makes these interfacing functions along with its other functions operate smoothly and these functions are mostly transparent to the user.

1.7.4 Operating System Concerns

As mentioned previously, an operating system is a computer program. Operating systems are written by human programmers who make mistakes. Therefore there can be errors in the code even though there may be some testing before the product is released. Some companies have better software quality control and testing than others so you may notice varying levels of quality from operating system to operating system. Errors in operating systems cause three main types of problems:

- System crashes and instabilities: These can happen due to a software bug typically in the operating system, although computer programs being run on the operating system can make the system more unstable or may even crash the system by themselves. This varies depending on the type of operating system. A system crash is the act of a system freezing and becoming unresponsive which would cause the user to need to reboot.

Notes

- Security flaws: Some software errors leave a door open for the system to be broken into by unauthorized intruders. As these flaws are discovered, unauthorized intruders may try to use these to gain illegal access to your system. Patching these flaws often will help keep your computer system secure. How this is done will be explained later.
- Sometimes errors in the operating system will cause the computer not to work correctly with some peripheral devices such as printers.


Caution

Giving sound command make sure sound card is enabled.

1.7.5 Types of Operating System

Within the broad family of operating systems, there are generally four types, categorized based on the types of computers they control and the sort of applications they support. The categories are real-time operating system, single user single task, single user multitasking and multi-user.

1.7.5.1 Real-Time Operating System (RTOS)

Real-time operating systems are used to control machinery, scientific instruments and industrial systems such as embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control (manufacturing, production, power generation, fabrication, and refining), and scientific research equipment.

An RTOS typically has very little user-interface capability, and no end-user utilities, since the system will be a "sealed box" when delivered for use. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time, every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.

An RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the given amount of work it can perform over time. Key factors in an RTOS are therefore a minimal interrupt latency (the time between the generation of an interrupt by a device and the servicing of the device which generated the interrupt) and a minimal thread switching latency (the time needed by the operating system to switch the CPU to another thread).

An early example of a large-scale real-time operating system was Transaction Processing Facility. Current users include Sabre (reservations), Amadeus (reservations), VISA Inc (authorizations), Holiday Inn (central reservations), CBOE (order routing), Singapore Airlines, KLM, Qantas, Amtrak, Marriott International , Worldspan and the NYPD (911 system).

1.7.5.2 Single User, Single Task

As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm handheld computers is a good example of a modern single-user, single-task operating system.

Notes

Figure 1.6: The Palm Treo 700p is one of Many Smartphones produced that combines Palm PDA functions with a cell phone, allowing for built-in voice and Data



1.7.5.3 Single User, Multitasking

This is the type of operating system most people use on their desktop and laptop computers today. Microsoft's Windows and Apple's Mac OS platforms are both examples of operating systems that will let a single user have several programs in operation at the same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message.

Multi-user: Multi-user is a term that defines an operating system or application software that allows concurrent access by multiple users of a computer. A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user doesn't affect the entire community of users. Unix, VMS and mainframe operating systems, such as MVS, are examples of multi-user operating systems.

Time-sharing systems are multi-user systems. Most batch processing systems for mainframe computers may also be considered "multi-user", to avoid leaving the CPU idle while it waits for I/O operations to complete. However, the term "multi-tasking" is more common in this context.

An example is a Unix server where multiple remote users have access (such as via Secure Shell) to the Unix shell prompt at the same time. Another example uses multiple X Window sessions spread across multiple terminals powered by a single machine — this is an example of the use of thin client.

Management systems are implicitly designed to be used by multiple users, typically one system administrator or more and an end-user community.

It's important to differentiate between multi-user operating systems and single-user operating systems that support networking. Windows 2000 and Novell Netware can each support hundreds or thousands of networked users, but the operating systems themselves are not true multi-user operating systems. The **system administrator** is the only "user" for Windows 2000 or Netware. The network support and all of the remote user logins the network enables are, in the overall plan of the operating system, a program being run by the administrative user.

1.7.5.4 Multiprogramming

Disjoint Processes:

Our starting point is the concurrent statement:

cobegin S1; S2; . . . ; Sn coend

Notes

This notation indicates that statements S1, S2, . . . , Sn can be executed concurrently; when all of them are terminated, the following statement in the program (not shown here) is executed. This restricted form of concurrency simplifies the understanding and verification of programs considerably, compared to unstructured fork and join primitives. Algorithm 1 illustrates the use of the concurrent statement to copy records from one sequential file to another.

```
var f, g: ¯le of T;
s, t: T; eof: Boolean;
begin
     input(f, s, eof );
     while not eof do
     begin t := s;
     cobegin
     output(g, t);
     input(f, s, eof );
     coend
     end
 end
```


Caution

Giving video command make sure Video card is enabled.

Algorithm 1: Copying of a sequential file

The variables here are two sequential files, f and g, with records of type T; two buffers, s and t, holding one record each; and a Boolean, eof, indicating whether or not the end of the input file has been reached.

Input and output of single records are handled by two standard procedures. The algorithm inputs a record, copies it from one buffer to another, outputs it, and at the same time, inputs the next record. The copying, output, and input are repeated until the input file is empty.

Now suppose the programmer by mistake expresses the repetition as follows:

```
while not eof do
cobegin
t := s;
output(g, t);
input(f, s, eof );
coend
```

The copying, output, and input of a record can now be executed concurrently. To simplify the argument, we will only consider cases in which these processes are arbitrarily interleaved but not overlapped in time. The erroneous concurrent statement can then be executed in six different ways with three possible results: (1) if copying is completed before input and output are initiated, the correct record will be output; (2) if output is completed before copying is

Notes

initiated, the previous record will be output again; and (3) if input is completed before copying is initiated, and this in turn completed before output is initiated, the next record will be output instead. This is just for a single record of the output file. If we copy a file of 10,000 records, the program can give of the order of 310;000 different results.

The actual sequence of operations in time will depend on the presence of other (unrelated) computations and the (possibly time-dependent) scheduling policy of the installation. It is therefore very unlikely that the programmer will ever observe the same result twice. The only hope of locating the error is to study the program text. This can be very frustrating (if not impossible) when it consists of thousands of lines and one has no clues about where to look. Multiprogramming is an order of magnitude more hazardous than sequential programming unless we ensure that the results of our computations are reproducible in spite of errors. In the previous example, this can easily be checked at compile time. In the correct version of Algorithm 1, the output and input processes operate on disjoint sets of variables (g;t) and (f, s, eof). They are called disjoint or non-interacting processes. In the erroneous version of the algorithm, the processes are not disjoint: the output process refers to a variable t changed by the copying process; and the latter refers to a variable s changed by the input process. This can be detected at compile time if the following rule is adopted: a concurrent statement defines disjoint processes S1; S2; : : : ; Sn which can be executed concurrently. This means that a variable vi changed by statement Si cannot be referenced by another statement S_i (where j =6i). In other words, we insist that a variable subject to change by a process must be strictly private to that process; but disjoint processes can refer to shared variables not changed by any of them.

Throughout this paper, we tacitly assume that sequential statements and assertions made about them only refer to variables which are accessible to the statements according to the rules of disjointness and mutual exclusion. The latter rule will be defined in Section 3. Violations of these rules must be detected at compile time and prevent execution. To enable a compiler to check the disjointness of processes the language must have the following property—it must be possible by simple inspection of a statement to distinguish between its constant and variable parameters. We will not discuss the influence of this requirement on language design beyond mentioning that it makes unrestricted use of pointers and side effects unacceptable. The rule of disjointness is due to Hoare (1971). It makes the axiomatic property of a concurrent statement S very simple: if each component statement Si terminates with a result Ri provided a predicate Pi holds before its execution then the combined effect of S is the following:

"P" S "R"

where

P = P1 & P2 & …….. & Pn

R = R1 & R2 & ……. & Rn

As Hoare puts it: "Each S_i makes its contribution to the common goal."

Mutual Exclusion: The usefulness of disjoint processes has its limits. We will now consider interacting processes and concurrent processes which access shared variables. A shared variable v of type T is declared as follows:

Var v: shared T

Notes

Concurrent processes can only refer to and change a shared variable inside a structured statement called a critical region.

region v do S

This notation associates a statement S with a shared variable v.

Critical regions referring to the same variable exclude each other in time. They can be arbitrarily interleaved in time. The idea of progressing towards a final result (as in a concurrent statement) is therefore meaningless. All one can expect is that each critical region leaves certain relationships among the components of a shared variable v unchanged. These relationships can be defined by an assertion It is about v which must be true after initialization of v and before and after each subsequent critical region associated with v. Such an assertion is called an invariant.

When a process enters a critical region to execute a statement S, a predicate P holds for the variables accessible to the process outside the critical region and an invariant. It holds for the shared variable v accessible inside the critical region. After the completion of S, a result R holds for the former variables and invariant. It has been maintained. So a critical region has the following axiomatic property:

"P "

region v do \P &I" S \R&I";

"R"

Process Communication: Mutual exclusion of operations on shared variables makes it possible to make meaningful statements about the effect of concurrent computations. But when processes cooperate on a common task they must also be able to wait until certain conditions have been satisfied by other processes. For this purpose it introduce a synchronizing primitive, await, which delays a process until the components of a shared variable v satisfy a condition.

B:

region v do

begin . . . await B; . . . end

The await primitive must be textually enclosed by a critical region. If critical regions are nested, the synchronizing condition B is associated with the innermost enclosing region.

"Consumer" "Producer"

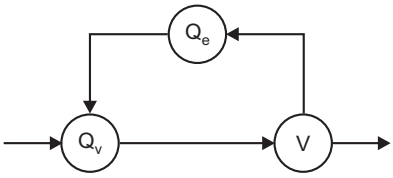
region v do region v do S_2

begin await B; S_1 end

The implementation of critical regions and await primitives is illustrated in Fig.1.7. When a process, such as the consumer above, wishes to enter a critical region, it enters a main queue Qv associated with a shared variable v. After entering its critical region, the consumer inspects the shared variable to determine whether it satisfies a condition B. In that case, the consumer completes its critical region by executing a statement S1; otherwise, the process leaves its critical region temporarily and joins an event queue Qe associated with the shared variable.

Notes

Figure 1.7: Scheduling of Conditional Critical Regions V by Means of Process Queues Q_v and Q_e



All processes waiting for one condition or another on variable v enter the same event queue. When another process (here called the producer) changes v by a statement S_2 inside a critical region, it is possible that one or more of the conditions expected by processes in the event queue will be satisfied. So, after completion of a critical region, all processes in the event queue Q_e are transferred to the main queue Q_v to enable them to reenter their critical regions and inspect the shared variable v again. It is possible that a consumer will be transferred in vain between Q_v and Q_e several times before its condition B holds. But this can only occur as frequently as producers change the shared variable. This controlled amount of busy waiting is the price we pay for the conceptual simplicity achieved by using arbitrary Boolean expressions as synchronizing conditions. The desired invariant it for the shared variable v must be satisfied before an await primitive is executed. When the waiting cycle terminates, the assertion B & I holds.

As an example, consider the following resource allocation problem—two kinds of concurrent processes, called readers and writers, share a single resource. The readers can use the resource simultaneously, but the writers must have exclusive access to it. When a writer is ready to use the resource, it should be enabled to do so as soon as possible. This problem is solved by Algorithm 2. Here variable v is a record consisting of two integer components defining the number of readers currently using the resource and the number of writers currently waiting for or using the resource. Both readers and writers are initialized to zero. var v: shared record readers, writers: integer end w: shared Boolean;

"Reader"	"Writer"
region v do	region v do
begin	begin
await writers = 0;	writers := writers + 1;
readers := readers + 1;	await readers = 0;
end	end
read;	region w do write;
region v do	region v do
readers := readers ¡ 1;	writers := writers ¡ 1;

Algorithm 2: Resource sharing by readers and writers

Mutual exclusion of readers and writers is achieved by letting readers wait until the number of writers is zero, and vice versa. Mutual exclusion of individual writers is ensured by the critical region on the Boolean w. The priority rule is obeyed by increasing the number of writers as soon as one of them wishes to use the resource. This will delay subsequent reader requests until all pending writer requests are satisfied. Algorithm 2 demonstrates the conceptual advantage of a structured notation.

Notes

The conceptual simplicity of critical regions is achieved by ignoring details of scheduling—the programmer is unaware of the sequence in which waiting processes enter critical regions and access shared resources. This assumption is justified for processes which are so loosely connected that simultaneous requests for the same resource rarely occur. But in most computer installations resources are heavily used by a large group of users. In this situation, an operating system must be able to control the scheduling of resources explicitly among competing processes. To do this, a programmer must be able to associate an arbitrary number of event queues with a shared variable and control the transfers of processes to and from them.

The declaration

var e: event v;

associates an event queue e with a shared variable v.

A process can leave a critical region associated with v and join the event queue e by executing the standard procedure

await(e)

Another process can enable all processes in the event queue e to reenter their critical regions by executing the standard procedure

cause(e)

A consumer producer relationship must now be expressed as follows:

"Consumer"	"Producer"
region v do	region v do
begin	begin
while not B do await(e);	S2;
S1;	cause(e);
end	end

Although less elegant than the previous notation, the present one still clearly shows that the consumer is waiting for condition B to hold. And we can now control process scheduling to any degree desired. To simplify explicit scheduling,

```
var v: shared record
                    available: set of R;
                    requests: set of P;
                    grant: array P of event v;
                     end
procedure reserve(process: P; var resource: R);
region v do
begin
        while empty(available) do
        begin enter(process, requests);
                await(grant[process]);
```

Notes

```
end
        remove(resource, available);
end
procedure release(resource: R);
var process: P;
region v do
begin enter(resource, available);
if not empty(requests) then
        begin remove(process, requests);
        cause(grant[process]);
        end
end
```

Algorithm 3: Scheduling of heavily used resources

If the scheduling rule is completely unknown to the programmer as before, additional variables are required to ensure that resources granted to waiting processes remain available to them until they reenter their critical regions.

Algorithm 3 is a simple example of completely controlled resource allocation. A number of processes share a pool of equivalent resources. Processes and resources are identified by indices of type P and R respectively. When resources are available, a process can acquire one immediately; otherwise, it must enter a request in a data structure of type set of P and wait until a resource is granted to it. It is assumed that the program controls the entry and removal of set elements completely.

Conclusion

The essential properties of these concepts are:

1. A distinction between disjoint and interacting processes;
2. An association of shared data with operations defined on them;
3. Mutual exclusion of these operations in time;
4. Synchronizing primitives which permit partial or complete control of process scheduling.

1.7.5.5 Multiprocessing

Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined.

Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the terms multitasking or multiprogramming are more appropriate to describe this concept, which is implemented

Notes

mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two.

Processor Symmetry: In a multiprocessing system, all CPUs may be equal, or some may be reserved for special purposes. A combination of hardware and operating-system software design considerations determine the symmetry (or lack thereof) in a given system. For example, hardware or software considerations may require that only one CPU respond to all hardware interrupts, whereas all other work in the system may be distributed equally among CPUs; or execution of kernel mode code may be restricted to only one processor (either a specific processor, or only one processor at a time), whereas user-mode code may be executed in any combination of processors. Multiprocessing systems are often easier to design if such restrictions are imposed, but they tend to be less efficient than systems in which all CPUs are utilized.

Systems that treat all CPUs equally are called symmetric multiprocessing (SMP) systems. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including asymmetric multiprocessing (ASMP), non-uniform memory access (NUMA) multiprocessing, and clustered multiprocessing.

Processor Coupling: Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory (SMP or UMA), or may participate in a memory hierarchy with both local and shared memory (NUMA). Chip multiprocessors, also known as multi-core computing, involves more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. Mainframe systems with multiple processors are often tightly-coupled.

Loosely-coupled multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system (Gigabit Ethernet is common). A Linux Beowulf cluster is an example of a loosely-coupled system.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly; nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

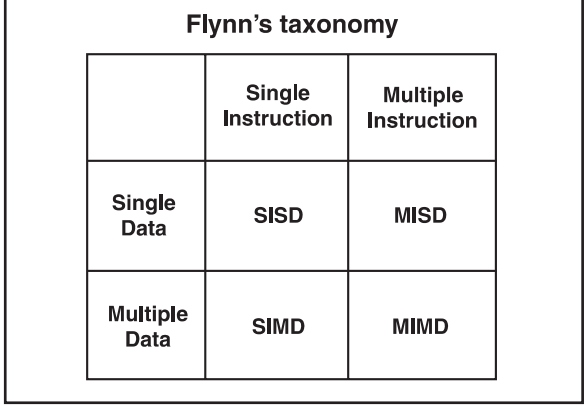
Power consumption is also a consideration. Tightly-coupled systems tend to be much more energy efficient than clusters. This is because considerable economies can be realized by designing components to work together from the beginning in tightly-coupled systems, whereas loosely-coupled systems use components that were not necessarily intended specifically for use in such systems.

Instruction and Data Streams: In multiprocessing, the processors can be used to execute a single sequence of instructions in multiple contexts (single-instruction, multiple-data or SIMD, often used in vector processing), multiple sequences of instructions in a single context (multiple-instruction, single-data or MISD), or multiple sequences of instructions in multiple contexts (multiple-instruction, multiple-data or MIMD).

Notes

Software Implementation Issues

Figure 1.8: SISD Multiprocessing



In 1Tcomputing1T, SISD (Single Instruction, Single Data) is a term referring to a computer architecture in which a single processor, a uniprocessor, executes a single instruction stream, to operate on data stored in a single memory.

In a single instruction stream, single data stream computer one processor sequentially processes instructions; each instruction processes one data item.

SISD is one of the four main classifications. In this system classifications are based upon the number of concurrent instructions and data streams present in the computer architecture. SISD can have concurrent processing characteristics. Instruction fetching and pipelined execution of instructions are common examples found in most modern SISD computers.

SIMD Multiprocessing: In a single instruction stream, multiple data stream computer one processor handles a stream of instructions, each one of which can perform calculations in parallel on multiple data locations.

SIMD multiprocessing is well suited to parallel or vector processing, in which a very large set of data can be divided into parts that are individually subjected to identical but independent operations. A single instruction stream directs the operation of multiple processing units to perform the same manipulations simultaneously on potentially large amounts of data.

For certain types of computing applications, this type of architecture can produce enormous increases in performance, in terms of the elapsed time required to complete a given task. However, a drawback to this architecture is that a large part of the system falls idle when programs or system tasks are executed that cannot be divided into units that can be processed in parallel.

Additionally, programs must be carefully and specially written to take maximum advantage of the architecture, and often special optimizing compilers designed to produce code specifically for this environment must be used.

Some compilers in this category provide special constructs or extensions to allow programmers to directly specify operations to be performed in parallel.

SIMD multiprocessing finds wide use in certain domains such as computer simulation, but is of little use in general-purpose desktop and business computing environments.

Notes

MISD Multiprocessing: Multiple Instruction, Single Data is a type of 1Tparallel computing1T 1Tarchitecture1T where many functional units perform different operations on the same data. 1TPipeline1T architectures belong to this type, though a purist might say that the data is different after processing by each stage in the pipeline.

MISD multiprocessing offers mainly the advantage of redundancy, since multiple processing units perform the same tasks on the same data, reducing the chances of incorrect results if one of the units fails. MISD architectures may involve comparisons between processing units to detect failures. Fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors, in a manner known as 1Ttask replication1T, may be considered to belong to this type.

Apart from the redundant and fail-safe character of this type of multiprocessing, it has few advantages, and it is very expensive .It does not improve performance. Not many instances of this architecture exist, as 1TMIMD1T architectures may be used in a number of application areas such as 1Tcomputer-aided design1T/1Tcomputer-aided manufacturing1T, 1Tsimulation1T, 1Tmodeling1T, and as communication switches. MIMD machines can be of either 1Tshared memory1T or 1Tdistributed memory1T categories. These classifications are based on how MIMD processors access memory.

MIMD multiprocessing architecture is suitable for a wide variety of tasks in which completely independent and parallel execution of instructions touching different sets of data can be put to productive use. For this reason, and because it is easy to implement, MIMD predominates in multiprocessing.

MIMD does raise issues of deadlock and resource contention, however, since threads may collide in their access to resources in an unpredictable way that is difficult to manage efficiently. MIMD requires special coding in the operating system of a computer but does not require application changes. Both system and user software may need to use software constructs such as semaphores (also called locks or gates) to prevent one thread from interfering with another if they should happen to cross paths in referencing the same data. This gating or locking process increases code complexity, lowers performance, and greatly increases the amount of testing required, although not usually enough to negate the advantages of multiprocessing.

Symmetric Multiprocessing: In computing, symmetric multiprocessing or SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory. Most common multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores.

Treating them as Separate Processors: SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

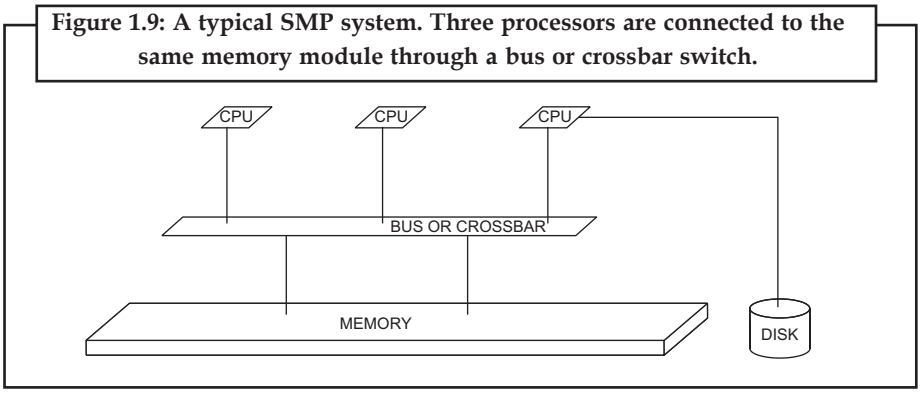
SMP represents one of the earliest styles of multiprocessor machine architectures, typically used for building smaller computers with up to 8 processors. Larger computer systems might use newer architectures such as NUMA (Non-Uniform Memory Access) and 1TSIMD1T are often more appropriate for common data parallel techniques. Specifically, they allow better scaling and use of computational resources than MISD does.

MIMD Multiprocessing: In 1Tcomputing1T, MIMD (Multiple Instruction stream, Multiple Data stream) is a technique employed to achieve parallelism. Machines using MIMD have a number of 1Tprocessors1T that function 1Tasynchronously1T and independently.

At any time, different processors may be executing different instructions on different pieces of data MIMD.

Notes

Figure 1.9: A typical SMP system. Three processors are connected to the same memory module through a bus or crossbar switch.



Asymmetric Multiprocessing

Asymmetric multiprocessing varies greatly from the standard processing model that we see in personal computers today. Due to the complexity and unique nature of this architecture, it was not adopted by many vendors or programmers during its brief stint (1970–1980). Whereas a symmetric multiprocessor or SMP treats all of the processing elements in the system identically, an ASMP system assigns certain tasks only to certain processors.

Asymmetric hardware systems commonly dedicated individual processors to specific tasks. For example, one processor may be dedicated to disk operations, another to video operations, and the rest to standard processor tasks. These systems do not have the flexibility to assign processes to the least-loaded CPU, unlike an SMP system.

Although hardware-level ASMP may not be in use, the idea and logical process is still commonly used in applications that are multiprocessor intensive. Unlike SMP applications, which run their threads on multiple processors, ASMP applications will run on one processor but outsource smaller tasks to another. Although the system may physically be an SMP, the software is still able to use it as an ASMP by simply giving certain tasks to one processor and deeming it the "master", and only outsourcing smaller tasks to "slave "processors.

Task Prepare a list of system requirements before installing an operating system.

1.8 Hardware ASMP

1.8.1 Overview

Asymmetrical multiprocessors are defined by the characteristic that each processor is unique (non-symmetrical). It is common to have one processor that has access to the memory map as a whole, and other processors which simply act as slaves to the main or master processor. Usually, these slave processors will have their own memory which is not tied to the primary processors memory. Slave processors are required to exchange data with the main processor through a partitioned segment of memory that is allocated solely for the purpose of communication. Depending on the hardware in question, each processor may or may not be able to speak to other processors directly.

1.8.2 Differences between Hardware ASMP and SMP

In the symmetrical multiprocessing design, each processor is able to access the entire memory map; there are no master or slave processors. In this case each processor is non-unique and has equal power. This means that they can share memory between themselves and can interact with

Notes

each other directly, regardless of how many there are in the system. People commonly confuse these architectures and as such it is important to define the differences.

1.9 Software ASMP

1.9.1 Overview

Asymmetric multiprocessing as opposed to asymmetrical multiprocessors is the term that refers to software side ASMP. In software each program or application is a process ASMP for software means that all tasks/processes are unique. Thus a given task such as your operating system or favourite game would be assigned to a certain processor. In a more general context "a certain task not runs on every processor. "It is common for application which uses ASMP to work in the following way. The main processor will determine what work needs to be done and will take the bulk of the load, from there it can create instances of the given task on other processors to complete work. Take a video rendering program, the main processor could run the application and the user interface, while offloading the rendering component to a slave processor. This type of action needs to be written into the software and is not decided at the hardware level. It is the programmers' responsibility to determine what jobs should be completed by a given processor.

It must be noted that most applications will ONLY run on the master processor and that the slave processors can merely take on the role of completing tasks that the master processor asks. It is rare that an entire application will or can be run from a slave processor.

1.9.2 Differences between Software ASMP and SMP

Symmetrical multiprocessing, when referring to software, implies the exact opposite of ASMP. In regards to the operating system, a SMP machine is able to spawn any process/task on any of the processors available. Because SMP systems have no master or slave processors, each logical unit is able to complete a given task. In an ASMP system, a certain processor may not be able to complete a task for a number of reason such as the inability to access the entire memory map, special purpose nature of the processor (e.g. a coprocessor) and thus tasks must be give to it by master processor. Therefore, it is up to the programmer to make sure the processors are being used to their maximum potential. In an ASMP environment, a programmer has to worry about whether a processor can complete a given task and how to make the processors communicate effectively to distribute tasks.

1.9.3 Modern Applications of ASMP

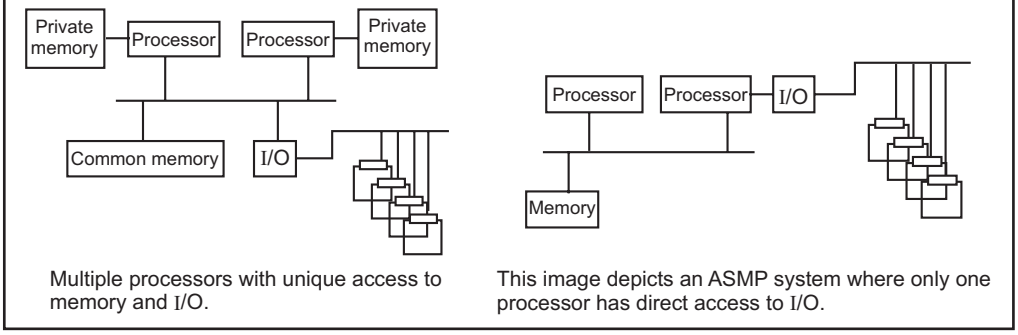
Currently, there are no consumer level production computers that use asymmetric multiprocessor designs. There are, however, computers that are able to distribute tasks Asymmetrically. In theory you are able to use a Symmetrical processor to do asymmetrical computations. A programmer can choose to use one processor as a main, and only offload certain tasks to the other processor. Although each physical or logical processor is able to complete any given task, priority is given to one as the "master" processor, and the other is given the position of "slave".

1.9.4 Graphical Representation of Asymmetric Multiprocessing

Below are examples of what a cluster of asymmetrical multiprocessors would look like. Observe the extremely unique nature of these designs and how only one processor has access to the I/O part of the system. As stated before, these systems work best and were originally designed to do very specific tasks. One processor may simply do physics calculations while another is dedicated to rendering 2D video. Above those two processors, will be a master processor that assigns tasks. Notice also that the main memory is not accessible by all of the processors. The master processor will usually relay information on a "need to know" basis, to the slave processors.

Notes

Figure 1.10: Slave Processors



1.10 Multitasking

An operating system that utilizes multitasking is one that allows more than one program to run simultaneously. If that operating system has cooperative multitasking, it's up to each running program to yield control of system resources to allow the other running applications to perform their chores. In other words, programs must cooperate. In a cooperative multitasking environment, programs can be written such that they do not cooperate graciously—or even such that they do not cooperate at all. A better method of implementing multitasking is for an operating system to employ preemptive multitasking. In a preemptive multitasking environment the operating system can, and does, preempt currently running applications.

With preemptive multitasking, the burden of passing control from one program to another falls on the operating system rather than on running applications. The advantage is that no one program can grab and retain control of system resources. If you have not already guessed, the BeOS has preemptive multitasking. The BeOS microkernel (a low-level task manager discussed later in this unit) is responsible for scheduling tasks according to priority levels. All tasks are allowed use of a processor for only a very short time—three-thousandths of a second. If a program does not completely execute a task in one such time-slice, it will pick up where it left off the next time it regains use of a processor.

1.11 Distributed Systems

A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. By being able to communicate, distributed systems are able to share computational tasks, and provide a rich set of features to users.

Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, although ATM and other protocols are in widespread use. Likewise, operating-system support of protocols varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device-a network adapter, for example-with a device driver to manage it, and software to package data in the communications protocol to send it and to unpackage it to receive it. These concepts are discussed throughout the book.

Networks are typecast based on the distances between their nodes. A local-area network (LAN), exists within a room, a floor, or a building. A wide-area network (WAN), usually exists between buildings, cities, or countries. A global company may have a WAN to connect its offices, worldwide. These networks could run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area**

Notes

network (MAN), could link buildings within a city. BlueTooth devices communicate over a short distance of several feet, in essence creating a **small-area network**.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate they use or create a network. These networks also vary by their performance and reliability.

1.11.1 Client-Server Systems

As PCs have become faster, more powerful, and cheaper, designers have shifted away from the centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user-interface functionality that used to be handled directly by the centralized systems is increasingly being handled by the PCs. As a result, centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client-server system is depicted in Figure Server systems can be broadly categorized as compute servers and fileservers.

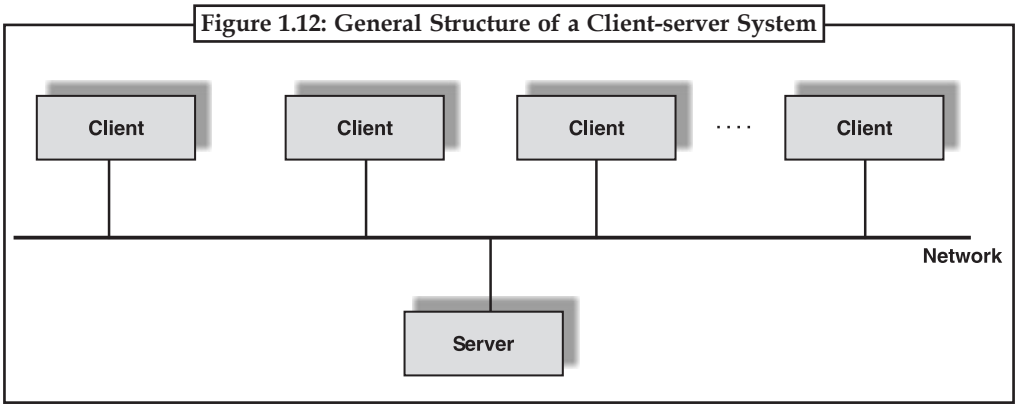
Compute-server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.

File-server systems provide a file-system interface where clients can create, update, read, and delete files.

1.11.2 Peer-to-Peer Systems

The growth of computer networks—especially the Internet and World Wide Web (WWW)—has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for "personal" use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1980s for electronic mail, ftp, and gopher, many PCs became connected to computer networks. With the introduction of the Web in the mid-1990s, network connectivity became an essential component of a computer system.

Figure 1.12: General Structure of a Client-server System



Virtually all modern PCs and workstations are capable of running a web browser for accessing hypertext documents on the Web. Operating systems (such as Windows, OS/2, MacOS, and UNIX) now also include the system software (such as TCP/IP and PPP) that enables a computer to access the Internet via a local-area network or telephone connection. Several include the web browser itself, as well as electronic mail, remote login, and file-transfer clients and servers.

Notes

The computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as **loosely coupled systems** (or **distributed systems**). Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, and that includes a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system is a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network. We cover computer networks and distributed systems in units coming up.



Task Distinguish between the client-server and peer-to-peer models of distributed system.



Case Study

V Distributed Operating Systems

The V operating system is a microkernel operating system that was developed by faculty and students in the Distributed Systems Group at Stanford University in the 1980s. V was the successor to the Thoth and Verax operating systems.

The key concepts in V are multithreading and synchronous message passing. Communication between threads in V uses synchronous message passing, with short, fixed-length messages that can include access rights for the receiver to read or write part of the sender's address space before replying. The same message-passing interface is used both between threads within one process, between threads of different processes within one machine, and between threads on different machines connected by a local Ethernet. A thread receiving a message is not required to reply to it before receiving other messages; this distinguishes the model from Ada rendezvous.

One common pattern for using the messaging facility is for clients to send messages to a server requesting some form of service. From the client side, this looks much like RPC (remote procedure call). The convenience of an automatic stub generator is lacking, but on the other hand, the client can pass one parameter by reference, which is not possible with RPC. From the server side the model differs more from RPC, as by default all client requests are multiplexed onto one server thread. The server is free to explicitly fork threads to handle client requests in parallel, however; if this is done, the server-side model is much like RPC too.

Questions:

1. Differentiate between V distributed operating system and distributed operating system.
2. Define remote procedure call.

Notes

Self Assessment

Multiple choice questions:

5. The operating system manages
   (*a*) Memory (*b*) Processes
   (*c*) Disks and I/O devices (*d*) All of the above
6. Which is not the function of the operating system?
   (*a*) Memory management (*b*) Disk management
   (*c*) Application management (*d*) Virus protection
7. What is a shell?
   (*a*) It is a hardware component (*b*) It is a command interpreter
   (*c*) It is a part in compiler (*d*) It is a tool in CPU scheduling
8. Multiprogramming systems
   (*a*) are easier to develop than single programming systems
   (*b*) execute each job faster
   (*c*) execute more jobs in the same time
   (*d*) are used only on large main frame computers
9. Which of the following operating systems is better for implementing a Client-Server network?
   (*a*) MS DOS (*b*) Windows 95
   (*c*) Windows 98 (*d*) Windows 2000

1.12 Summary

- An operating system (OS) is a software program that manages the hardware and software resources of a computer.
- Command Line Interface (CLI) operating systems can operate using only the keyboard for input.
- Modern OS's use a mouse for input with a graphical user interface (GUI) sometimes implemented as a shell. The Unix-like family, The Microsoft Windows family of operating systems.
- The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system and functions of operating system.

1.13 Keywords

Asymmetric Multiprocessing: Asymmetric hardware systems commonly dedicated individual processors to specific tasks.

Computer Server System: Computer-server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.

Notes

File Server System: File-server systems provide a file-system interface where clients can create, update, read, and delete files.

Kernel: Kernel is a program that constitutes the central core of a computer operating system.

MISD Multiprocessing: Multiple Instruction, Single Data is a type of 1Tparallel computing1T 1Tarchitecture1T where many functional units perform different operations on the same data.

Multitasking: An operating system that utilizes multitasking is one that allows more than one program to run simultaneously.

Operating System: An operating system (OS) is a software program that manages the hardware and software resources of a computer.

Peer-to-Peer System: Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers.

Real Time Operating System (RTOS): Real-time operating systems are used to control machinery, scientific instruments and industrial systems such as embedded systems.

SIMD Multiprocessing: In a single instruction stream, multiple data stream computer one processor handles a stream of instructions, each one of which can perform calculations in parallel on multiple data locations.

Symmetric Multiprocessing: SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory.

System Calls: System call is the mechanism used by an application program to request service from the operating system.

Unix-like: Unix-like family is a diverse group of operating systems, with several major subcategories including System V, BSD, and Linux.



Lab Exercise

1. What are the key ingredients of an operating system?
2. What is the usefulness of system call?

1.14 Review Questions

1. What does an operating system do?
2. What are the three main purposes of an operating system?
3. List the four steps needed to run a program on a completely dedicated machine.
4. What is the main advantage of multiprogramming?
5. What are the main differences between operating systems for mainframe computers and PCs?
6. In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.
   (*a*) What are two such problems?
   (*b*) Can we ensure the same degree of security in a time-shared machine as we have in a dedicated machine? Explain your answer.

Notes

7. How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?
8. How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.
9. Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?
10. Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:
    (*a*) Single-processor systems
    (*b*) Multiprocessor systems
    (*c*) Distributed systems
11. Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.
12. Define the essential properties of the following types of operating systems:

(*a*) Batch	(*b*) Interactive	(*c*) Time sharing
(*d*) Real time	(*e*) Network	(*f*) Parallel
(*g*) Distributed	(*h*) Clustered	(*i*) Handheld

13. What are the tradeoffs inherent in handheld computers?
14. What are the advantages and disadvantages of using the same systemcall interface for manipulating both files and devices?
15. What is the purpose of the command interpreter? Why is it usually separate from the kernel? Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?
16. Why is the separation of mechanism and policy desirable?
17. In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?
18. What is the relationship between a guest operating system and a host operating system in a system like VMware? What factors need to be considered in choosing the host operating system?
19. Describe the actions taken by a kernel to context-switch between processes.
20. What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

Answers to Self Assessment

1. operating system	2. user	3. control program	4. system call
5. (*d*)	6. (*d*)	7. (*b*)	8. (*c*)
9. (*d*)			

Notes

1.15 Further Readings


Books

Introduction to Operating Design and Implementation, by Michael Kifer, Scoott A. Smolka.

Operating Systems, by Stuart E. Madnick, John J. Donovan.


Online link

wiley.com/coolege.silberschatz

Unit 2: Process Management-I

Notes

CONTENTS

Objectives

Introduction

2.1 Process Concept

2.1.1 Process

2.1.2 Process States

2.2 Process Control Block

2.3 Process Scheduling

2.3.1 Scheduling Queues

2.3.2 Schedulers

2.3.3 Context Switch

2.4 Cooperating Processes

2.5 Overview of Inter-Process Communication

2.5.1 Message-passing System

2.5.2 Naming

2.5.3 Direct Communication

2.5.4 Indirect Communication

2.5.5 Synchronization

2.5.6 Buffering

2.5.7 An Example: Mach

2.5.8 An Example: Windows 2000

2.6 Summary

2.7 Keywords

2.8 Review Questions

2.9 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Discuss process concept
- Explain process control block
- Understand process scheduling
- Explain cooperating processes
- Discuss overview of inter process communication

Introduction

Multiprogramming systems explicitly allow multiple processes to exist at any given time, where only one is using the CPU at any given moment, while the remaining processes are performing I/O or are waiting.

The process manager is of the four major parts of the operating system. It implements the process abstraction. It does this by creating a model for the way the process uses CPU and any system resources. Much of the complexity of the operating system stems from the need for multiple processes to share the hardware at the same time. As a conseuence of this goal, the process manager implements CPU sharing (called scheduling), process synchronization mechanisms, and a deadlock strategy. In addition, the process manager implements part of the operating system's protection and security.

2.1 Process Concept

One impediment to our discussion of operating systems is the question of what to call all the CPU activities. A batch system executes jobs, whereas a timeshared system has user programs, or tasks. Even on a single-user system, such as Microsoft Windows and Macintosh OS, a user may be able to run several programs at one time: a word processor, web browser, and e-mail package.

Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes. The terms job and process are used almost interchangeably in this text. Although we personally prefer the term process, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job.

2.1.1 Process

A process is a sequential program in execution. The components of a process are the following:

- The object program to be executed (called the program text in UNIX).
- The data on which the program will execute (obtained from a file or interactively from the process's user).
- Resources required by the program (for example, files containing requisite information).
- The status of the process's execution.

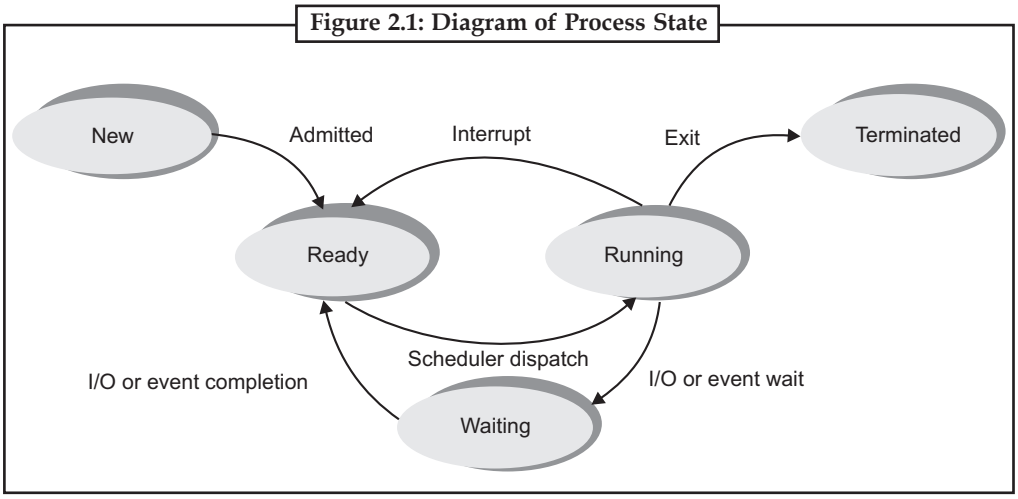
Notes

2.1.2 Process States

During the lifespan of a process, its execution status may be in one of four states (associated with each state is usually a queue on which the process resides):

- **Executing:** The process is currently running and has control of a CPU.
- **Waiting:** The process is currently able to run, but must wait until a CPU becomes available.
- **Blocked:** The process is currently waiting on I/O, either for input to arrive or output to be sent.
- **Suspended:** The process is currently able to run, but for some reason the OS has not placed the process on the ready queue.
- **Ready:** The process is in memory, will execute given CPU time.

Figure 2.1: Diagram of Process State



- **Terminated:** The process has finished execution. These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems more finely delineate process states.



Did u know?

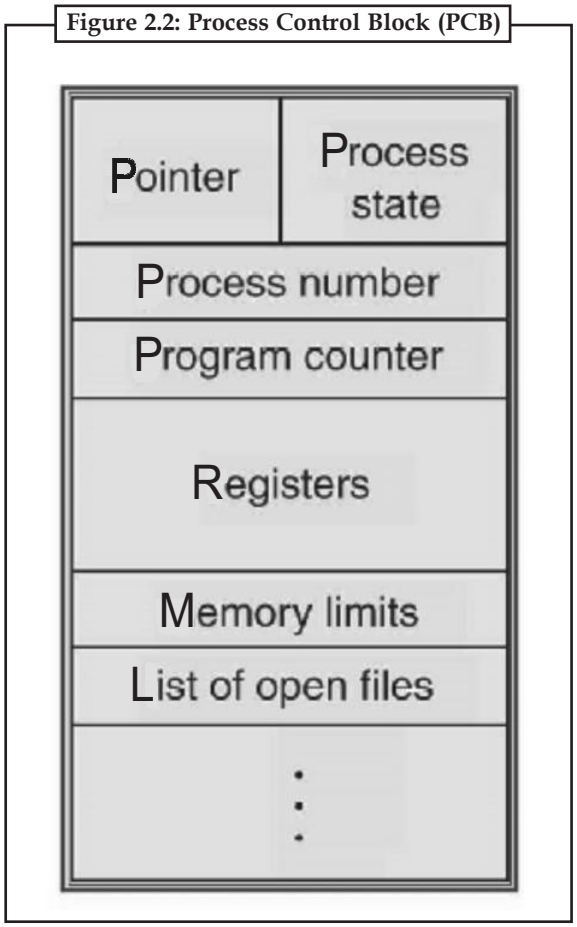
Only one process can be running on any processor at any instant, although many processes may be ready and waiting.

2.2 Process Control Block

Each process is represented in the operating system by a process control block (PCB), also called a task control block. A PCB is shown in Figure 2.2. It contains many pieces of information associated with a specific process, including these:

Notes

Figure 2.2: Process Control Block (PCB)



Process state: The state may be new, ready, running, waiting, halted, and so on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

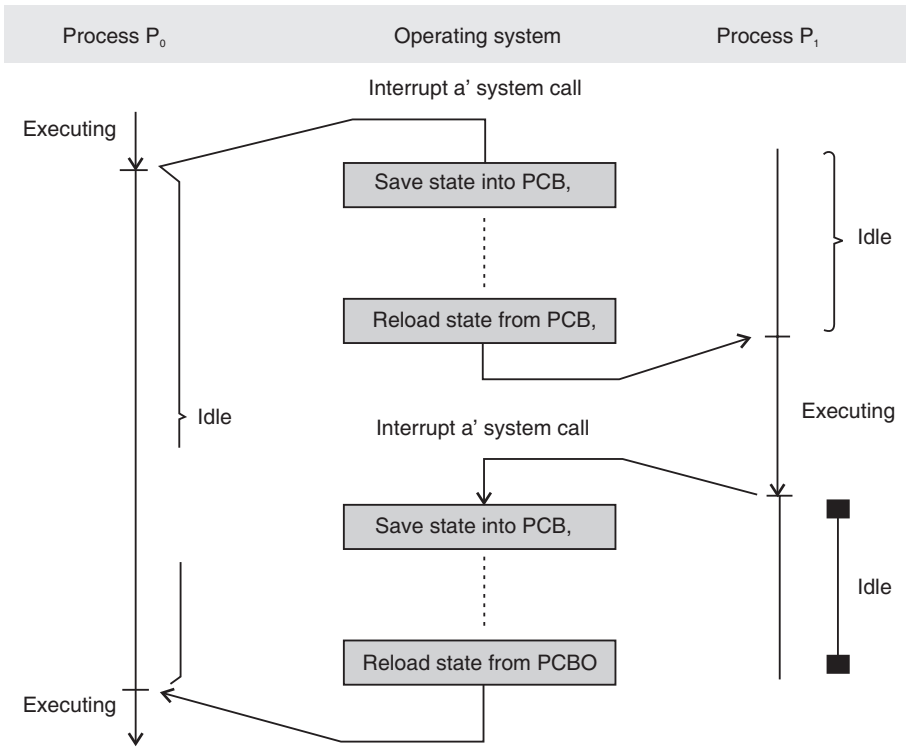
Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 2.3).

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Notes

Figure 2.3: Diagram Showing CPU Switch from Process to Process



Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

Status information: The information includes the list of I/O devices allocated to this process, a list of open files, and so on. The PCB simply serves as the repository for any information that may vary from process to process.


Did u know?

Threads: The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, if a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time.

2.3 Process Scheduling

The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process.

If more processes exist, the rest must wait until the CPU is free and can be rescheduled.

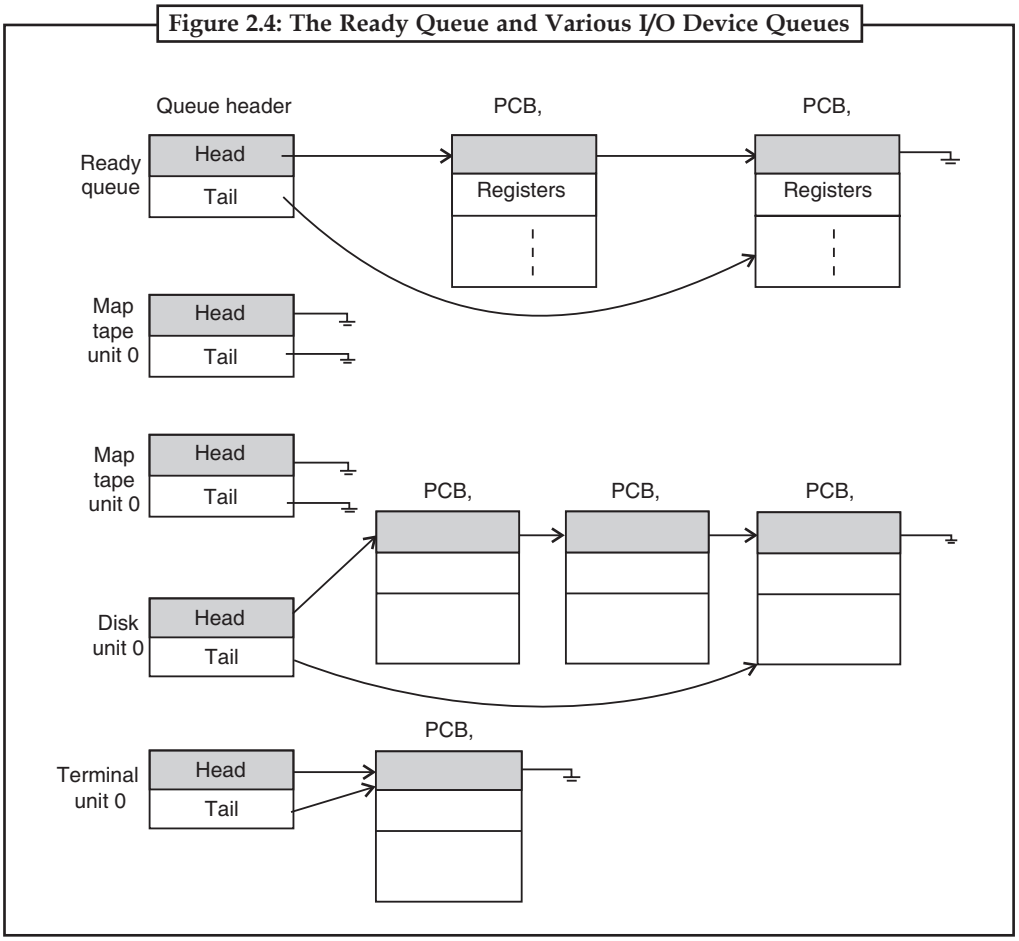
Notes

2.3.1 Scheduling Queues

As processes enter the system, they are put into a job queue. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The operating system also has other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of a I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since the system has many processes, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue (Figure 2.4).

Figure 2.4: The Ready Queue and Various I/O Device Queues



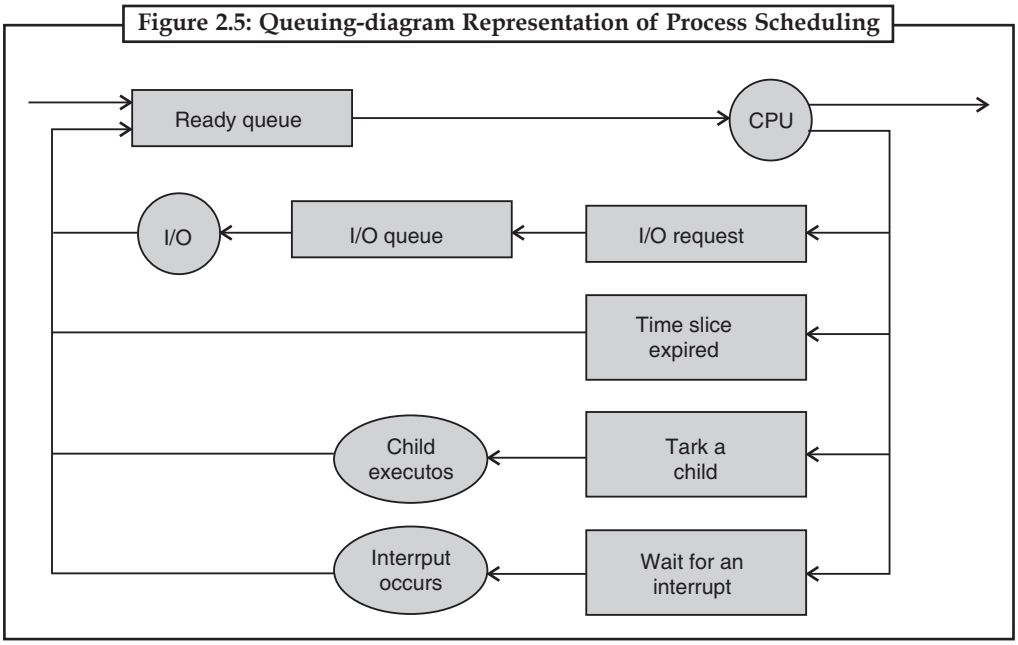
A common representation of process scheduling is a queueing diagram, such as that in Figure 2.5. Each rectangular box represents a queue. Two types of queues are present—the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of several events could occur:

Notes

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Figure 2.5: Queuing-diagram Representation of Process Scheduling



In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

2.3.2 Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

In a batch system, often more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them. The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the brief time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (or wasted) simply for scheduling the work.

The long-term scheduler, on the other hand, executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the degree of multiprogramming—the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate

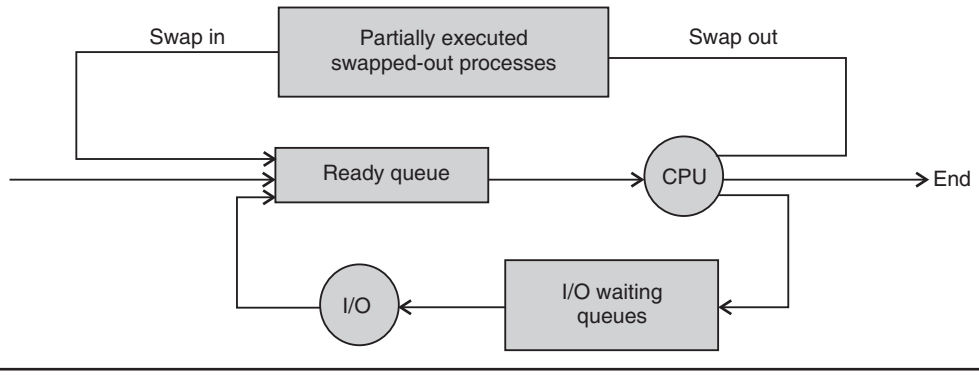
Notes

of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An 110-bound process spends more of its time doing I/O than it spends doing computations. **A CPU-bound** process, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses. The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX often have no long-term scheduler, but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If the performance declines to unacceptable levels, some users will simply quit. Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler, diagrammed in Figure 2.6, removes processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

Figure 2.6: Addition of Medium-term Scheduling to the Queueing Diagram



Task Give the process of PCB with all step.

2.3.3 Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process

Notes

scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds. Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch simply includes changing the pointer to the current register set. Of course, if active processes exceed register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. Advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.



Did u know?

Context switching has become such a performance bottleneck that programmers are using new structures (threads) to avoid it whenever possible.

Self Assessment

Multiple choice questions:

1. PCB = ...

   (*a*) Program Control Block (*b*) Process Control Block

   (*c*) Process Communication Block (*d*) None of the above

2. FIFO scheduling is ..... .

   (*a*) Preemptive Scheduling (*b*) Non Preemptive Scheduling

   (*c*) Deadline Scheduling (*d*) Fair share scheduling

3. Switching the CPU to another process requires to save state of the old process and loading new process state is called as ..... .

   (*a*) Process Blocking (*b*) Context Switch

   (*c*) Time Sharing (*d*) None of the above

4. The state of a process after it encounters an I/O instruction is ..... .

   (*a*) Ready (*b*) Blocked/Waiting

   (*c*) Idle (*d*) Running

2.4 Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent. On the other hand, a process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Notes

We may want to provide an environment that allows process cooperation for several reasons:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.

To illustrate the concept of cooperating processes, let us consider the producer-consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

The unbounded-buffer producer-consumer problem places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded-buffer producer consumer problem assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The buffer may either be provided by the operating system through the use of an Inter Process Communication (IPC) facility or by explicitly coded by the application programmer with the use of shared memory. Let us illustrate a shared-memory solution to the bounded-buffer problem. The producer and consumer processes share the following variables:

```
#define BUFFER-SIZE 10
typedef s t r u c t {
. . .
) item;
item buffer [BUFFER-SIZE] ;
int in = 0;
i n t out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers—in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out ; the buffer is full when ((in + 1) % BUFFERSIZE)

Notes

== out. The code for the producer and consumer processes follows. The producer process has a local variable nextproduced in which the new item to be produced is stored:

```
while (1) {
/* produce an item in nextproduced */
while ( ( (in + 1) % BUFFER-SIZE) == out)
; /* do nothing */
buffer [in] = nextproduced;
in = (in + 1) % BUFFER-SIZE;
1
```

The consumer process has a local variable next consumed in which the item to be consumed is stored:

```
while (I) {
while (in == out)
; // do nothing
nextconsumed = buffer [out] ;
out = (out + 1) % BUFFER-SIZE;
/* consume the item in nextconsumed */
1
```

2.5 Overview of Inter-Process Communication

In the previous section, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a common buffer pool, and that the code for implementing the buffer be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via an inter-process communication (PC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network. An example is a chat program used on the World Wide Web. IPC is best provided by a message-passing system, and message systems can be defined in many ways. In this section, we look at different issues when designing message-passing systems.

2.5.1 Message-passing System

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. We have already seen message passing used as a method of communication in Microkernels. In this scheme, services are provided as ordinary user processes. That is, the services operate outside of the kernel. Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least the two operations— send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not

Notes

with the link's physical implementation but rather with its logical implementation. Here are several methods for logically implementing a link and the send/receive operations:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

We look at each of these types of message systems next.

2.5.2 Naming

Processes that want to communicate must have a way to refer to each other.

They can use either direct or indirect communication.

2.5.3 Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as:

- `send(P, message) –Send a message to process P.`
- `receive(Q, message) –Receive a message from process Q.`

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send and receive primitives are defined as follows:

- `send(P, message) –Send a message to process P.`

receive (id, message)—Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

The disadvantage in both symmetric and asymmetric schemes is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

2.5.4 Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The send and receive primitives are defined as follows:

- `send(A, message) –Send a message to mailbox A.`
- `receive(A, message) –Receive a message from mailbox A.`

Notes

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while P2 and P3 each execute a receive from A. Which process will receive the message sent by P1? The answer depends on the scheme that we choose:

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a receive operation.
- Allow the system to select arbitrarily which process will receive the message.

(that is, either P2 or P3, but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

On the other hand, a mailbox owned by the operating system is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

2.5.5 Synchronization

Communication between processes takes place by calls to send and receive primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking—also known as synchronous and asynchronous.

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send:** The sending process sends the message and resumesoperation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Nonblocking receive:** The receiver retrieves either a valid message or a null.

Notes

Different combinations of send and receive are possible. When both the send and receive are blocking, we have a rendezvous between the sender and the receiver.

2.5.6 Buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implemented in three ways:

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.


Did u know?

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

2.5.7 An Example: Mach

As an example of a message-based operating system, consider the Mach operating system, developed at Carnegie Mellon University. The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control. Most communication in Mach—including most of the system calls and all intertask information—is carried out by messages. Messages are sent to and received from mailboxes, called ports in Mach.

Even system calls are made by messages. When each task is created, two special mailboxes—the Kernel mailbox and the Notify mailbox—are also created. The kernel uses the Kernel mailbox to communicate with the task. The kernel sends notification of event occurrences to the Notify port. Only three system calls are needed for message transfer. The msg-send call sends a message to a mailbox. A message is received via msgxeceive. Remote procedure calls (RPCs) are executed via msg-rpc, which sends a message and waits for exactly one return message from the sender. In this way, RPC model a typical subroutine procedure call, but can work between systems.

The port-allocate system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's owner. The owner also is given receive access to the mailbox. Only one task at a time can either own or receive from a mailbox, but these rights can be sent to other tasks if desired.

The mailbox has an initially empty queue of messages. As messages are sent to the mailbox, the messages are copied into the mailbox. All messages have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order, but does not guarantee an absolute ordering. For instance, messages sent from each of two senders may be queued in any order.

The messages themselves consist of a fixed-length header, followed by a variable-length data portion. The header includes the length of the message and two mailbox names. When a message is sent, one mailbox name is the mailbox to which the message is being sent. Commonly, the sending thread expects a reply; the mailbox name of the sender is passed on to the receiving task, which may use it as a "return address" to send messages back.

Notes

The variable part of a message is a list of typed data items. Each entry in the list has a type, size, and value. The type of the objects specified in the message is important, since operating-system-defined objects-such as the ownership or receive access rights, task states, and memory segments-may be sent in messages.

The send and receive operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.
2. Wait at most n milliseconds.
3. Do not wait at all, but rather return immediately.
4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a notification message is sent back to the sender; only one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, these tasks may need to send a one-time reply to the task that had requested service, but must also continue with other service requests, even if the reply mailbox for a client is full. The receive operation must specify from which mailbox or mailbox set to receive a message. A mailbox set is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive from only a mailbox or mailbox set for which that task has receive access. A port-status system call returns the number of messages in a given mailbox. The receive operation attempts to receive from either of the following:

1. Any mailbox in a mailbox set
2. A specific (named) mailbox


Caution

If no message is waiting to be received, the receiving thread may wait, wait at most n milliseconds, or not wait.

The Mach message system attempts to avoid double-copy operations by using virtual-memory-management techniques. Essentially, Mach maps the address space containing the sender's message into the receiver's address space. The message itself is never actually copied. This message-management technique provides a large performance boost, but works only for intrasystem messages.

2.5.8 An Example: Windows 2000

The Windows 2000 operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows 2000 provides support for multiple operating environments or subsystems, with which application programs communicate via a message-passing mechanism. The application programs can be considered to be clients of the Windows 2000 subsystem server.

The message-passing facility in Windows 2000 is called the local procedure call (LPC) facility. The LPC in Windows 2000 communicates between two processes that are on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows 2000. Like Mach, Windows 2000 uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows 2000 uses two types of ports—connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named objects, which are

Notes

visible to all processes; they give applications a way to set up a communication channel. This communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports, and returns the handle to one of them to the client.


Task Give the current Windows name.

If a client needs to send a larger message, it passes the message through a section object (or shared memory). The client has to decide, when it sets up the channel, whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Likewise, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about that section object. This method is more complicated than the first method, but it avoids the data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.


Did u know?

Windows 2000 uses three types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.


Case Study

How could a system be designed to allow a choice of operating systems to boot from? What would the bootstrap program need to do?

Consider a system that would like to run both Windows XP and three different distributions of Linux (e.g., RedHat, Debian, and Mandrake). Each operating system will be stored on disk. During system boot-up, a special program (which we will call the boot manager manager) will determine which operating system to boot into. This means that rather initially booting to an operating system, the boot manager will first run during system start-up. It is this boot manager that is responsible for determining which system to boot into. Typically boot managers must be stored at certain locations of the hard disk to be recognized during system start-up. Boot managers often provide the user with a selection of systems to boot into; boot managers are also typically designed to boot into a default operating system if no choice is selected by the user.

Questions:

1. Explain process of Bootstrap loader.
2. Define working of Bootstrap loader in the term of process management.

Notes



Lab Exercise

1. Write an Algorithm for PCB concept.
2. Give the step for intercrosses communication.

Self Assessment

Fill in the blanks:

5. Any process that shares data with other processes is a ..... process.
6. Message sent by a process can be of either fixed or ..... size.
7. A ..... is associated with more than two processes.
8. A ..... owned by the operating system is independent.

True or False:

9. The Mach kernel supports the creation and destruction of multiple task.
10. Window 2000 uses two types of message passing techniques over a port.

2.6 Summary

- A process is a sequential program in execution. A process migrates between the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.
- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

2.7 Keywords

Buffering: A buffer is a temporary storage location for data while the data is being transferred.

Context Switch: A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) from one process or thread to another.

Cooperating Processes: Processes can cooperate with each other to accomplish a single task. Cooperating processes can:

- Improve performance by overlapping activities or performing work in parallel.
- Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program.

CPU Registers: The central processing unit (CPU) contains a number of memory locations which are individually addressable and reserved for specific purpose. These memory locations are called **registers**.

'Inter-process Communication' (IPC): In computing, '**Inter-process communication'** (**IPC**) is a set of techniques for the exchange of data among multiple threads in one or more processes.

Notes

Message-Passing System: Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and interprocess communication.

Process Control Block (PCB): The PCB is a certain store that allows the operating systems to locate key information about a process.

Process Counter: Program instructions uniquely identified by their program counters (PCs) provide a convenient and accurate means of recording the context of program execution and PC-based prediction techniques have been widely used for performance optimizations at the architectural level.

Process Management: The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated in a process.

Process Scheduling: The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling.

Process State: The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily.

Synchronization: In computer science, especially parallel computing, synchronization means the coordination of simultaneous threads or processes to complete a task in order to get correct runtime order and avoid unexpected race conditions.

Thread: A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams.

2.8 Review Questions

1. What is a process?
2. What about process states?
3. What is a process control block?
4. How do processes inter-communicate?
5. How do processes synchronize their activity?
6. How do processes protect critical data (Critical sections)?
7. Consider the interprocess-communication scheme where mailboxes are used:
   (*a*) Suppose a process *P* wants to wait for two messages, one from mailbox A and one from mailbox B. What sequence of **send** and **receive** should it execute?
   (*b*) What sequence of **send** and **receive** should *P* execute if *P* wants to wait for one message from mailbox A or from mailbox B (or from both)?
8. What are the benefits and the detriments of each of the following? Consider both the systems and the programmers' levels.
   (*a*) Direct and indirect communication
   (*b*) Symmetric and asymmetric communication
   (*c*) Automatic and explicit buffering

Notes

(*d*) Send by copy and send by reference

(*e*) Fixed-sized and variable-sized messages

9. Describe the actions taken by a kernel to switch context between processes.
10. Write a socket-based Fortune Teller server. Your program should create a server that listens to a specified port. When a client receives a connection, the server should respond with a random fortune chosen from its database of fortunes.
11. Describe the actions used in Buffering in the processes.
12. Describe about the process scheduling in the operating system.
13. How do processes interprocess communication?
14. What are the benefits and the detriments of Cooperating process.
15. Describe the Process States in operating system.

Answers to Self Assessment

1. (*b*) 2. (*b*) 3. (*b*) 4. (*b*) 5. cooperating
6. Variable 7. link 8. mailbox 9. True 10. False

2.9 Further Readings



Books

Operating Systems, by Harvey M. Deitel , Paul J. Deitel, David R. Choffnes.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.



Online link

wiley.com/coolege.silberschatz

Notes

Unit 3: Process Management-II

CONTENTS

Objectives

Introduction

3.1 Concept of Threads

3.1.1 Processes vs Threads

3.1.2 Why Threads?

3.1.3 User-level Threads

3.1.4 Kernel-level Threads

3.1.5 Advantages of Threads over Multiple Processes

3.1.6 Disadvantages of Threads over Multiple Processes

3.1.7 Application that Benefits from Threads

3.1.8 Application that cannot Benefit from Threads

3.1.9 Resources Used in Thread Creation and Process Creation

3.2 Context Switch

3.2.1 Major Steps of Context Switching

3.2.2 Action of Kernel to Context Switch among Threads

3.2.3 Action of Kernel to Context Switch among Processes

3.3 Multi-threading in IDL

3.3.1 Multi-threading

3.3.2 Multi-threading to IDL

3.3.3 Modeling Multi-threaded Processors

3.3.4 Single-threaded versus Multi-threaded Symmetric Multiprocessing

3.3.5 Performance Scaling in Multi-threaded Systems

3.3.6 Team Quest Model Reporting Changes

3.3.7 Context Switches and Mode Switches

3.3.8 The Cost of Context Switching

3.3.9 Monitoring Context Switches

Notes

3.3.10 Context Switching Perform

3.3.11 Super OS Context Switching in Power Managed Environment

3.4 Scheduling Criteria

3.5 Types of Scheduling

3.5.1 Long Term Scheduling

3.5.2 Medium Term Scheduling

3.5.3 Short Term Scheduling

3.6 Scheduling Algorithms

3.6.1 Conventional Time-share Scheduling

3.6.2 Real-time Scheduling

3.6.3 Real-time Scheduling Algorithms Off-line/Pre-run-time Scheduling

3.6.4 On-line Scheduling Algorithms

3.6.5 Non-Preemptive Static Priority Algorithms

3.6.6 Preemptive Static Priority-based Algorithms

3.6.7 Dynamic Planning-based Algorithms

3.6.8 Dynamic Best Effort Algorithms

3.6.9 Equal Request Times

3.6.10 Equal Execution Times

3.6.11 Monotonic Absolute Deadlines

3.6.12 Equal Relative Deadlines

3.6.13 Equal Absolute Deadlines

3.7 An Overview of Thread Scheduling

3.7.1 Priority-based Scheduling

3.7.2 The Scheduling Process

3.8 Priority Exceptions

3.8.1 Complex Priorities

3.9 Summary

3.10 Keywords

3.11 Review Questions

3.12 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Explain concept of threads
- Discuss introduction to multi-threading in IDL
- Understand context switching
- Explain scheduling criteria
- Understand types of scheduling
- Explain scheduling algorithms
- Understand overview of thread scheduling

Introduction

A process is a program in execution. A process must have system resources, such as memory and the underlying CPU. The kernel supports the illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute. On a multiprocessor, multiple processes may really execute concurrently. This unit describes the composition of a process, the method that the system uses to switch between processes, and the scheduling policy that it uses to promote sharing of the CPU. It also introduces process creation and termination, and details the signal facilities and process-debugging facilities.

3.1 Concept of Threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

A thread is a single sequence stream within a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process, i.e. process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space.


Did u know?

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Notes

3.1.1 Processes vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

3.1.1.1 Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

3.1.1.2 Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task.
- Unlike processes, threads are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

3.1.2 Why Threads?

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads makes a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

3.1.2.1 Threads are Cheap in the Following Sense:

1. They only need a stack and storage for registers therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3. Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free—the biggest drawback is that there is no protection between threads.

3.1.3 User-level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Notes

3.1.3.1 Advantages

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are:

- User-level threads does not require modification to operating systems.
- **Simple Representation:** Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- **Simple Management:** This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- **Fast and Efficient:** Thread switching is not much more expensive than a procedure call.

3.1.3.2 Disadvantages

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads require non-blocking systems call, i.e. a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

3.1.4 Kernel-level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

3.1.4.1 Advantages

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

3.1.4.2 Disadvantages

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

3.1.5 Advantages of Threads over Multiple Processes

- **Context Switching:** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- **Sharing:** Threads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file, etc.

Notes

3.1.6 Disadvantages of Threads over Multiple Processes

- **Blocking:** The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.
- **Security:** Since there is an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

3.1.7 Application that Benefits from Threads

A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi-threaded process. In general, any program that has to do more than one task at a time could benefit from multitasking. For example, a program that reads input, process it, and outputs could have three threads, one for each task.

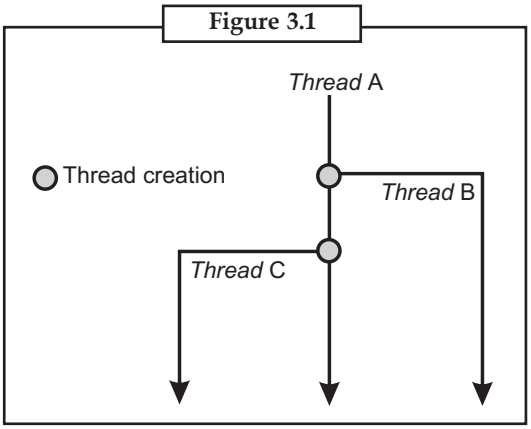
3.1.8 Application that cannot Benefit from Threads

Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes. For example, a program that displays the time of the day would not benefit from multiple threads.

3.1.9 Resources Used in Thread Creation and Process Creation

When a new thread is created it shares its code section, data section and operating system resources like open files with other threads. But it is allocated its own stack, register set and a program counter.

Figure 3.1



The creation of a new process differs from that of a thread mainly in the fact that all the shared resources of a thread are needed explicitly for each process. So though two processes may be running the same piece of code they need to have their own copy of the code in the main memory to be able to run. Two processes also do not share other resources with each other. This makes the creation of a new process very costly compared to that of a new thread.


Caution

A virus is also a thread in operating system.

3.2 Context Switch

To give each process on a multiprogrammed machine a fair share of the CPU, a hardware clock generates interrupts periodically. This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals. Each time a

Notes

clock interrupt occurs, the interrupt handler checks how much time the current running process has used. If it has used up its entire time slice, then the CPU scheduling algorithm (in kernel) picks a different process to run. Each switch of the CPU from one process to another is called a context switch.

3.2.1 Major Steps of Context Switching

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

In a multiprogrammed uniprocessor computing system, context switches occur frequently enough that all processes appear to be running concurrently. If a process has more than one thread, the Operating System can use the context switching technique to schedule the threads so they appear to execute in parallel. This is the case if threads are implemented at the kernel level. Threads can also be implemented entirely at the user level in run-time libraries. Since in this case no thread scheduling is provided by the Operating System, it is the responsibility of the programmer to yield the CPU frequently enough in each thread so all threads in the process can make progress.

3.2.2 Action of Kernel to Context Switch among Threads

The threads share a lot of resources with other peer threads belonging to the same process so a context switch among threads for the same process is easy. It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplish this task.

3.2.3 Action of Kernel to Context Switch among Processes

Context switches among processes are expensive. Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the following information:

- The process state.
- The program counter, PC.
- The values of the different registers.
- The CPU scheduling information for the process.
- Memory management information regarding the process.
- Possible accounting information for this process.
- I/O status information of the process.

When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.


Caution

Before a process can be switched its process control block (PCB) must be saved by the operating system.

3.3 Multi-threading in IDL

ITT-VIS has added support for using threads internally in IDL to accelerate specific numerical computations on multi-processor systems. Multi-processor capable hardware has finally become

Notes

cheap and widely available. Most operating systems have support for SMP (Symmetric Multi-Processing), and IDL users are beginning to own such hardware. In the future, it is reasonable to imagine that most machines will have multiple CPUs.

The multi-threading capability, first added in IDL 5.5, applies to binary and unary operators, many core mathematical functions, and a number of image processing, array manipulation and type conversion routines. Although performance results will vary, the execution time of these computations can be significantly reduced on systems with multiple processors. The ability to exploit multiple CPUs will become very important in coming years, and the list of threaded routines is expected to grow with each release of IDL. IDL users should be aware that ITT-VIS offers a Global Services Group (GSG) that can be hired to help optimize user-written code or to parallelize specific algorithms beyond those that use the thread pool. The interface for controlling the IDL thread pool is simple, allowing immediate and measurable benefits with little effort. In addition, the IDL thread pool is safe and transparent on platforms that are unable to support threading. Those platforms that can benefit will use threads, and those that cannot will continue to produce correct results using a single thread, and with the same level of performance as previous versions of IDL. This unit provides background and motivation for IDL's multi-threading capability.

3.3.1 Multi-threading

The concept of multi-threading involves an operating system that is multi-thread capable allowing programs to split tasks between multiple execution threads. On a machine with multiple processors, these threads can execute concurrently, potentially speeding up the task significantly. Mathematical computations on large amounts of scientific data can be quite intensive and are ideal candidates for threading on systems with multiple CPUs.

The most common type of program is the single-threaded program. IDL has traditionally been single-threaded. When the program runs, this single thread starts at the main() function in the program, and runs until it either exits, or performs an illegal operation and is killed by the operating system. Since it is the only thread, it knows that anything that happens in this program is caused solely by it. Most modern operating systems time slice between various programs, so at any given time, a single thread is either running or sleeping. There are two reasons why it may be sleeping—It is waiting for a needed, but currently unavailable resource (e.g. memory, data (input, output)...). The operating system is letting some other program run. This time slicing, which is usually preemptive multitasking, happens so quickly that the end user is fooled into thinking that everything is running simultaneously.

To move from single-threaded to multi-threaded (MT) programs requires a small conceptual generalization of the above. Instead of having only a single thread, we allow more than one thread in a single process. Each thread has its own program counter and stack, and is free to run, unimpeded by any other thread in the same program. All threads in the same program share any other resources, including code, data, and open files. The operating system still schedules which threads run and when, but instead of scheduling by process, it now schedules the individual threads within a process. If your system has a single CPU, preemptive multitasking still gives the illusion that more than one thing is going on simultaneously. If the system has more than one CPU, then more than one thread can be running on the system at a given time. It is even possible that more than one thread within a given program will run simultaneously. This is actual simultaneous execution, not the mere illusion of it as with a uniprocessor. It is important to realize that the software concept of threading is an abstraction provided by the operating system, and it is available whether or not the underlying hardware has multi-processing (MP) capabilities. It is reasonable to run a MT program on a uniprocessor, unless your program requires actual simultaneous execution of multiple threads to work properly. For example, a program might use a thread to wait for incoming data from a slow source, while other threads

Notes

manage the user interface and perform other tasks. Multi-processor hardware is not necessary for such a program. In contrast, if you are using threads to speed up a numerical computation, you will require actual MP hardware to see any benefit. On a uniprocessor, this program will work harder (MT code adds overhead) and will take essentially the same amount of time to complete as a single-threaded version. Common sense suggests that threading does not make a uniprocessor able to compute any faster.

3.3.2 Multi-threading to IDL

Simply put, ITT-VIS has implemented multi-threading in IDL in order to allow users to harness additional CPUs to do more work in less time. Scientific data sets continue to grow in size faster than computers can process them. Multi-processors offer one way to handle larger problems.

Multi-processor hardware and Symmetric Multi-Processing (SMP) have become cheap and easily available. There are some powerful trends driving this change:

1. As transistor densities on processor chips increase with each generation, there is room for replicated processing units.
2. At any given point in time, the cost of the second most powerful CPU in production is much lower than the most powerful CPU. It makes sense that if you can harness multiple cheap, but only slightly less powerful, CPUs, you can do more work for less money.
3. There are physical limits that govern how fast a single CPU can possibly go, and we expect to hit those limits within a few (10-20, max) years. Once we hit this limit, the only way to increase computing power may be to add CPUs.

The development of SMP systems has been driven not by the need to run multi-threaded programs, but by a need to increase throughput on servers that run multiple single-threaded programs simultaneously (e.g., to serve files, mail and printing). Economies of scale allow computer vendors to apply this technology to desktop machines. It is becoming common for individuals to have such machines, and it appears that this trend will continue.

3.3.3 Modeling Multi-threaded Processors

Symmetric multi-processing (SMP) has been employed by computer makers for some time. Multiple processors are connected to a common memory pool and a combination of hardware and operating system functions permit work to be balanced across the entire unit. Each processor had a single "thread" that processed programming instructions.

Recently chip manufacturers added additional threads to the processors to further increase efficiency of the chips. The added capacity of multi-threaded processors is welcomed but they provide some challenges for the capacity planner. As work is added to a system, multi-threaded CPU cores perform differently from multiple single-threaded CPU cores in a symmetric multi-processing (SMP) environment. The meaning of per-process CPU time measurements depend upon chip technology, therefore, measurement results and expected future performance may not be intuitively obvious any more.

This paper explains a conceptual architecture developed for modeling multi-threaded processors and the new functionality provided in Team Quest Model to support multi-threaded processors. A modeling example using Linux on Intel chips will be provided to aid your understanding.

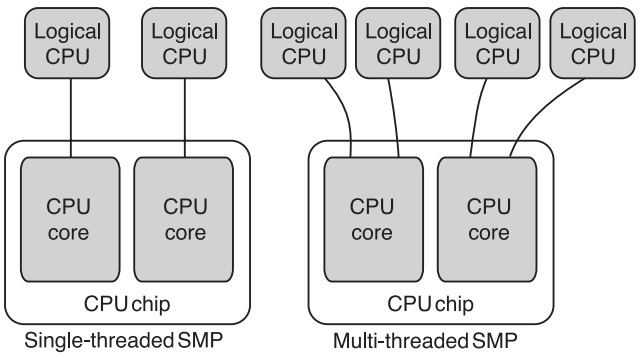
3.3.4 Single-threaded versus Multi-threaded Symmetric Multiprocessing

Let's begin by comparing older single-threaded SMP technology with newer multi-threaded technologies. In both cases, the CPU chip is the hardware component that provides the instruction processing capability. Within each chip there may be multiple CPU cores, and each core

Notes

contains the complete functionality of what we used to consider a CPU. In traditional symmetric multiprocessing, each CPU core supports a single hardware instruction thread that interfaces with the operating system (diagram on left in Figure 3.2). When activating multi-threading, each core supports multiple hardware instruction threads, each interfacing with the operating system. Each hardware instruction thread is recognized by the operating system as a logical CPU.

Figure 3.2: Single-threaded SMP and Multi-threaded SMP



Older SMP systems exhibited performance limitations as more CPU's were added to a configuration. For those of us familiar with the history of mainframe computers, we saw that each incremental processor added a lesser amount of additional capacity. In fact, one vendor, Amdahl Corporation, increased the computational power of the last two processors in their 12-way computer in order to overcome the SMP shortfall. These limitations resulted from hardware and operating system architectures designed to ensure data integrity through the use of various tactics such as signaling and locks. Over the years, all the major vendors have made significant improvements in this area. As a result, most SMP systems today have near linear performance scaling in the hardware and operating systems. In a multiprocessing architecture, there are two approaches to providing additional processing power. Each additional core, bearing a single logical CPU, delivers a nearly equal quantity of CPU capacity. In most of today's architectures, this results in a commensurate increase in capacity when cores are added. The multi-threading option adds multiple threads to each core. Each thread adds some additional amount of CPU capacity. However, because these threads share the CPU core resources, the addition of a thread typically delivers only a portion of the capacity of a single-threaded core.

Examples of multi-threaded chips include Sun UltraSPARC T1 and T2, SPARC64 VII, Intel Xeon, Intel Itanium2, Intel Pentium 4, IBM POWER5 and IBM POWER6.



Did u know?

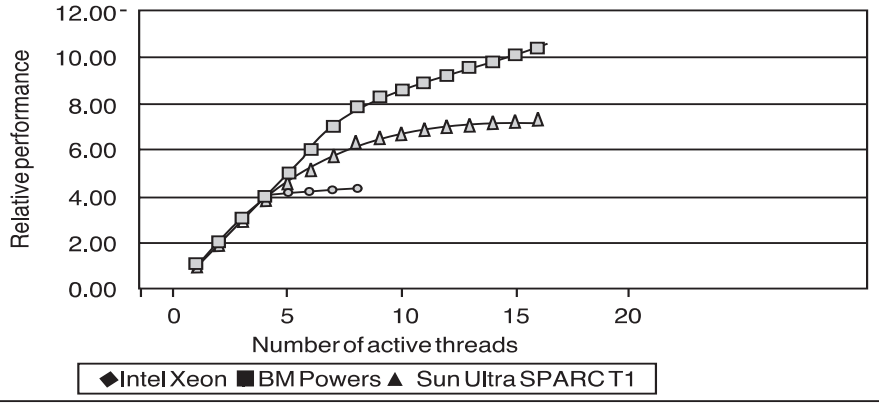
Multi-processor hardware and Symmetric Multi-Processing (SMP) have become cheap and easily available. There are some powerful trends driving this change.

3.3.5 Performance Scaling in Multi-threaded Systems

When more threads are added to cores in multi-threaded systems, performance depends upon chip technologies. All deviate from a linear growth line graph once you get beyond the point where a single thread is active on each core and core resources are shared. Chip performance differences as seen during Team Quest testing can be seen in Figure 3.3.

Notes

Figure 3.3: Comparison or Multi-threaded System



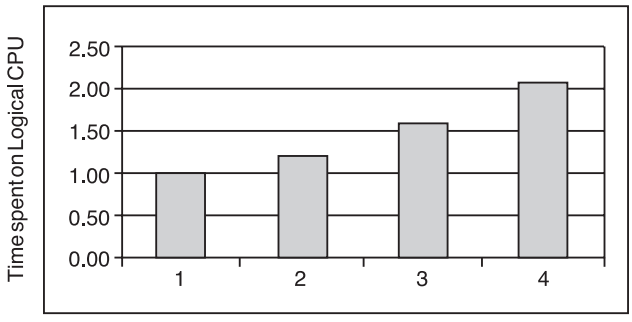
Intel Xeon (2 chips, 2 cores per chip, 2 threads per core) delivers minimal performance gain once four threads are exceeded. Sun UltraSPARC T1 (1 chip, 4 cores per chip, 4 threads per core) shows a linear increase in performance up to four threads, slightly degraded performance when there are two threads per core active, and then only nominal gain after core sharing increases as more than two threads become active.

IBM POWER5 (4 chips, 2 cores per chip, 2 threads per core), shows linear gain up to eight threads (one per core) and then the gain per thread drops from that point forward. Where performance becomes non-linear, it is because more than one thread has become active on a CPU core.

Similarly, when you view transaction behavior with multi-threaded CPUs you see some interesting results. Figure 3.4 displays testing results from a transaction that executes a fixed number of instructions and a CPU core that supports four threads per core. If only one thread is active, each transaction will complete in one second. If two threads are active per core it will take 1.25 seconds for the same transaction. If three threads are active, each transaction will take about 1.6 seconds. If four threads are active, each transaction will take about 2.1 seconds. The behavior of transaction, therefore, depends on how many simultaneous logical CPU threads are active on a core.

The results show that the best performance for a single transaction comes when there is only one CPU hardware thread is active on the core on which it is consuming resources.

Figure 3.4: Active Threads per Core



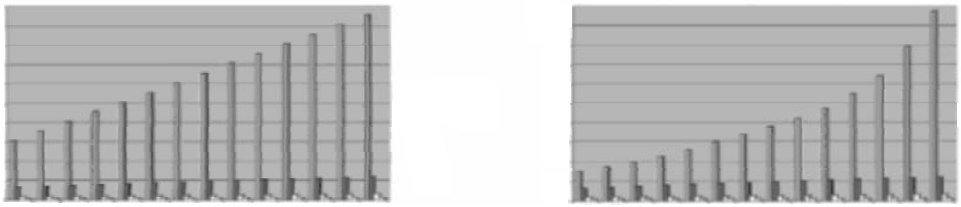
Notes

However, even though each logical CPU runs slower when multiple threads (logical CPUs) are active, there is a greater total capacity. It is the same whether two active threads on a core or more. The best throughput (transactions completed per second) occurs when all CPU hardware threads are active.

3.3.6 Team Quest Model Reporting Changes

In a traditional single-threaded SMP CPU architecture, the Active Resource Statistic Service Time is always equal to the Effective Service Time. Therefore, when there are not any other bottlenecks in the system the CPU utilization will grow at a linear rate.

Figure 3.5 (*a*): Single-threaded Architecture Figure 3.5 (*b*): Multi-threaded Architecture



This is no longer true with a multi-threading CPU architecture. When the CPU has more than a minimal utilization, the Effective Service Time will sometimes be greater than the Service Time. Figure 3.5(*b*) shows how multi-threading deviates from a traditional SMP straight line trend line (Figure 3.5(*a*)).

In the case of this Solaris system using a Sun Ultra SPARC T1 processor with four threads per core, the service time is indeed less than the effective service time. At low utilizations when there is only one thread per core active, however, it is likely that the service time will equal the effective service time and the utilization growth will be linear until more than one thread per core is active but as the processor gets busier and more threads become active within the core, effective service will get progressively larger than the service time and the utilization growth will be larger than linear.

3.3.7 Context Switches and Mode Switches

Context switches can occur only in kernel mode. Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. Other programs, including applications, initially operate in user mode, but they can run portions of the kernel code via system calls. A system call is a request in a like operating system by an active process (i.e., a process currently progressing in the CPU) for a service performed by the kernel, such as input/output (I/O) or process creation (i.e., creation of a new process). I/O can be defined as any movement of information to or from the combination of the CPU and main memory (i.e. RAM), that is, communication between this combination and the computer's users (e.g., via the keyboard or mouse), its storage devices (e.g., disk or tape drives), or other computers.

The existence of these two modes in Unix-like operating systems means that a similar, but simpler operation is necessary when a system call causes the CPU to shift to kernel mode. This is referred to as a mode switch rather than a context switch, because it does not change the current process.

Context switching is an essential feature of multitasking operating systems. A multitasking operating system is one in which multiple processes execute on a single CPU seemingly simultaneously and without interfering with each other. This illusion of concurrency is achieved

Notes

by means of context switches that are occurring in rapid succession (tens or hundreds of times per second). These context switches occur as a result of processes voluntarily relinquishing their time in the CPU or as a result of the scheduler making the switch when a process has used up its CPU time slice.

A context switch can also occur as a result of a hardware interrupt, which is a signal from a hardware device (such as a keyboard, mouse, modem or system clock) to the kernel that an even (e.g., a key press, mouse movement or arrival of data from a network connection) has occurred.

Intel 80386 and higher CPUs contain hardware support for context switches. However, most modern operating systems perform software context switching, which can be used on any CPU, rather than hardware context switching in an attempt to obtain improved performance. Software context switching was first implemented in Linux for Intel-compatible processors with the 2.4 kernel.

One major advantage claimed for software context switching is that, whereas the hardware mechanism saves almost all of the CPU state, software can be more selective and save only that portion that actually needs to be saved and reloaded. However, there is some question as to how important this really is in increasing the efficiency of context switching. Its advocates also claim that software context switching allows for the possibility of improving the switching code, thereby further enhancing efficiency, and that it permits better control over the validity of the data that is being loaded.

3.3.8 The Cost of Context Switching

Context switching is generally computationally intensive. That is, it requires considerable processor time, which can be on the order of nanoseconds for each of the tens or hundreds of switches per second. Thus, context switching represents a substantial cost to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system.

Consequently, a major focus in the design of operating systems has been to avoid unnecessary context switching to the extent possible. However, this has not been easy to accomplish in practice. In fact, although the cost of context switching has been declining when measured in terms of the absolute amount of CPU time consumed, this appears to be due mainly to increases in CPU clock speeds rather than to improvements in the efficiency of context switching itself.

One of the many advantages claimed for Linux as compared with other operating systems, including some other Unix-like systems, is its extremely low cost of context switching and mode switching.

3.3.9 Monitoring Context Switches

A context switch occurs when the kernel switches the processor from one thread to another—for example, when a thread with a higher priority than the running thread becomes ready. Context switching activity is important for several reasons. A program that monopolizes the processor lowers the rate of context switches because it does not allow much processor time for the other processes' threads. A high rate of context switching means that the processor is being shared repeatedly—for example, by many threads of equal priority. A high context-switch rate often indicates that there are too many threads competing for the processors on the system.



Did u know?

The rate of context switches can also affect performance of multiprocessor computers. For information about how to monitor and tune context-switch activity on multiprocessor systems, see "Measuring Multiprocessor System Activity" in this book.

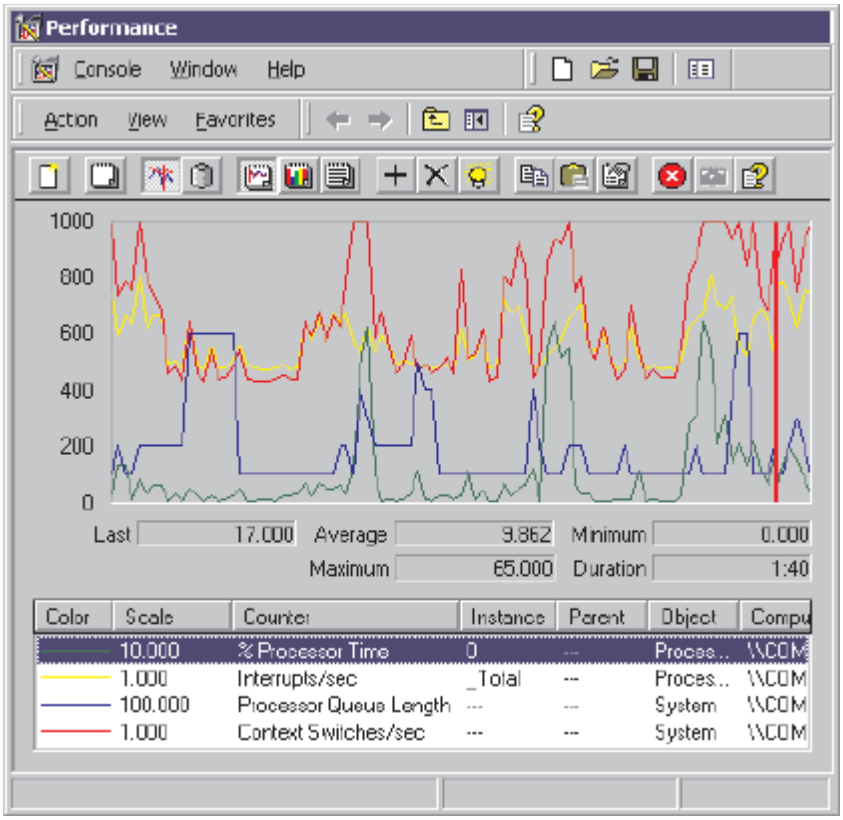
Notes

We can view context switch data in two ways:

- The System\Context Switches/sec counter in System Monitor reports systemwide context switches.
- The Thread (_Total)\Context Switches/sec counter reports the total number of context switches generated per second by all threads.

Although these counters might vary slightly due to sampling, generally they will be nearly equal.

Figure 3.6: System Wide Context Switches During a Processor Bottleneck



In Figure 3.6, Processor (_Total)\% Processor Time jumps to about 60 per cent during the sample interval. System\Processor Queue Length (scaled by a factor of 10), shows that the queue varies from 2 to 6, with a mean near 4. System\Context Switches (shown scaled by a factor of 10), reveals an average of about 750 switches per second. A rate of context switches from 500 to 2,000 per second might indicate a problem with a network adapter or a device driver or that you are using an inefficient server-based application that spawns too many threads.

The Pviewer utility on the on the Windows 2000 operating system CD reports context switch data. For information about installing and using the Windows 2000 Support Tools and Support Tools Help, see the file Sreadme.doc in the \Support\Tools folder of the Windows 2000 operating system CD.

Notes

3.3.10 Context Switching Perform

Typically there are several tasks to perform in a computer system. So if one task requires some I/O operation, you want to initiate the I/O operation and go on to the next task. You will come back to it later.

This act of switching from one process to another is called a "Context Switch".

When you return back to a process, you should resume where you left off. For all practical purposes, this process should never know there was a switch, and it should look like this was the only process in the system.

To implement this, on a context switch, you have to

- save the context of the current process
- select the next process to run
- restore the context of this new process.

What is the context of a process?

- Program Counter
- Stack Pointer
- Registers
- Code + Data + Stack (also called Address Space)
- Other state information maintained by the OS for the process (open files, scheduling info, I/O devices being used, etc.)

All this information is usually stored in a structure called Process Control Block (PCB). All the above has to be saved and restored.

3.3.10.1 What does a context_switch() routine look like?

```
context_switch()
{
        Push registers onto stack
        Save ptrs to code and data.
        Save stack pointer
        Pick next process to execute
        Restore stack ptr of that process /* You have now switched the stack */
        Restore ptrs to code and data.
        Pop registers
        Return
}
```

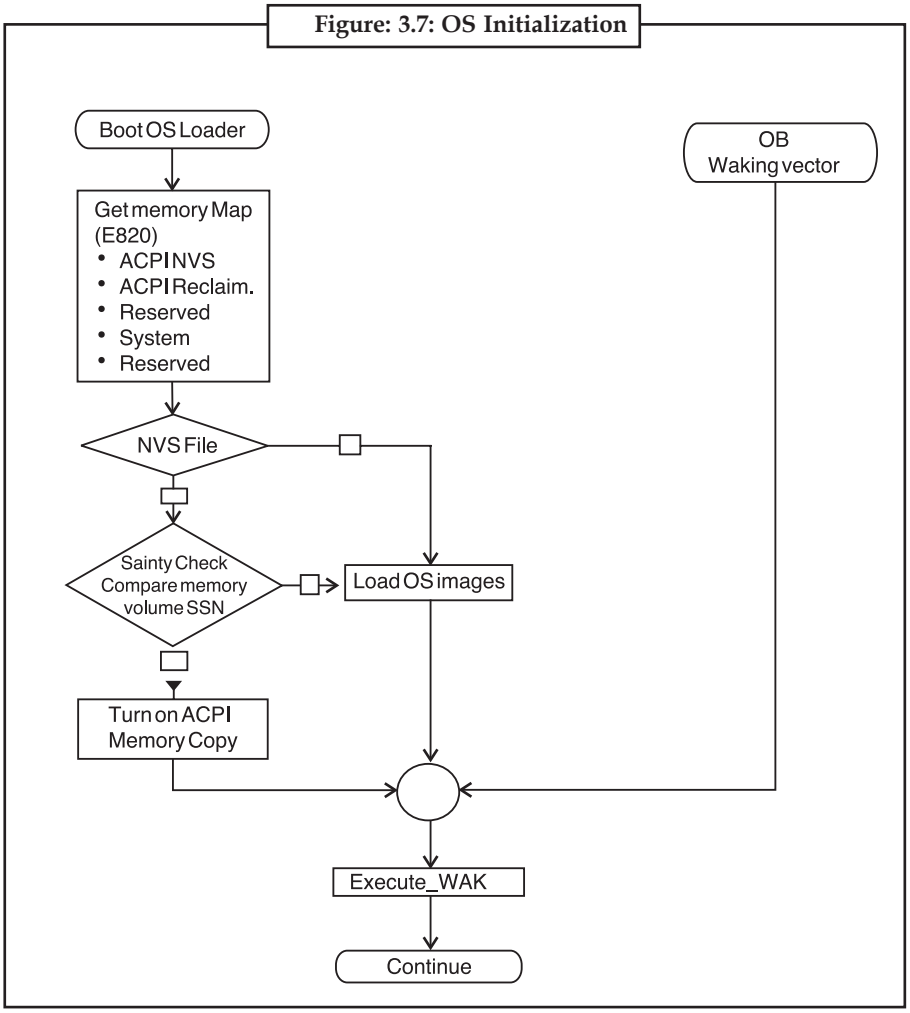
3.3.11 Super OS Context Switching in Power Managed Environment

Super OS context switching consists of a process which suspends the currently operating OS, when and as determined by user and transfer of the systems control to another OS, which may be booting from shut down state (Start) or a previously suspended state (Resume).

Following is the original Power Managed (ACPI, APM) Start (OS Initialization) logic diagram which may follow as a result of external power on or internal OS waking Interrupt.

Notes

Figure: 3.7: OS Initialization



Self Assessment

Multiple choice questions:

1. A thread
   (*a*) is a lightweight process where the context switching is low.
   (*b*) is a lightweight process where the context switching is high.
   (*c*) is used to speed up paging.
   (*d*) none of the above.
2. Process is
   (*a*) program in High level language kept on disk
   (*b*) contents of main memory
   (*c*) a program in execution
   (*d*) a job in secondary memory

Notes

3. Fork is

   (*a*) the dispatching of a task
   (*b*) the creation of a new job
   (*c*) the creation of a new process
   (*d*) increasing the priority of a task

4. The components that process data are located in the:

   (*a*) input devices
   (*b*) output devices
   (*c*) system unit
   (*d*) storage component

5. System software is the set of programs that enables your computer's hardware devices and ..... software to work together.

   (*a*) management
   (*b*) processing
   (*c*) utility
   (*d*) application

3.4 Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU Utilization:** We want to keep the CPU as busy as possible.
- **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround Time:** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting Time:** The CPU scheduling algorithm does not affect the amount of the time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sums of periods spend waiting in the ready queue.
- **Response Time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time. Investigators have suggested that, for interactive systems, it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize varianc.

Notes

3.5 Types of Scheduling

There are three levels for scheduling in Operating Systems:

- Long Term Scheduling
- Short Term Scheduling
- Medium Term Scheduling

3.5.1 Long Term Scheduling

In this level of scheduling, Operating System manage that which process should be admitted to the system; ready or suspended ready queue. OR which process competes for the system resources.

3.5.2 Medium Term Scheduling

Which process should compete for CPU, i.e. scheduling for the process should move from suspended ready state.

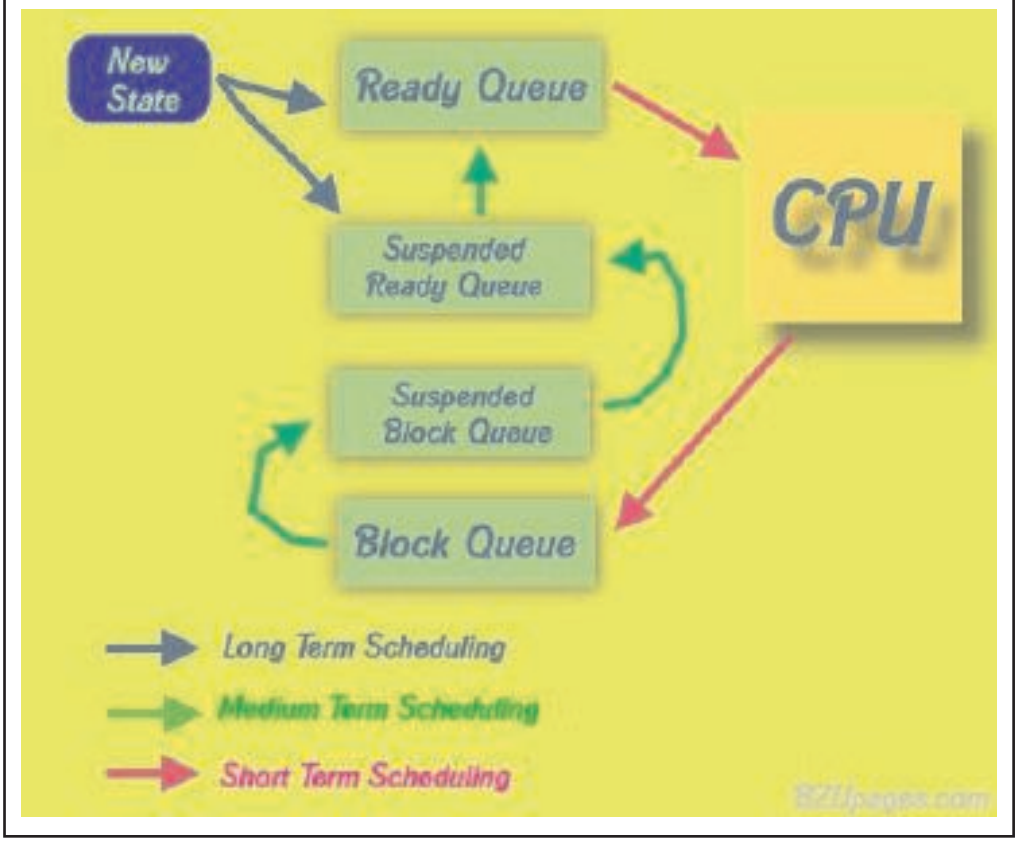
3.5.3 Short Term Scheduling

Which process should assign to the CPU. It is done by the dispatcher, which carry the process from ready queue and assign it to the CPU for processing.



Long Term Scheduling may also known as admission scheduling of Job scheduling.

Figure 3.8: Graphical Representation of Scheduling



Notes

3.6 Scheduling Algorithms

Real-time systems are systems in which correctness depends not only upon the logical correctness of the system but also upon the temporal correctness. If the system produces the correct logical results, but at the wrong time it has failed. Laplante defines a real-time system as "a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure". Examples of real-time systems are robotics controls, weapons guidance systems, anti-lock brakes, flight control systems, and engine and emissions controls. Characteristically, these systems have rigid time requirements on the operation of the processor or the flow of data. The system is normally multi-tasking and uses periodic or aperiodic tasking (or some combination of the two). A real-time system usually has requirements for bounded response time and typically does not utilize secondary storage or virtual memory because of the non-determinism introduced by these techniques. Failure to meet response time constraints results in system failure. In the literature, real-time tasks are usually referred to as having either hard or soft constraints. These are differentiated by the criticality of their on-time completion. A task with hard timing constraints must complete within these constraints or the system fails. Tasks with soft timing constraints that do not complete within their constraints degrade the system but do not cause system failure.

In hard real-time systems, there are two types of processes—periodic and aperiodic (asynchronous). A periodic process is one which is repeatedly executed once each period. An aperiodic process is one that occurs in response to some internal or external event. They are unpredictable in nature but generally have some form of constraint that will allow them to be modeled. Xu and Parnas state that periodic processes with hard deadlines dominate most hard real-time systems. Aperiodic processes are minimal in number and require only minimal CPU time. Additionally, although precise request times for aperiodic processes are not known, usually a minimum amount of time between two consecutive requests can be defined. Today's real-time systems have several drawbacks. Even though timing errors (missed deadlines) are among the most difficult and costly errors to find and correct, many safety-critical hard real-time systems are built using techniques and methodologies that provide no guarantees that timing constraints will be met. The systems are generally based on static designs if the system is predictable. This makes the designs inflexible in operation and difficult and expensive to change. If the systems are not based on static designs, then they tend to lack predictability except under strict conditions that are usually not realistic in real world applications. Next generation real-time systems must be designed and constructed to be dynamic in operation to adjust to changing conditions, predictable in performance so that correctness can be guaranteed and flexible so that changes can be made easily and inexpensively.

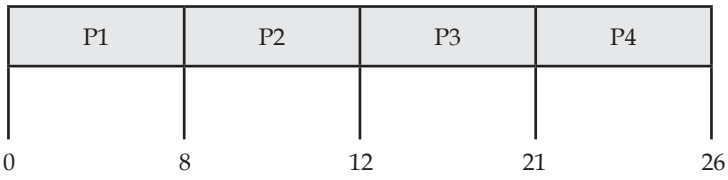


Example: 1

Process	Arrival Time	Service Time
1	0	8
2	1	4
3	2	9
4	3	5

FCFS

Notes



Average wait = ((8 – 0) + (12 –1) + (21 – 2) + 26 – 3))/4 = 61/4 = 15.25

Residence Time at the CPU

5: CPU-Scheduling

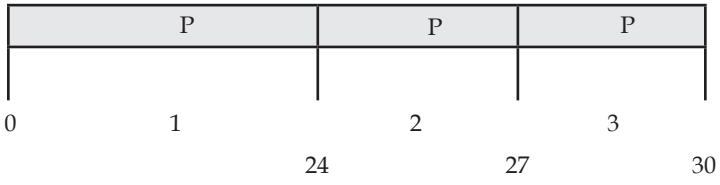
For example, the time slot could be 100 milliseconds. If job1 takes a total time of 250 ms to complete, the round-robin scheduler will suspend the job after 100 ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100 ms each), job1 will get another allocation of CPU time and the cycle will repeat. This process continues until the job finishes and needs no more time on the CPU.

- **Job1 = Total time to complete 250 ms (quantum 100 ms).**
  1. First allocation = 100 ms.
  2. Second allocation = 100 ms.
  3. Third allocation = 100 ms but job1 self-terminates after 50 ms.
  4. Total CPU time of job1 = 250 ms.

Example: 2 **First-Come, First-Served (FCFS) Scheduling**

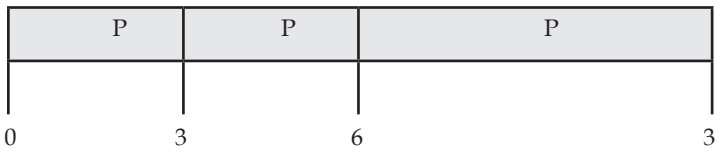
Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: P1, P2, P3. The Gantt Chart for the schedule is:



- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

➢ Suppose that the processes arrive in the order P2, P3, P1. The Gantt chart for the schedule is:



- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

Notes

- Much better than previous case.
- Convoy effect short process behind long process

Example: 3

Shortest-Job-First (SJR) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

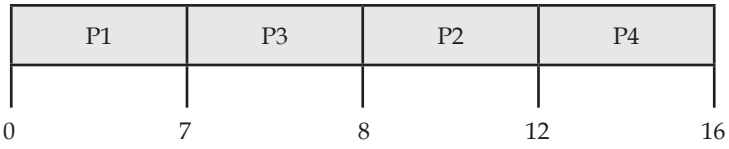
Two schemes:

1. non pre- emptive - once CPU given to the process it cannot be preempted until completes its CPU burst.
2. preemptive - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

   SJF is optimal - gives minimum average waiting time for a given set of processes.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (non-preemptive)

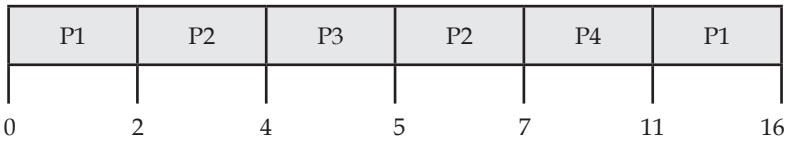


Average waiting time = [0 +(8-2)+(7-4) +(12-5)] /4 =4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (preemptive)



Average waiting time = (9 + 1 + 0 +2)/4 =3

Determining Length of Next CPU Burst

Notes

Can only estimate the length.

Can be done by using the length of previous CPU bursts, using exponential averaging.

Prediction of the Length of the Next CPU Burst

Pn+1 = a tn +(1-a)Pn

This formula defines an exponential average

Pn stores the past history

tn contents are most recent information

the parameter "a "controls the relative weight of recent and past history of in our prediction

If a =0 then Pn +1 =Pn

That is prediction is constant

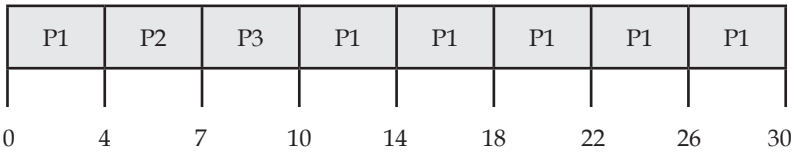
If a = 1 then Pn +1 = tn

Prediction is last cpu burst

Example: 4 **RR with Time Quantum = 4**

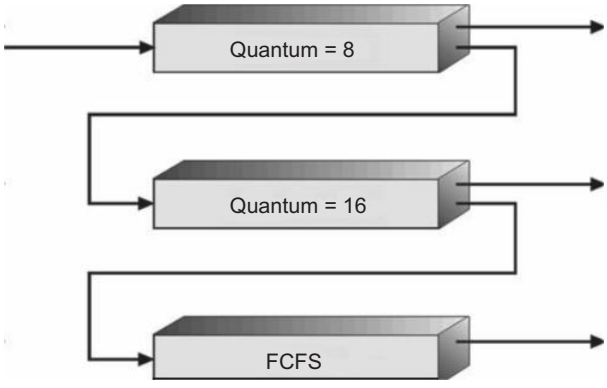
Process	Burst Time
P1	24
P2	3
P3	3

The Gantt chart is:



Average waiting time = [(30-24)+4+7]/3 = 17/3 = 5.66

Example: 5 **Multilevel Feedback Queue**



Notes

Three queues:

1. Q0 - time quantum 8 milliseconds
2. Q1 - time quantum 16 milliseconds
3. Q2 - FCFS

Scheduling:

1. A new job enters queue Q0 which is served FCFS . When it gains CPU, job receives 8 milliseconds.

   If it does not finish in 8 milliseconds, job is moved to queue Q1.

2. At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.

3.6.1 Conventional Time-share Scheduling

The goals for conventional time-share scheduling are concerned with maximizing CPU utilization and overall throughput, and minimizing average turn-around time, average waiting time, and response time. These goals are designed to maximize the use of system resources while appearing to provide an acceptable level of performance to the user. The algorithms used in conventional time-share scheduling are generally relatively simple to implement and understand. They try to effect a balance between system resource utilization and user perception of response and turn-around time. First Come First Served (FCFS) scheduling is a simple first in, first out queue. It is simple to implement but it has several deficiencies. Its average wait time is typically quite long. It is non-preemptive, and it is subject to the convoy effect if there are many I/O bound processes mixed with a few CPU bound processes. In this case, there can be large amounts of idle system resource time as the I/O bound processes sit idle waiting for the CPU bound process to complete. Shortest Job First (SJF) scheduling is provably optimal but requires clairvoyance to fully implement, since it is usually not possible to know *a* priori which job in the wait queue is the shortest job. There are methods, such as using the last processing burst or some function of the last burst, that allow SJF scheduling to be emulated. SJF can be implemented either preemptively or non-preemptively and its average waiting time is low. Priority scheduling is somewhat similar to many real-time scheduling algorithms. Priority scheduling can be preemptive or non-preemptive. Process starvation can be a problem with priority scheduling when many high priority processes are competing for time. Process aging can help this situation but aging effectively modifies the relative priorities of all process in the system and makes the system non-deterministic. Round Robin scheduling is similar to FCFS scheduling but preemption is added so that each time quantum a new process receives access to the system resources. This way each process gets a share of the system resources without having to wait for all processes in front of it to run to completion. Its average waiting time is typically rather long and its performance is directly related to the size of the time quantum. Round Robin scheduling is the degenerative case of preemptive priority scheduling when all priorities are equal. These scheduling algorithms are all in common use but they have deficiencies when applied to real-time scheduling. In particular, the lack of determinacy is a serious problem that makes them unsuitable for real-time scheduling.

3.6.2 Real-time Scheduling

In direct contrast with the goals for time-share scheduling, the goals for real-time scheduling are concerned with minimizing the time from stimulus to response, completing tasks within specific time constraints, and providing deterministic performance. Although system resource utilization is still of interest, it is no longer a primary driver. Determinism and temporal correctness are

Notes

now the principal concerns. The algorithms used, or proposed for use; in real-time scheduling vary from the relatively simple to the extremely complex. They try to provide deterministic performance while meeting the timing constraints of the system. Real-time algorithms can be divided into several classes of algorithm. The two major classes are off-line algorithms, which generate scheduling information prior to system execution which is then utilized by the system during runtime, and online algorithms, which generate scheduling information while the system is running. Online algorithms can be subdivided into static priority based and dynamic priority based algorithms. Static priority based algorithm's are relatively simple to implement but lack flexibility. Dynamic priority based algorithms have the flexibility to cope with changing environments but are typically complicated to implement and require a large amount of system resources to execute. Off-line algorithms are theoretically fully predictable since they are fully deterministic if the system is properly constrained. They are good for applications where all characteristics are known a priority and change very infrequently. Off-line algorithms require a fairly complete characterization of all processes involved, such as execution times, deadlines, and ready times. They typically require large amounts of off-line processing time to produce the final schedule and due to this they are quite inflexible. Any change to the system processes, either adding or deleting processes or changing the characteristics of one or more processes requires starting the scheduling problem over from the beginning. In its favor off-line scheduling requires minimal run-time processing time. Online Static Priority Based algorithms are arguably the most common in practice. They have a fairly complete theoretical base and they may be either preemptive or non-preemptive. They work well with fixed periodic tasks but do not handle aperiodic tasks particularly well, although there are methods to adapt the algorithms so that they can also effectively handle aperiodic tasks. A severe problem that can occur with Static Priority Based algorithms is priority inversion. This occurs when a higher priority task is blocked by a lower priority task which is using a resource required by the higher priority task. Dynamic Priority Based algorithms require the largest amount of on-line resources. However, this allows them to be extremely flexible. Many Dynamic Priority Based algorithms also contain an off-line component. This reduces the amount of online resources required while still retaining the flexibility of a dynamic algorithm. There are two subsets of dynamic algorithms—planning based for guaranteed performance and best effort to maximize performance in overload conditions. Planning based algorithms guarantee that if a task is accepted for execution, it and all other previously accepted tasks will meet their time constraints. On the other hand, best effort algorithms attempt to provide better response to aperiodic tasks or soft tasks while still meeting the timing constraints of the hard periodic tasks. This is often accomplished by utilizing spare processor capacity to service these soft or aperiodic tasks.

3.6.3 Real-time Scheduling AlgorithmsOff-line/Pre-run-time Scheduling

In systems using pre-run-time scheduling, the schedule is computed off-line. This requires a fairly complete characterization of all processes, such as ready times, deadlines, and execution time requirements. However, it does allow consideration of many different schedule possibilities since the only time constraint in generating the schedule is the amount of time the developer is willing to invest. If different system modes or some form of error handling is desired, multiple schedules can be computed, one for each alternate situation. At run-time, a small run-time scheduler can choose proper one. This scheduler can also be used for a limited number of aperiodic processes. Although a strict pre-run-time scheduler has no provisions for handling aperiodic tasks, it is possible to translate an aperiodic process into a periodic one, thus allowing aperiodic processes to be scheduled using pre-run-time scheduling.

()

One way to do this is to translate each aperiodic process

cd,,min

Notes

aa a()

into a corresponding periodic process

rc d prd,, , that satisfies the *pp p p*()

following conditions:

ccddcprd dd r=== = — =, , min , min ,10 *p aa pa p a p ap*

where *c*is the worst case computation time, *d*is the deadline, *r*is the release time,*prd* is the period and min is the minimum time between invocations of the aperiodic task. A major advantage of pre-run-time scheduling is a significant reduction in run-time resources required for scheduling. However, it is inflexible. Any change requires re-computing the entire schedule. Additionally, it cannot handle an environment that is not completely predictable. Its real advantage is that in a predictable environment it can guarantee system performance. In many cases, it is the only practical way to provide absolute predictability in a complex real-time system Although pre-run-time scheduling has several advantageous characteristics, there are also several concerns that must be addressed. A pre-run-time schedule is of little use if it does not satisfying the system timing constraints. Using pre-run-time scheduling, this can be guaranteed for sets of periodic and aperiodic functions if aperiodic functions are translated to periodic functions and a feasible schedule can be found where every process starts after its release time and completes before its deadline within a period that is contained within the least common multiple of their periods. In systems using pre-run-time scheduling, there is generally (if not always) a required ordering of the execution of processes. This can be accommodated by using precedence relations that are enforced during the pre-run-time scheduling. Preventing simultaneous access to shared resources and devices is another function that pre-run-time scheduling must enforce. This can be accomplished by defining which portion of a process cannot be preempted by another and then defining exclusion constraints and enforcing them during the pre-run-time scheduling. Another goal that may be desired for pre-run-time schedules is reducing the cost of context switches caused by preemption. This can be accomplished by choosing algorithms that do not result in a large number of preemptions, such as Earliest Deadline First. It is also desirable to increase the chances that a feasible schedule can be found. If the input to the chosen pre-run-time scheduling algorithm is exactly the input to the real system and not an approximation, then the mathematical pre-run-time algorithms are more likely to find a feasible schedule. As an example of a pre-run-time algorithm, their systems consist of a set of semi-preemptable activities that is modeled as a set of non-preemptible beads. These beads have four types of constraints:

1. *Absolute timing constraints*, where each bead must be scheduled between an earliest start time and its deadline.
2. *Relative timing constraints*, consisting of precedence relations between the beads.
3. *Consistency constraints,* which require an order between subsets of beads to enforce consistent usage of shared resources.
4. *Independency constraints*, which are used to exploit knowledge that only one alternative of a conditional is executed at run-time. The algorithm uses a heuristic scheduling approach, where the application designer specifies processes, resources, and constraints. Processes are then divided into non-preemptible beads based on pre-defined preemption points and functional constraints are derived. At this stage, each process and all its resources are assigned to an individual processor. Now all communication paths are known so communication beads are added and functional constraints are adjusted as necessary. Beads are then grouped into non-preemptible blocks to decrease context switching overhead and to reduce the number of discrete items that need to be scheduled. Windows are defined in which if all beads in a block complete within the window, all constraints have been met. Blocks are assigned a sequence so each falls within its window and no blocks overlap (on any discrete processor). If a feasible schedule can not be found, the non-preemptible

Notes

blocks are split into smaller pieces and more processors are added to the problem. The algorithm then begins again at the point of assigning processes and their resources to a processor.

Future Research

More work needs to be done on incorporating error detection, recovery and fault tolerance in pre-run-time scheduled systems. Although a technique for doing this was presented earlier in this paper, more robust methods need to be developed. Many pre-run-time scheduling techniques require a severe set of constraints in order to bound the amount of computation necessary to produce a schedule. These constraints often limit the applicability of the technique to real world conditions. More work needs to be done to eliminate as many of the constraints as possible if pre-run-time scheduling is going to be a viable solution for other than only the highly constrained system.

3.6.4 On-line Scheduling Algorithms

On-line scheduling algorithms compute a schedule in real-time as processes arrive. The on-line scheduler does not assume any knowledge of process characteristics for processes which have not yet arrived. Major advantages of on-line scheduling are that there is no requirement to know process characteristics in advance and they tend to be flexible and easily adaptable to environmental changes. Many arguments have been made against run-time scheduling. Static priority driven schedulers are capable of generating only a limited subset of the possible schedules. This, and the basic assumption that the system has no knowledge of process characteristics for processes that have not yet arrived, severely restricts the potential for the system to meet timing and resource sharing requirements. If the scheduler does not have such knowledge, it is impossible to guarantee that system timing constraints will be met. No matter how sophisticated the scheduler is, it is always possible that some newly arriving task will have characteristics that make either the process or some other process(es) miss its deadline. As an example, there are times when it is necessary to allow the processor to become idle if all timing constraints are to be met, such as when a high priority task with a far away deadline idles to wait for the arrival of a lower priority task with a near-term deadline. Neither static nor dynamic algorithms can deal with this type of problem. Because of the relative small number of possible schedules that can be produced by a run-time scheduler, CPU utilization is usually lower than that provided by a pre-run-time scheduler. Reliance on run-time mechanisms for process synchronization and mutual exclusion, such as rendezvous and monitors creates timing issues that are very difficult to predict. Additionally, the use of such mechanisms allows scheduling decisions to be made at the process level, rather than relegating them to the system. This introduces yet another source of uncertainty in predicting system performance. Other problems include starvation, priority inversion, and deadlock. In pre-run-time scheduling, it is possible to avoid all this overhead by defining precedence and exclusion relations between processes and process segments to allow synchronization and mutual exclusion. Allowing interrupts to occur at any random time from internal or external events further increases the unpredictability of a system. It also requires a significant amount of resources for context switching. Pre-run-time scheduling significantly reduces the amount of run-time resources needed for scheduling and context switching. In pre-run-time scheduling, a periodic interrupts are noted but not serviced until the corresponding periodic handler is scheduled to run. This reduces time spent in context switches and gives greater latitude in scheduling the entire system. Systems scheduled using on-line algorithms are usually "proved" by testing and/or simulation. As has been observed, you can only show the presence of errors, no amount of testing or simulation can prove the system error free. Additionally, in on-line scheduling the amount of time and resources available for scheduling is severely restricted and the scheduling problem is computationally hard. Although these arguments are compelling, many of today's real-time systems use on-line scheduling simply because it does perform reasonably well under most circumstances and it is flexible.

Notes

The flexibility of on-line scheduling makes up for most of the deficiencies when it comes to implementing a real world system.


Did u know?

The flexibility of on-line scheduling makes up for most of the deficiencies when it comes to implementing a real world system.

3.6.5 Non-Preemptive Static Priority Algorithms

Many real-time systems have the characteristic in which the order of task execution is known a priori and each task must complete before another task can start. These systems can be scheduled non-preemptively. This scheduling technique avoids the overhead associated with multiple context switches per task and can achieve very high processor utilization. These systems are modelled as systems of mutually constrained, sequential tasks. These constraints are of the form—Static non-preemptive run-time scheduling has the advantage of high efficiency since the only context switches performed are to initiate a new task. Additionally, tasks are guaranteed of meeting execution deadlines. Two non-preemptive algorithms will be examined—the parametric dispatching algorithm. Both of these algorithms attempt to provide high processor utilization while preserving task deadline guarantees and system schedulability. The parametric dispatching algorithm uses a calendar of functions, which maintains for each task functions, F min and F max, describing the upper and lower bounds on the allowable start times for the task. During an off-line component, the timing constraints between tasks are analyzed to generate the calendar of functions. During system execution, a calendar evaluator determines the upper and lower bounds for the start times for each task. These bounds are then passed to a dispatcher which then determines when within the window to start execution of the task. This decision can be based on whether there are other non-real-time tasks waiting to execute. Singh's predictive algorithm depends upon known a priori task execution and arrival times. When it is time to schedule a task for execution, the scheduler not only looks at the first task in the ready queue, but also looks at the deadlines for tasks that are predicted to arrive prior to the first task's completion. If a later task is expected to arrive with an earlier deadline then the current task, the scheduler may insert CPU idle time and wait for the pending arrival if this will produce a better schedule. In particular, the insertion of idle time may keep the pending task from missing its deadline. Suppose there is a task t with deadline D currently at the head of the queue 11 and there is another task, t with deadline D whose predicted arrival time, r,222 is later then the current time. If t and t's execution times result in t 12 2 completing later than t's deadline, then t will miss its deadline if t is 221 dispatched first. In this case, the only feasible schedule requires that the CPU idle while waiting for t's arrival2. These algorithms both have drawbacks when applied to real-world systems. Both algorithms require significant a priori knowledge of the system tasks, both execution times and ordering. Because of this, they are quite rigid and inflexible. Even minor task changes require significant rework of the scheduler. Additionally, the parametric algorithm is not a scheduling algorithm in its own right since it requires a predetermined task ordering. It is best used in conjunction with another scheduling technique. Future areas of development in non-preemptive scheduling include determining necessary and sufficient conditions for a task set to be schedulable. Additionally, neither of these algorithms has any technique for handling aperiodic tasks. Further work needs to go into incorporating non-preemptive scheduling with some other scheduling algorithm to handle aperiodic tasking and thus be useful for other than severely constrained systems.

3.6.6 Preemptive Static Priority-based Algorithms

In the early days of real-time computing, real-time systems were built around a cyclic executive, similar to a very rudimentary off-line scheduler, and constructed in a fairly undisciplined

Notes

manner, primarily due to a lack of any other technique. In the late '70s and early '80s, researchers realized that this method of constructing systems was producing systems that were inflexible and difficult to maintain. It was also realized that new methods and techniques were needed to support the analysis, design, and implementation of real-time systems. Although many scheduling techniques were proposed, preemptive static priority-based scheduling has emerged as one of the more widely studied and implemented techniques. Preemptive static priority-based scheduling algorithms form the basis for most commercially available real-time operating systems. The Ada95 and POSIX standards are also based upon preemptive static priority based algorithms. A literature search will reveal that preemptive static priority-based algorithms are among the most widely studied and understood of any class of scheduling algorithm. The model assumes:

1. All processes are periodic.
2. All processes have a deadline that is equal to their period.
3. All processes are independent of one another.
4. All processes have a fixed computation time.
5. No process may voluntarily suspend itself.
6. All processes are released as soon as they are ready.
7. Overheads are ignored.

The basic timing constraint assumption is that the computation time of all processes is less than or equal to the deadline, which is equal to the period. Clearly, if this is not the case, the problem is unsolvable since the computation time will exceed the period. There are several reasons for the popularity of the preemptive static priority-based algorithms. This class of algorithms tends to be easy to understand and easy to implement. In the most basic scenario, the implementation consists of little more than a priority queue. Additionally, given the necessary system constraints, these algorithms can guarantee task completion. If execution guarantees are not required, this algorithm class is also quite flexible. Tasks can be added or deleted with no change to the scheduler. The rate-monotonic algorithm is one of the most well known in preemptive static priority-based scheduling and many algorithms are based upon variations of it. That a set of *n* independent periodic tasks, where a periodic task t is characterized by a I period T and a worst case execution time C.

The upper bound on the utilization *ii Ti*1is ln 2 = 0.69 as *n* approaches infinity. Fortunately, this bound is *n*() 21-*n* very pessimistic. The average CPU utilization for a task set scheduled using the rate-monotonic algorithm was 88%. The deadline-monotonic scheduling algorithm is really just a special case of the rate-monotonic algorithm. The rate-monotonic algorithm assumes that a task's deadline is the same as the end of the task's period. Deadline-monotonic scheduling was developed to comprehend periodic task's which have deadlines prior to the end of their period, resulting in a narrower window of opportunity than the task's period. That a set of *n* periodic tasks scheduled by the deadline-monotonic algorithm will always meet its deadlines if:()()*CBEC*1== + ++ ++ = -*I in C*,,121 L *ii I* 1 2 *I T TT i*1 2*i* where *B* is the duration in which task t is blocked by lower priority tasks,*C* is *ii I* the task's required execution time,*T* is the task's period,*E* is the difference *I I* between the task's period and it's deadline. When the task's deadline is () earlier than the end of the period, *ETD* =- where *D* is the earlier *I iii* deadline. It has also been proven that assigning higher priorities to tasks with narrower windows is optimal for scheduling.

The main advantage of deadline-monotonic scheduling over rate-monotonic scheduling is that there are task sets that cannot be scheduled using rate-monotonic scheduling but are schedulable using deadline-monotonic scheduling. The weight-monotonic scheduling algorithm attempts to generate schedules that exhibit temporal fairness. Temporal fairness is a property that provides for more equal allocation of processor time over some period to all tasks, thus providing relative

Notes

smooth progression, without significant bursts of activity for any one task or group of tasks. Proportionate progress is used as a measure of temporal fairness. Assuming that processing time is divided into equal length slots, a schedule exhibits proportionate progress if for all processes p at all integer times t, the difference between the amount of processor time (number of slots) that should have been allocated to process p and the amount that was allocated is strictly less than the absolute value of 1 for all p at time t. Baruah has shown that proportionate progress is a stronger requirement than periodic scheduling. That is, all schedules exhibiting proportionate progress are periodic, but not all periodic schedules display proportionate progress. The weight-monotonic algorithm first defines a weight for each task in the system, where the task's weight is the ratio of its required execution time to *xe* its period. At each time slot, the processor is assigned to the task with *xp* the greatest weight, with ties broken arbitrarily. At first glance, this looks very similar to the rate-monotonic algorithm. The difference is that the rate-monotonic makes scheduling decisions based solely on the task's period while the weight-monotonic algorithm also considers its execution requirement. This has the effect of elevating the priority of a task that would have been given low priority under the rate-monotonic algorithm so that it receives a proportionally larger share of the available processing time. It seems that this would negate the advantages gained from rate-monotonic scheduling but experiments show that the weight-monotonic algorithm often successfully schedules task sets that were unschedulable under the rate-monotonic algorithm. The reason for this apparent contradiction is the difference in the definition of "contending tasks" between schedules exhibiting proportionate progress and periodic schedules. Essentially, in proportionate progress schedules, high priority tasks do not completely take over the processor, thus allowing lower priority tasks an opportunity to run and avoid starvation. Static priority-based scheduling algorithms have a number of problems that have to be addressed when they are used for real-world problems. Under overload conditions, it is difficult to guarantee performance of static priority-based algorithms. It can also be quite difficult to guarantee performance and achieve high processor utilization. Without providing some kind of algorithmic extension, such as a server task, static priority-based algorithms do not have provisions to handle "soft" or aperiodic processes. This topic will be addressed in more detail later in this paper when dynamic priority algorithms are discussed. A phenomenon known as priority inversion, which in essence makes a high priority task block while waiting on a lower task to release some resource can also be a problem although there are several techniques to counter this situation. The Priority Inheritance Protocol, Priority Ceiling Emulation, and Priority Ceiling Protocol are all useful in alleviating the problem. The Priority Inheritance Protocol acts when a higher priority task is blocked by a lower priority task. When this occurs, the lower priority task inherits the priority of the higher priority task. This prevents a medium priority task from preempting the lower priority task. When the lower priority task releases the resource the higher priority task was blocking on, the lower priority task is returned to its original priority level. Although the Priority Inheritance Protocol prevents unbounded blocking of a higher priority task by a lower priority task, it does not guarantee that mutual deadlocks will not occur. It also suffers from the possibility of chained blocking. Chained blocking occurs when two lower priority tasks each hold a resource that a higher priority task desires. The higher priority task now has to wait on both lower priority tasks to release their resources before it can run. Priority Ceiling Emulation combats the problem of priority inversion by selectively inhibiting preemption. With this method, the priority of a low priority task is raised high enough to prevent it being blocked by a medium priority task. To accomplish this, the highest priority of any task that will lock a resource is kept as an attribute of that resource. Whenever a task is granted access to that resource, its priority is temporarily raised to the maximum priority associated with the resource. When the task has finished with the resource, the task is returned to its original priority. Under this protocol, deadlocks cannot occur and a task can be blocked at most once by a lower priority task. The Priority Ceiling Protocol is a combination of the two

Notes

previous methods that prevents chained blocking and mutual deadlocks. In this protocol, each resource is assigned a priority ceiling which is the highest priority of any task that may request its service. The following rules then apply:

1. A higher priority task always preempts a lower priority task.
2. A task cannot enter its critical section unless its priority is higher than the priority ceiling of all resources currently locked by other tasks.
3. A lower priority task that blocks a higher priority task temporarily inherits the priority of the higher priority task.

The only difference between this protocol and the Priority Inheritance Protocol is the addition of rule 2. However, this rule is what prevents chained blocking and mutual deadlocking by forcing a total ordering of executing and suspended critical sections. The relative complexity of implementing this protocol is its primary drawback. A real-world problem that often arises is the existence of a finite number of priority levels. Most algorithmic research tends to assume the availability of infinite priority levels for ease of modeling and analysis but in practical systems, this is just not the case. The effects of limited priorities and have shown that limited priorities affect the degree of schedulability (maximum processor utilization) and produce a form of priority inversion. For a system scheduled using the rate-monotonic algorithm with the potential requirement of 100,000 priority levels, limiting the system to only 256 priority levels reduced the maximum processor utilization by a factor of 0.9986, which is negligible. However, reducing the number of priority levels to 16 reduces the maximum processor utilization by 0.7025 which is starting to become significant. Reducing the number of priority levels to four reduces the maximum utilization by 0.0811. This translates into less than 7.5% processor utilization if task deadlines are guaranteed. Additionally, reducing the number of priority levels means that tasks of slightly different priority must be grouped together under a single priority level. This has the effect of causing another form of priority inversion that has no work around, such as the priority ceiling protocol, due to a higher real priority task blocking on a task within the same priority group that has a lower real priority. The lack of performance guarantees under overload conditions does not have an easy work around with static priority-based algorithms. Most attempts to counter this deficiency involve modifying the scheduling algorithm to use some form of adaptive priority. These algorithms will be explored later when dynamic priority algorithms are discussed. The difficulty in obtaining high processor utilization is another problem that has no simple work around. This tends to be task set specific and general methods to ensure high processor utilization under these scheduling algorithms do not exist. Most attempts at countering this problem add a "server task" or something similar to attempt to "capture" the idle processor time, but then the scheduling algorithm falls into the class of best effort algorithms, another form of dynamic priority algorithm that will be discussed later.

Future Research

Much of the current work has focused on uniprocessor applications of the static priority-based scheduling algorithms. When multiple processors are involved, the complexity of the problem increases exponentially. However, with today's multiprocessor systems becoming more and more common, this is an area that needs further research. Burchard, Liebeherr, Oh, and Son among others, have begun to investigate the problem of adapting static priority-based scheduling algorithms to multiprocessors. They have proposed a method that develops more stringent schedulability conditions so that more tasks can be assigned to each processor using rate-monotonic scheduling then was previously considered feasible. However, their results depend upon "novel" schedulability conditions and more work is needed before their techniques are ready for real-world use. Lortz and Shin have also investigated multiprocessor scheduling and have concluded that in multiprocessor systems, the implementation of mutual exclusion and task

Notes

synchronization is handled better by global semaphores based on a task's blocking tolerance (the amount of time it can be blocked without missing its deadline) rather than the task's priority. In this scenario, simple FIFO queues for the semaphores perform better than rate-monotonic queues in direct opposition to what provides the best performance on a uniprocessor. These results underscore the vast differences between uniprocessor and multiprocessor real-time scheduling that need to be explored. Another topic that needs additional work is the integration of static priority-based scheduling algorithms with adaptive scheduling algorithms to handle aperiodic and soft deadline tasks and to provide a smooth, graceful degradation of system performance under overload conditions while still providing the advantages of static priority-based scheduling under normal conditions.

3.6.7 Dynamic Planning-based Algorithms

Static priority-based scheduling algorithms were shown to have many advantages but two of their disadvantages have received a significant amount of study. Their low processor utilization and poor handling of aperiodic and soft-deadline tasks have prompted researchers to search for ways to combat these deficiencies. This research has resulted in a class of scheduling algorithms known as Dynamic Planning-Based algorithms. Dynamic planning-based algorithms attempt to improve the response and performance of a system to aperiodic and soft-deadline tasks while continuing to guarantee the performance of the hard-deadline periodic tasks. The traditional way of handling aperiodic and soft-deadline tasks in a system that contained periodic tasks with hard-deadlines was to allow the aperiodic or soft tasks to run in the background. This meant that these types of tasks only got serviced when the processor had nothing else to do. The result of this was unpredictable and normally rather poor response to these tasks. The other approach used was to model aperiodic tasks as periodic tasks with a period equal to the minimum time between their arrivals and then schedule them using the same algorithm as for the real periodic tasks. This tended to be extremely wasteful of CPU cycles because the minimum period between arrivals was usually significantly smaller than the average. The general model for these types of algorithms is a system where all periodic tasks have hard deadlines equal to the end of their period, their periods are constant, and their worst case execution times are constant. All aperiodic tasks (and soft-deadline tasks) are assumed to have no deadlines and their arrival, or ready, times are unknown. Dynamic Planning-Based algorithms tend to be quite flexible in servicing aperiodic tasks while still maintaining the completion guarantees for hard-deadline tasks. Most of the algorithms also provide a form of guarantee for aperiodic tasks. They will not accept the task for execution if they cannot guarantee its on-time completion. Additionally, most of the algorithms can provide higher processor utilization than static priority-based algorithms while still including task completion guarantees. Earliest Deadline First (EDF) scheduling was one of the first dynamic planning-based algorithms proposed. It provides the basis for many of the algorithms currently being studied and proposed. The concept behind EDF is relatively simple. When it is time to select a task to execute, select the task that has the earliest deadline. One advantage of EDF is that if a task set is scheduled using the Earliest Deadline First algorithm, there will be no idle processor time prior to system overload. This means that unlike static priority-based algorithms, when using EDF, the system will only suffer overload when the execution time requirement for the task set exceeds 100% of the available time. This compares with approximately 69% for rate-monotonic scheduling. Of course, in reality there will be some overhead associated with scheduling and context switches but when using EDF scheduling, processor utilization can approach 100%. The Predictive Deadline (PD) algorithm extends the EDF algorithm to predict timing faults and to reject less critical tasks during overload conditions. The PD algorithm assumes cyclic task behavior where each cycle is defined by the task's ready time and its deadline. Tasks are required to estimate their execution time, which is used by the algorithm to predict overloads. Each task is also assigned a priority which is used to shed less important tasks during overload conditions. Tasks are then scheduled by earliest deadline. As each task is inserted into the ready queue, the sum of the execution times for all tasks preceding

Notes

each task with a later deadline is computed. When this is complete, if no overloads are predicted, the task with the earliest deadline is removed from the queue and begins execution. If an overload is predicted, the algorithm scans the ready queue and removes the lowest priority task. It then re-computes the execution sums. This process continues until there is either no overload predicted or until the ready queue is empty. One obvious drawback to this algorithm is that under overload conditions, the processor can spend a significant amount of time attempting to perform scheduling, which adds to the overload problem by using up CPU resources.

Another algorithm that is based upon the EDF algorithm is the Total Bandwidth server. This algorithm is based on the concept that many real-time systems consist of both hard periodic tasks and firm aperiodic tasks. To guarantee the timely completion of the hard periodic tasks in overload conditions, it is sometimes necessary to reject some of the aperiodic tasks. A special Server task is included in the system whose computation time, or capacity, is used to service these aperiodic tasks. This way, aperiodic tasks can be serviced in a more predictable fashion without jeopardizing the completion of hard periodic tasks. In this algorithm, a value is assigned to each aperiodic task. This value is only received if the task completes by its deadline. It is also used to determine which tasks should be rejected during overload conditions. During overload conditions, the algorithm has to make choices on which tasks will complete and which will not in order to maximize the value of the system. The original Total Bandwidth server algorithm works by assigning a suitable deadline to each aperiodic task when it arrives and then scheduling it with the periodic tasks using the EDF algorithm. The deadline for the aperiodic task is determined as follows. When the *k*-th aperiodic task arrives at time *t=r*, it is given the deadline *d* where: *k k*() *drd C*=+ max , *k U kkk*1 –*S* where *d* is the deadline of the *k*-1-th task,*C* is the maximum execution time *k-1k*of the task and *U*is the server utilization factor (its bandwidth in terms of *S* CPU execution time). The task is then placed in the ready queue as defined by EDF.

The algorithm has been extended to add resource reclamation and to provide a robust guarantee mechanism to provide graceful degradation of the system under overload conditions. The original algorithm used the maximum execution time of the task to schedule a portion of the server's bandwidth. Not all tasks take the maximum amount of time to complete. In order to reclaim the server bandwidth that was scheduled for a task, the actual execution time of a task that completes early is used to compute the deadline that could have been assigned to it if its actual execution time had been known. This adjusted value is then used to compute the deadline of the following request. Graceful degradation during overload conditions is added to the algorithm by calculating the deadline as shown above and comparing it to the task's actual deadline. If the computed deadline is less than or equal to the actual deadline, the task is accepted and scheduled. If the computed deadline is later, the algorithm searches for the lowest value task that has been scheduled and rejects it in the hope that the newly arrived task can be scheduled with guaranteed completion. This search continues until the task has either been scheduled or the ready queue is empty. Another server algorithm is the Dynamic Priority Exchange (DPE) server. This algorithm trades the server execution with lower priority periodic tasks (those with later deadlines) when there are no aperiodic tasks to be serviced. This way no CPU time is wasted, it is only exchanged between tasks, unless the system is idle.

The server has a period *T* and a capacity *C*. At the beginning of each *S S* period, the server's capacity is set to *C* . Each periodic task that has a *S* deadline within the current period is assigned an aperiodic capacity which is initially set to 0. All tasks are then assigned priorities via the EDF algorithm. When the highest priority in the system is an aperiodic capacity of *C* units of time, the following occurs:

- If there are aperiodic tasks waiting to be serviced, they are executed until they either complete or the capacity of the server has been exhausted. If there are no aperiodic tasks awaiting service, the periodic task with the earliest deadline is executed and a capacity equal to the execution time of that task is added to the aperiodic capacity of the task's deadline and subtracted from *C*. This has the effect of exchanging the deadlines of the highest priority capacity and the periodic task.

Notes

- If there are no periodic or aperiodic tasks waiting to execute the capacity of *C* is consumed in idle time. In order to implement the algorithm, the capacity of the server and the periodic process capacity must be updated every time there is an exchange and the server must be checked to ensure that its capacity has not been exhausted. The Dual Priority algorithm is an example of the class of algorithms that identify and exploit spare processing capacity in the system. Under this algorithm, hard periodic tasks execute at either an upper or lower band priority level. At their ready time, the tasks assume the lower band priority. After a fixed time from their ready time, they are promoted to the high priority band. Other soft deadline tasks are assigned a medium priority in between the two bands. This way soft deadline tasks are given preferential scheduling until hard priority tasks undergo promotion. The offset from the ready time in which promotion takes place is determined for each task via off-line analysis, using worst case arrivals and execution times. The spare capacity reclamation is accomplished by identifying spare capacity that becomes available when tasks do not arrive at their maximum rate. The Polling Server (PS) algorithm schedules a periodic task, the polling server that is used to provide relative high priority services to aperiodic tasks. Every time it executes, it is available to service existing or newly arriving aperiodic tasks during its execution period. The server is subject to preemption by higher priority tasks and runs until it either its period ends or until there is no aperiodic tasks left to execute. In the latter instance, it looses any time that was left in its period and is unable to service aperiodic tasks until the beginning of its next execution period. The Deferrable Server (DS) algorithm, a derivation of the rate-monotonic algorithm, is designed to improve the response time of the system to aperiodic tasks by deferring the completion time of hard periodic tasks while ensuring that their deadlines are still met. This algorithm is another instance of dynamic algorithms that utilize a special server process to service aperiodic tasks. The Deferrable Server algorithm schedules a periodic task to service aperiodic tasks with high priority. Like the Polling Server, it is ready to service aperiodic tasks that are waiting or arrive during its period unless it is preempted or runs out of execution time. Aperiodic tasks that arrive outside of its period are scheduled as background tasks. Unlike the Polling Server, however, it does not lose its remaining execution time once no more aperiodic tasks are pending. It remains able to service aperiodic tasks until the end of its period, at which time any unused execution time is lost. It is important to note that the Deferrable Server (DS) task is different from the other periodic tasks in the system. It is demand driven and does not run unless there is an aperiodic task to service. This means that it does not necessarily begin execution at the beginning of its period. It defers its execution until needed. The DS task is assigned a priority based on its period. If it is given the highest priority by making its period no longer than the period of the shortest periodic task, then it can guarantee responsiveness to aperiodic tasks. At lower priority levels, the system response to aperiodic tasks begins to degrade. The Reservation-Based algorithm was designed to guarantee all periodic task deadlines while minimizing the chances of missing an aperiodic task's deadline. The algorithm schedules all periodic tasks using the rate-monotonic technique. Aperiodic tasks are assumed to have lower priorities than periodic tasks and are scheduled First-Come, First-Served by using processor time that is left over after all the periodic tasks have been executed. This is accomplished by designating a unit cycle, which is the greatest common divisor of all task periods. The time left over in each unit cycle is reserved for the execution of aperiodic tasks. The Reservation-Based algorithm differs from the Polling and Deferrable server algorithms in that it does not create a periodic server to deal with the aperiodic tasks. This makes it much simpler to implement. It also has the goal of ensuring that aperiodic tasks meet their deadline, in contrast to the PS and DS goal of quick response. The largest drawback of dynamic priority-based algorithms is the overhead required to identify spare capacity in the system. Along with this overhead, there is a processor utilization issue. Several of the algorithms sacrifice some portion of useable CPU time in order to rapidly service aperiodic tasks. This can require a more powerful processor in order to provide the extra computing capacity or there may be other periodic tasks that could have been incorporated into the design that were eliminated to provide the extra capacity.

Notes

Future Research

The overhead required to identify spare system capacity needs to be incorporated into the models that are currently used to study Dynamic Priority-Based algorithms. Until this is done, the results lack real-world validity. There also needs to be an effort directed at adapting these algorithms or developing new ones for distributed systems. Identifying spare capacity on a global instead of local scale should introduce many new challenges.

3.6.8 Dynamic Best Effort Algorithms

In many real-time systems, there are a set of tasks which absolutely must complete by their deadlines or catastrophic system failure occurs. These systems often also have another set of tasks in which it is not necessary for every instance of the task to meet its deadline or in which the repetition rate of some set of tasks can be varied as system load varies. Examples are packet audio and packet video. As long as most of the packets arrive by their deadlines, the requisite information will be conveyed with minimal degradation. Another example is in a radar system where the sample rate for a target can be varied dependent upon its course and speed. In this case, the number of targets being tracked can dynamically vary as a function of the target parameters. In these kinds of systems, Dynamic Best Effort algorithms provide a means to cope with the situation where not every task can complete by its deadline. In particular, when a system begins to overload, dynamic best effort scheduling can provide a graceful and orderly degradation of performance for all task groups rather then randomly letting some fail while others randomly succeed. Unfortunately, many scheduling algorithms that work well under normal conditions fail miserable when the system begins to overload. As an example, the Earliest Deadline First algorithm, which has been shown to be optimal under non-overload conditions, has also been shown to perform even worse than random scheduling under overload conditions. The system model for dynamic best effort scheduling is the system with multiple processing streams where the failure of some quantity of the repetitive tasks within the stream can fail without causing catastrophic failure of the stream or in which the repetition rate of some periodic task can be varied as some function of the system processing parameters. Dynamic Best Effort scheduling has several advantages. Most important is the ability of most of these algorithms to provide relatively good performance in overload conditions in comparison to other scheduling techniques. By using this scheduling paradigm, a system can maximize the likelihood of the maximum possible number of tasks completing by their deadlines, or the most critical tasks completing by their deadlines, or the highest value tasks to the system completing by their deadlines. This gives the designer significant latitude in determining how the system will respond in overload conditions. Additionally, Dynamic Best Effort scheduling can maximize processor utilization when tasks have periods which vary dynamically.

The task model for Dynamic Best Effort scheduling varies depending upon the assumptions made about the tasks in the system. Task models that have been used for analysis are given below:

3.6.9 Equal Request Times

All tasks in the interval under study request execution at the same time. An example of a system for which this would hold is a communications switch which periodically polls its incoming lines for packets to forward. All packets that have arrived since the last poll request immediate service even though the processing time and deadlines for the packets may vary.

3.6.10 Equal Execution Times

In this case, all tasks have identical execution times, even though they may be ready for execution at different times and have different deadlines. An example is a fire-control system where all targets take an identical amount of time to process but may arrive at different times and have different deadlines for the targeting solution dependent upon target speed and location.

Notes

3.6.11 Monotonic Absolute Deadlines

A task requesting service is guaranteed to have a deadline that is at least as late as all other tasks that have previously requested service. This is essentially a first-come, first-served system, such as might occur in a process control system where the processing steps are strictly ordered.

3.6.12 Equal Relative Deadlines

All tasks in this model have the same relative deadlines. This is similar to the offer by a well known pizza delivery company that your pizza will be there in 30 minutes or it is free.

3.6.13 Equal Absolute Deadlines

In this case, all tasks have the same absolute deadline. An example is a banking system in which transactions occur at random times during the day and require varying amounts of processing but must all be completed by the end of the business day. Algorithms of this class have the advantage of being very flexible in dealing with a varying system environment and being able to deal with overload conditions without total failure. In fact, except for the condition of Equal Request Times and Equal Absolute Deadlines, the efficiency of the schedule generated by a Dynamic Best Effort scheduling algorithm is only one half to two-thirds that of an optimal scheduler. In the most general terms, the Dynamic Best Effort scheduling algorithms differ only in the way they assign priorities. For example, the Earliest Deadline First algorithm attempts to minimize task failures by scheduling the task with the earliest deadline to execute first. Unfortunately, this algorithm suffers from the domino effect, where the failure of one task dominos and causes other tasks further down the line to also miss their deadline. Because of this, its performance rapidly degrades under overload conditions. Another approach is to schedule the task with the highest value first. In this case, the task that has the most importance to the system is the first to run. A third approach is to prioritize task dispatching by task density, where density is the importance value of the task divided by the cost to execute the task. This approach tends to maximize the cumulative value of the tasks that complete by their deadlines. A mixed approach has also been used. Here the importance value of a task and its deadline are both used to compute a weighted sum of the importance and the deadline which is then used to determine the task's priority. Unfortunately, none of these algorithms as presented provide any form of guarantee of task completion. Because of this, they are all prone to the domino effect described earlier since they have no awareness of the current processor loading and therefore if scheduling a new task will cause a problem. Buttazzo, et al. have proposed and characterized a class of extensions to the previous algorithms they refer to as guaranteed. These extensions perform an acceptance test before accepting a new task for execution. If a processor overload will occur by accepting the newly arrived task, it is rejected. Unfortunately, this extension does not consider the importance of the task so an extremely important task may be rejected so that a task of low importance can run. To counter this problem they have proposed a robust extension. This extension performs by applying the acceptance test to each arriving task similar to the guaranteed extension. However, if an overload condition is detected, the least valued task that will remove the overload is rejected rather than blindly rejecting the newest task. Additionally, rejected tasks are temporarily maintained in a reject queue from which they can possibly be executed at a later point in time. Another Dynamic Best Effort algorithm, for situations such as voice or video, where m of n tasks must complete by their deadline has been proposed by Hamdaoui and Ramanathan. This system model is more m restrictive than a system in which percent of the tasks must complete by n their deadline. The latter model allows failing tasks to group together which still meets the percentage of task completion criteria but can severely degrade system performance. As an example, consider live video. Missing two frames out of every 20 may only cause a little static or jerkiness in the picture. However, missing the middle 40 frames out of a 400 frame sequence is clearly unacceptable. In the former model, where the errors are required to be spread out, the Distance Based Priority (DBP) algorithm

Notes

determines the priority of a task by calculating how close the associated processing stream is to failure. Streams that are close to dynamic failure are accorded higher priority than streams that are relatively safe. To illustrate, suppose we have three concurrent streams, S, S, and S,1 2 3 that each have an 8 of 10 constraint. Suppose that in the current window, S 1 has processed six tasks of which one failed to complete, S has processed 2 nine tasks of which two have failed to complete, and S has processed seven 3 tasks with no failures. Under these conditions, S, will receive the highest 2 priority since even one more task failure will cause a stream failure. S, will 1 receive the next highest priority, and S will receive the lowest priority since it 3 is the farthest from stream failure. One issue that arises for this algorithm is the effect of finite priority levels. Although many systems have 256 or more priority levels available, many backplanes and communication processors, where this algorithm is particularly appropriate, have a limited number of priority levels. In general, a stream with (m, n)-firm deadlines needs *n-m*+ 2 priority levels to fully implement the algorithm. However, simulation studies performed by Hamdaoui and Ramanathan have shown that there is a substantial reduction in dynamic stream failures with as few as three priority levels. Adaptive Best Effort scheduling has been proposed by Wang and Lin to handle repetitive tasks. These tasks differ from periodic tasks in that there is normally no control over the spacing between task instances. The spacing between two instances of a specific task can vary from zero to twice the period of the task, depending upon the current workload of the system. In this situation, rather than accepting or rejecting tasks, or just letting certain tasks fail, the system attempts to adjust the period of these tasks. The difficulty associated with this form of scheduling is that the standard periodic task model does not comprehend dynamically varying periods. This requires developing an adaptive task model. The primary difference between the periodic task model and the adaptive task model is in selecting the task ready times. In the periodic model, a task's next ready time coincides with its last completion time. In the adaptive model, the ready time can be set to anytime prior to the task deadline, dependent upon task execution requirements. This way the task ready time and deadline can be considered the minimum and maximum spacing from the previous execution of the task. Although several different scheduling policies could be used with Adaptive Best Effort scheduling, Wang and Lin confined their studies to the Earliest-Deadline-First (EDF) and Rate-Monotonic (RM) policies since they have both been studied extensively and are relatively common in many commercial real-time operating systems. To carry out their study, they () rc-I I j defined a ready ratio, *R* , for the *j*-th instance of the task t I as *R* = - 1 I I j F j j i where *r* is the ready time, or earliest time that the *J* -th instance of task t can *I j* run, *c* is the time the task completes, and *F* is the frame time, or period of *ii j* the task. An adaptive system with fixed ready ratios is one in which all ready ratios are set to a constant value. An adaptive system with rate monotonic ready ratios is one in which the ready ratios are set according to the system workload indexes, which are the total execution times of a task divided by its period, and the task periods. The results of their studies showed that under a normal workload, the adaptive EDF scheduler with rate-monotonic ready ratios clearly provided the best performance. However, as the workload on the system increased, periodic EDF (non-adaptive) provided a better scheduling alternative. Fixed ready ratio EDF simply did not perform well. For use in real world applications, Adaptive Best Effort algorithms have several drawbacks that need to be overcome. Adding task importance parameters to the algorithm could allow it to more gracefully degrade under heavy workload conditions by allowing it to discard tasks of low importance, rather than selecting tasks essentially at random. However, this algorithm appears to need significant further study before it would be applicable to real world systems.

Future Research

Future work in Dynamic Best Effort algorithms should include developing algorithms that have more deterministic performance under overload conditions. Additionally, investigation of other periodic scheduling policies for the adaptive period algorithm might provide improved performance.

Notes

3.7 An Overview of Thread Scheduling

We will start by looking at the basic principles of how threads are scheduled. Any particular virtual machine (and underlying operating system) may not follow these principles exactly, but the principles form the basis for our understanding of thread scheduling. Let us start by looking at an example with some CPU-intensive threads. In this and subsequent units, we will consume CPU resources with a recursive Fibonacci number generator, which has the advantage (for our purposes) of being an elegant and very slow program:

```
import java.util.*;
import java.text.*;

public class Task implements Runnable {
    long n;
    String id;

private long fib(long n) {
    if (n == 0)
        return 0L;
    if (n == 1)
        return 1L;
    return fib(n - 1) + fib(n - 2);
}

public Task(long n, String id) {
    this.n = n;
    this.id = id;
}

public void run( ) {
    Date d = new Date( );
    DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");
    long startTime = System.currentTimeMillis( );
    d.setTime(startTime);
    System.out.println("Starting task" + id + "at" + df.format(d));
    fib(n);
```

Notes

```
    long endTime = System.currentTimeMillis( );
    d.setTime(endTime);
    System.out.println("Ending task" + id + "at" + df.format(d) +
       "after" + (endTime - startTime) + "milliseconds");
    }
}
```

We have made this class a Runnable object so that we can run multiple instances of it in multiple threads.

```
public class ThreadTest {

    public static void main(String[] args) {
       int nThreads = Integer.parseInt(args[0]);
       long n = Long.parseLong(args[1]);
       Thread t[] = new Thread[nThreads];

    for (int i = 0; i < t.length; i++) {
       t[i] = new Thread(new Task(n, "Task" + i));
       t[i].start( );

    }

    for (int i = 0; i < t.length; i++) {
    try {
    t[i].join( );
       } catch (InterruptedException ie) {}
       }
       }
       }
```

Running this code with three threads produces this kind of output:

Starting task: Task 2 at 00:04:30:324

Starting task: Task 0 at 00:04:30:334

Starting task: Task 1 at 00:04:30:345

Ending task: Task 1 at 00:04:38:052 after 7707 milliseconds

Notes

```
Ending task:  Task 2 at 00:04:38:380 after 8056 milliseconds
Ending task:  Task 0 at 00:04:38:502 after 8168 milliseconds
```

Let us look at this output. Notice that the last thread we created and started (Task 2) was the first one that printed its first output. However, all threads started within 20 milliseconds of each other. The actual calculation took about eight seconds for each thread, and the threads ended in a different order than they started in. In particular, even though Task 2 started first, it took 349 milliseconds longer to perform the same calculation as Task 1 and finished after Task 1.

Generally, we had expect to see similar output on almost any Java virtual machine running on almost any platform: the threads would start at almost the same time in some random order, and they would end in a (different) random order after having run for about the same amount of time. Certain virtual machines and operating systems, however, would produce this output:

```
Starting task:  Task 0 at 00:04:30:324
Ending task:  Task 0 at 00:04:33:052 after 2728 milliseconds
Starting task:  Task 1 at 00:04:33:062
Ending task:  Task 1 at 00:04:35:919 after 2857 milliseconds
Starting task:  Task 2 at 00:04:35:929
Ending task:  Task 2 at 00:04:37:720 after 2791 milliseconds
```

The total here takes about the same amount of time, but now they have run sequentially: the second task did not begin to execute until the first task was finished. Another interesting fact about this output is that each individual task took less time than it did previously.

3.7.1 Priority-based Scheduling

In each of these examples, multiple threads compete for time on the CPU. When multiple threads want to execute, it is up to the underlying operating system to determine which of those threads are placed on a CPU. Java programs can influence that decision in some ways, but the decision is ultimately up to the operating system.

A Java virtual machine is required to implement a preemptive, priority-based scheduler among its various threads. This means that each thread in a Java program is assigned a certain priority, a positive integer that falls within a well-defined range. This priority can be changed by the developer. The Java virtual machine never changes the priority of a thread, even if the thread has been running for a certain period of time. The priority value is important because the contract between the Java virtual machine and the underlying operating system is that the operating system must generally choose to run the Java thread with the highest priority. That is what we mean when we say that Java implements a priority-based scheduler. This scheduler is implemented in a preemptive fashion, meaning that when a higher-priority thread comes along, that thread interrupts (preempts) whatever lower-priority thread is running at the time. The contract with the operating system, however, is not absolute, which means that the operating system can sometimes choose to run a lower-priority thread. Java's requirement for a priority-based, preemptive scheduling mechanism maps well to many operating systems. Solaris, the various Windows operating systems, Linux, and most other operating systems on desktop computers and servers all provide the support for that kind of thread scheduling. Certain operating systems, particularly those on specialized devices and on smaller, handheld devices, do not provide that level of scheduling support; Java virtual

Notes

machine implementations for those operating systems must perform the necessary thread scheduling on their own.

Our first example, where the threads all complete at about the same time, is executed on a standard operating system (Solaris) where the thread scheduling is handled by the operating system. Our second example, where the threads run sequentially, is from a system where the Java virtual machine itself handles the thread scheduling. Both implementations are valid Java virtual machines.

3.7.2 The Scheduling Process

Let us examine how the scheduling process works in a little more detail. At a conceptual level, every thread in the Java virtual machine can be in one of four states:

3.7.2.1 Initial

A thread object is in the initial state from the period when it is created (that is, when its constructor is called) until the start()method of the thread object is called.

3.7.2.2 Runnable

A thread is in the runnable state once its start() method has been called. A thread leaves the runnable state in various ways, but the runnable state can be thought of as a default state: if a thread is not in any other state, it is in the runnable state. A thread that is in the runnable state may not actually be running; it may be waiting for a CPU. A thread that is running on a CPU is called a currently running thread.

3.7.2.3 Blocked

A thread that is blocked is one that cannot be run because it is waiting for some specific event to occur. Threads block for many reasons: they attempt to read data (e.g. from a socket) when no data is available; they execute a thread-blocking method (e.g. the sleep(), wait(), or join() methods); or they attempt to acquire a synchronization lock that another thread already holds. We have seen APIs that also block, but internally those methods are all executing the wait() method.

3.7.2.4 Exiting

A thread is in the exiting state once its run() method returns (or its deprecated stop( ) method has been called).

The basic process of thread scheduling is essentially the same whether it is performed by the Java virtual machine or the underlying operating system. Our intent here is to provide an illustration of how thread scheduling works, not to provide a blueprint of how any particular thread scheduler is actually implemented.

We can conceive that a thread scheduler keeps track of all the threads on which it operates by using linked lists; every thread is on a list that represents the state of the thread. A Java thread can have one of 11 priorities, so we conceive of 14 linked lists: one for all threads in the initial state, one for all threads in the blocked state, one for all threads in the exiting state, and one for each priority level. The list of threads at a given priority level represents only those threads that are currently in the runnable state: a thread in the runnable state at priority 7 is placed on the priority 7 list, but when the thread blocks, it moves to the blocked linked list. We're speaking here of having 11 priorities, but that number is a Java abstraction:

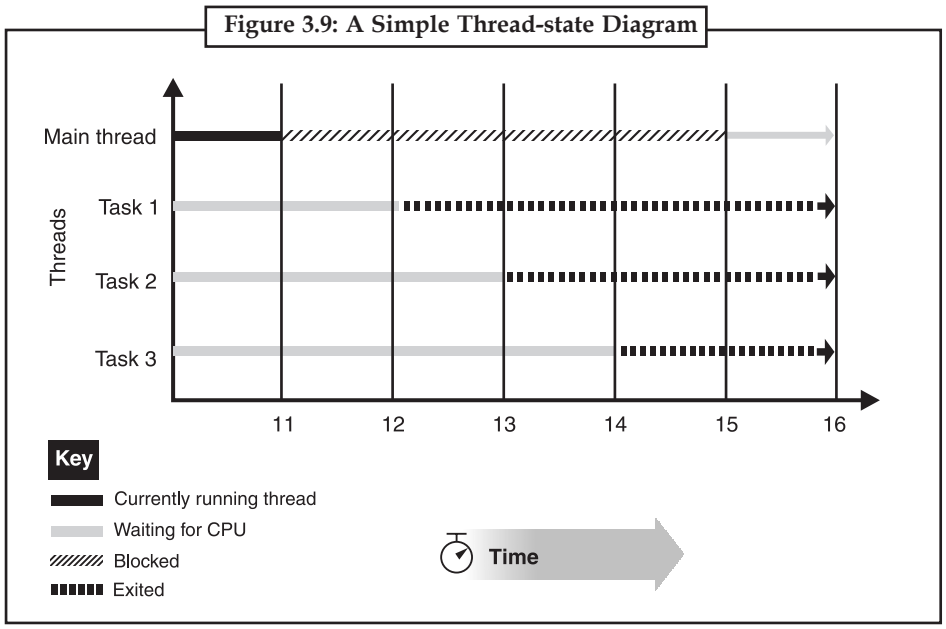
Notes

an operating system may have more or fewer priorities than that (but conceptually, each would still have its own linked list).

For simplicity, we conceive of these threads as being on an ordered list; in reality, they may be held in simple pools. Keeping the threads in a linked list implies that threads will be selected to run in a particular order. While that is a useful way of thinking about the process, it is not necessarily the way an implementation may work. Let us see how this scheduling will occur with the example we show at the beginning of the chapter. That example has a total of four threads: the initial thread (which executes the main( ) method) and the three task threads we started. In fact, as we have mentioned, there are more threads because the virtual machine starts various background threads (like the garbage collection thread). But for our discussion, we will consider only the four threads that are executing our code.

The threads that calculate a Fibonacci number never block: they move from the initial state to the runnable state to the executing state. The main thread is in the runnable state and then enters the blocking state when it executes the join( ) method to wait for the other threads. The second time that we run the program, the state of the threads follows the transition path shown in Figure 3.9. The main thread is the currently running thread until it blocks at time T1. At that point, one of the task threads becomes the currently running thread; it remains the currently running thread until time T2 when it finishes and transitions to the exiting state. Another task thread becomes the currently running thread, and the cycle continues until all threads have completed.

Figure 3.9: A Simple Thread-state Diagram



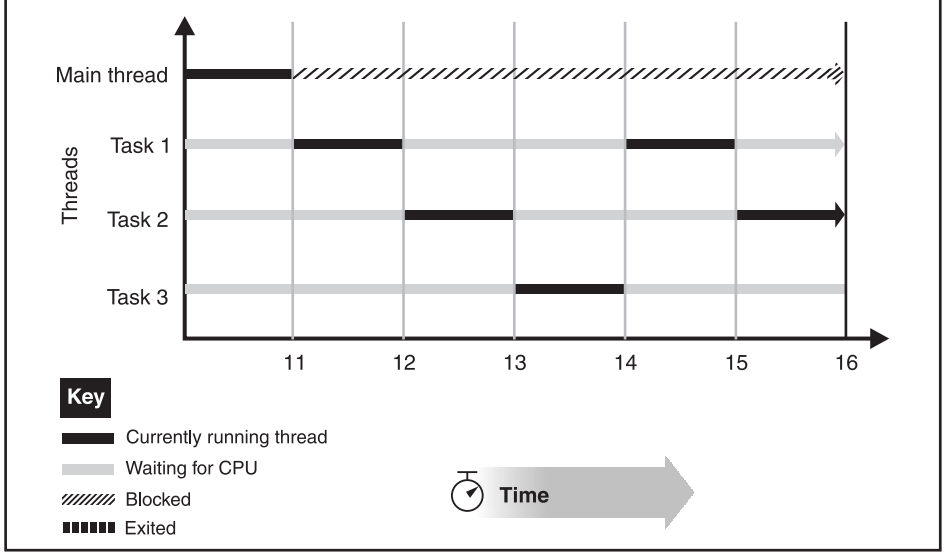
That explains the output that we see when we run the program for a second time: everything (including the output) proceeds sequentially. So why is the output different the first time we run the example?

The first time we run the example, we do so, on a typical operating system. The thread scheduler on that OS, in addition to being priority-based and preemptive, is also time-slicing. That means when threads are waiting for the CPU, the operating system allows one of them

Notes

to run for a very short time. It then interrupts that thread and allows a second thread to run for a very short time, and so on. A portion of the thread transitions on such an operating system is shown in Figure 3.10.

Figure 3.10: Thread States with OS Scheduling



Java does not mandate that its threads be time-sliced, but most operating systems do so. There is often some confusion in terminology here: preemption is often confused with time-slicing. In fact, preemption means only that a higher-priority thread runs instead of a lower-priority one, but when threads have the same priority, they do not preempt each other. They are typically subject to time-slicing, but that is not a requirement of Java.

There is one other important point about these two figures. In our first figure, the time points (T1, T2, and so on) are relatively far apart. The time transitions in that case are determined when a particular thread changes state: when the main thread changes to the blocked state, a task thread changes to become the currently running thread. When that thread changes to the exiting state, a second task thread changes to become the currently running thread and so on. In the second case, the time transitions occur at a much shorter interval, on the order of a few hundred milliseconds or so. In this case, the transitions of the threads between currently running and waiting for CPU are imposed by the operating system and not as a result of anything the thread itself has done. Of course, if a thread voluntarily changes to the exiting or waiting state, a transition occurs at that point as well.

3.8 Priority Exceptions

When an operating system schedules Java threads, it may choose to run a lower-priority thread instead of a higher-priority thread in two instances, described next.

Priority inversion: In a typical priority-based threading system, something unusual occurs when a thread attempts to acquire a lock that is held by a lower-priority thread: because the higher-priority thread becomes blocked, it temporarily runs with an effective priority of the lower-priority thread. Suppose that we have a thread with a priority of 8 that wants to acquire a lock that is held by a thread with a priority of 2. Because the priority 8 thread is waiting for the priority 2 threads to release the lock, it ends up running with an effective priority of 2. This is known as priority inversion.

Notes

Priority inversion is often solved by priority inheritance. With priority inheritance, a thread that holds a lock that is wanted by a thread with a higher priority has its priority temporarily and silently raised: its new priority becomes the same as the priority of the thread that it is causing to block. When the thread releases the lock, its priority is lowered to its original value. The goal of priority inheritance is to allow the high-priority thread to run as soon as possible. It is a common feature of operating systems, and Java virtual machines running on those operating systems are subject to it. However, it is not a requirement of the Java specification.

3.8.1 Complex Priorities

The second case involves the priority assigned to threads by the operating system. We mentioned that Java has 11 priority levels (10 of which are available to developers), but this is an abstraction of the Java language. Operating systems usually have many more priorities. More important, though, is that the priority that the operating system assigns to a thread is a complex formula that takes many pieces of information into account. A simple version of this formula might be this:

Real Priority = Java Priority + SecondsWaiting For CPU

This type of formula accounts for the length of time that the thread has been waiting for a CPU. After a sufficient amount of time has passed, a thread with a Java priority of 3 has a real priority that is higher than a currently running Java thread with a priority of 5. This gives the priority 3 thread an opportunity to run, even though it has a lower priority than other unblocked threads.

Complex priorities are advantageous because they help to prevent thread starvation. Without such a model, a low-priority thread would have to wait for all other high-priority threads to block before it is given a chance to execute; it is likely that it might have to wait forever. With complex priorities, it can still run much less often than it is higher-priority peers, but at least it will run sometimes.

On the other hand, complex priorities mean that you cannot guarantee thread scheduling. In particular, you cannot use thread priorities to try and prevent race conditions in data access: a lower-priority thread can interrupt a higher-priority thread while it is in the process of updating shared data. You also cannot use thread priorities to ensure a certain order of execution between tasks.



Lab Exercise

1. C Program For Round Robin Scheduling Method
2. C Program For Priority Scheduling Algorithm
3. C Program for Shortest Job First Scheduling Algorithm



Case Study

Applying Process Design Principles

Dan Madison is a principal in Value Creation partners. He facilitates process improvement using lean, six sigma, reengineering, and continuous improvement techniques. Dan is the author of Process Mapping, Process Improvement and Process Management.

Dan Madison has studied what a business process should look like for fifteen years. He studied what major corporations did in process improvement that made them successful and distilled his findings into design principles that anyone can use. He has come up with 38 design principles that apply to all business processes.

Contd...

Notes

His first example was from the permit and inspection department for the city of San Jose, California. Before the redesign, it had taken more than a month to get their permits approved. There was no money available from the city for process improvement. Madison helped the city develop a new permit form and process using four "lenses," or pictures of how the processes were done and how the results were judged. These were:

Frustration (as experienced by the people in the process)

Quality

Time

Cost

Frustration level is very important because Madison says that it has a high correlation to quality and there is usually immediate buy-in from people within the process.

After finding out the frustration level, it is important to find out why. In the permit process, the main problem was incomplete information on the permit forms, so the process was amended to ensure that all the information that was needed was in place from the beginning.

The developers needed to have a single point of contact with the city, in order to find out the information they needed. The city also began to cluster the permits so that similar projects had their own process. This meant that a developer who wanted to build a high rise had a different process and form than a homeowner who wanted to put in a swimming pool. It was found by studying the processes that the no-permit-needed people and the simple permits were found to take half the permit load for the city. With minor changes and adjustment, these customers were able to get what they needed from the city in just a few hours. Previously they had waited days and weeks.

In designing the new processes around the design principles, the first thing needed was to make sure all the required information was there. There were 14 different permits. One of the first things to work out was who reviewed the applications to see what was needed. The city hired and trained from within five generalist engineers to do the reviews. Each engineer had his own team, and each team was self-sufficient.

The most important Design Principles are:

Initially, design work flow around value adding activities, not functions or departments. Work is performed where it makes the most sense.

Provide a single point of contact for customers and suppliers whenever possible.

Consider every hand off as an opportunity for error. Have as few people as possible involved in the performance of a process.

If things coming into the process naturally cluster, create a separate process for each cluster. Redesign the process first, then automate it.

Bring downstream information needs upstream. Capture information once at the source and share it widely. Ensure 100% quality at the beginning of the process.

Ensure a continuous flow of the "main sequence" (those activities that directly add value to the customer-nothing should slow the value-added steps)

Look for places to use or create a "generalist" instead of multiple specialists. Push decision-making down to the lowest levels that make sense. Use simulation, practice, or role play to test new process designs risk free. If your process deals with complexity, then consider using teams. Co-locate the teams. If you can't do this, then network them.

Contd...

Notes

The people who work in the process should be very involved in the analysis, design, and implementation of improvements.

Create a process consultant for cross-functional processes.

It is important to look closely at every hand off, from one person or department to another because each one is an opportunity for errors to come in. Minimize the hand offs. And always remember, don't automate the as-is process. Improve the processes first, then automate them. Insure 100% quality at the front end because it all starts there.

Madison said that San Jose had asked the developers what they considered a good turn-around time was, and they answered that if they got their approvals within two weeks instead of a month, they would be happy. Using the new generalist teams, San Jose got the permit approval time down to two or three days. This result made the developers ecstatic.

This was first done back in 1995 just as the building boom was ramping up. Just before 2000 Madison checked back with San Jose and found that they were handling double the amount of work they did a few years earlier, and with no increase in staff.

Questions:

1. Who is Dan Madison?
2. Explain important Design Principles.

Self Assessment

Fill in the blanks:

6. The number of processes completed per unit time are known as ..... .
7. ..... in the sum of periods spent waiting in the ready queue.
8. A ..... process is one which is repeatedly executed once in each period.

True or False:

9. On-line scheduling algorithms does not compute a schedule in real-time as processes arrive.
10. Runnable state in a thread can be thought of as a default state.

3.9 Summary

- Thread is a single sequence stream within in a process. In this method, the kernel knows about and manages the threads.
- Context Switch ITT-VIS has added support for using threads internally in IDL to accelerate specific numerical computations on multi-processor systems.
- The concept of multi-threading involves an operating system that is multi-thread capable allowing programs to split tasks between multiple execution threads.

Notes

3.10 Keywords

***Process Management*:** Process management is a series of techniques, skills, tools, and methods used to control and manage a business process within a large system or organization.

***Threads*:** A thread is a single sequence stream within in a process.

***Process*:** A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS).

***Kernel*:** The kernel is the central component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level.

***Context Switch*:** A context switch is the computing process of storing and restoring state (context) of a CPU so that execution can be resumed from the same point at a later time.

Multitasking: Multitasking is the ability of an operating system to execute more than one program simultaneously.

***The Cost of Context Switching*:** Context switching represents a substantial *cost* to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system.

***BIOS*:** The BIOS software is built into the PC, and is the first code run by a PC when powered on ('boot firmware'). The primary function of the BIOS is to load and start an operating system.

***CPU Scheduling*:** CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

***Scheduling Algorithm*:** A scheduling algorithm is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth).

3.11 Review Questions

1. What is a thread? Describe the differences among short-term, medium-term, and long-term scheduling.
2. Provide two programming examples in which multi-threading does *not* provide better performance than a single-threaded solution.
3. Describe the actions taken by a thread library to context switch between user-level threads.
4. Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?
5. Which of the following components of program state are shared across threads in a multi-threaded process?

   (*a*) Register values (*b*) Heap memory

   (*c*) Global variables (*d*) Stack memory
6. Can a multi-threaded solution using multiple user-level threads achieve better performance on multi-processor system than on a single-processor system?

Notes

7. Consider multi-processor system and a multi-threaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.
   (*a*) The number of kernel threads allocated to the program is less than the number of processors.
   (*b*) The number of kernel threads allocated to the program is equal to the number of processors.
   (*c*) The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user level threads.
8. Write a multi-threaded Java, Pthreads, or Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.
9. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?
10. Discuss how the following pairs of scheduling criteria conflict in certain settings.
    (*a*) CPU utilization and response time
    (*b*) Average turnaround time and maximum waiting time
    (*c*) I/O device utilization and CPU utilization
11. Which of the following scheduling algorithms could result in starvation?
    (*a*) First-come, first-served
    (*b*) Shortest job first
    (*c*) Round robin
12. Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?
13. Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:
    (*a*) FCFS
    (*b*) RR
    (*c*) Multilevel feedback queues
14. Using the Windows XP scheduling algorithm, what is the numeric priority of a thread for the following scenarios?
    (*a*) A thread in the REALTIME PRIORITY CLASS with a relative priority of HIGHEST.
    (*b*) A thread in the NORMAL PRIORITY CLASS with a relative priority of NORMAL.
    (*c*) A thread in the HIGH PRIORITY CLASS with a relative priority of ABOVE NORMAL.
15. Consider the scheduling algorithm in the Solaris operating system for time sharing threads:
    (*a*) What is the time quantum (in milliseconds) for a thread with priority 10? With priority 55?

Notes

(*b*) Assume a thread with priority 35 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?

(*c*) Assume a thread with priority 35 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

16. What are Input and Output devices?

Answers to Self Assessment

1. (*a*)	2. (*c*)	3. (*c*)	4. (*c*)	5. (*d*)
6. throughput	7. Waiting time	8. periodic	9. False	10. True

3.12 Further Readings


Books

Introduction to Operating Design and Implementation, by Michael Kifer, Scoott A. Smolka.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.


Online link

wiley.com/coolege.silberschatz

Notes

Unit 4: Process Management-III

CONTENTS

Objectives
Introduction
4.1 Concept of Critical Section
4.1.1 Basic Concept
4.2 The Critical Section Problem
4.3 Way to Handle Critical Section Problem
4.4 Safety-Critical Operating Systems
4.4.1 Terminology
4.4.2 Memory Protection
4.4.3 Fault Tolerance and High Availability
4.4.4 Mandatory vs. Discretionary Access Control
4.4.5 Guaranteed Resource Availability: Space Domain
4.4.6 Guaranteed Resource Availability: Time Domain
4.4.7 Schedulability
4.4.8 Interrupt Latency
4.4.9 Bounded Execution Times
4.4.10 Priority Inversion
4.4.11 Changing Requirements
4.5 Semaphores
4.5.1 Producer-Consumer Problem Using Semaphores
4.6 Deadlock Concept and Handling
4.6.1 System Model
4.6.2 Deadlock Characterization
4.6.3 Resource-Allocation Graph
4.6.4 Methods for Handling Deadlocks
4.6.5 Deadlock Prevention
4.6.6 Deadlock Avoidance
4.6.7 Mutual Exclusion
4.6.8 Hold and Wait
4.6.9 No Preemption
4.6.10 Circular Wait
4.6.11 Safe State
4.6.12 Resource-Allocation Graph Algorithm
4.6.13 Banker's Algorithm
4.6.14 Resource-Request Algorithm
4.6.15 Deadlock Detection
4.6.16 Single Instance of Each Resource Type

Notes

4.6.17 Several Instances of a Resource Type
4.6.18 Detection-Algorithm Usage
4.6.19 Recovery from Deadlock
4.6.20 Process Termination
4.6.21 Abort All Deadlocked Processes
4.6.22 Abort One Process at a Time Until the Deadlock Cycle is Eliminated
4.6.23 Resource Preemption
4.7 Summary
4.8 Keywords
4.9 Review Questions
4.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain meaning of process management-III
- Understand concept of critical section
- Discuss the critical-section problem
- Explain way to handle critical-section problem
- Understand safety-critical operating systems
- Discuss semaphores
- Explain deadlock concept and handling

Introduction

A process is a program in execution. A process must have system resources, such as memory and the underlying CPU. The kernel supports the illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute. On a multiprocessor, multiple processes may really execute concurrently. This unit describes the composition of a process, the method that the system uses to switch between processes, and the scheduling policy that it uses to promote sharing of the CPU. It also introduces process creation and termination, and details the signal facilities and process-debugging facilities.

4.1 Concept of Critical Section

4.1.1 Basic Concept

If a data item is shared by a number threads, race conditions could occur if the shared item is not protected properly. The easiest protection mechanism is a lock. In general, if a set of data items must be protected so that at any time there is no more than one thread can have access to it, we can associate the set of data items with a lock. The use of locks is actually very easy. For every thread, before it accesses the set of data items, it acquires the lock. Once the lock is successfully acquired, the thread becomes the owner of that lock and the lock is locked. Then, the owner can access the protected items. After this, the owner must release the lock and the

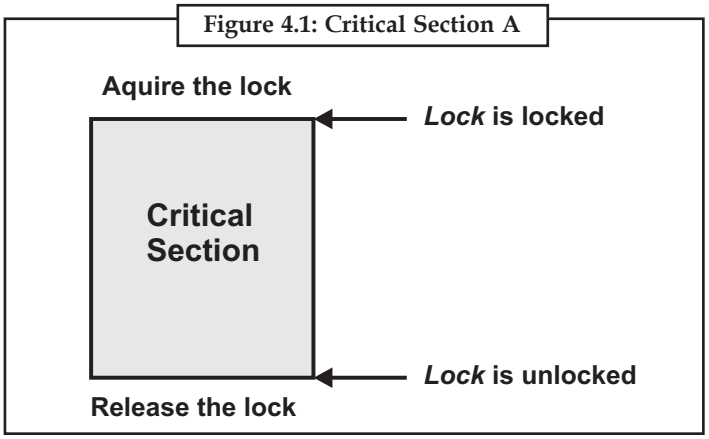
Notes

lock becomes unlocked. It is possible that while the owner is accessing one of the protected data items and another thread comes of course, this second thread must acquire that lock. However, since the lock is locked, this request is unsuccessful and the requesting thread will be suspended and queued at the lock. When the lock is released by its owner, one of the waiting threads will be allowed to continue and locks the lock.

This mechanism can be seen in everyday life. For example, on an airplane, before you use its lavatory, you check to see if it is locked. If it is, you join the waiting line; otherwise, you enter and lock the door. Once the door is locked, you are protected from the intrusion of anybody else (i.e. mutual exclusion). When you exit, you unlock the door so that one of those waiting can enter. There could be more than one waiting persons, and who will be allowed to enter and lock the door depends on some queuing policy (e.g., first-in-first-out). But, a good programmer should not make any assumption about this policy.

Therefore, the use of a lock simply establishes a critical section as shown below. Before entering a critical section, a thread acquires a lock. If it is successful, this thread enters the critical section and the lock is locked. As a result, all subsequent acquiring requests will be queued until the lock is unlocked. In this way, the owner of the lock (i.e., the thread that successfully acquired the lock) is the only thread that can execution the instructions of the indicated critical section. At the end of the execution of the instructions in a critical section, the owner releases the lock, and, at this point, the lock is unlocked, allowing the next thread to enter this critical section. Therefore, mutual exclusion is guaranteed. Because of this, a lock is also usually referred to as a mutex for mutual exclusion.

Figure 4.1: Critical Section A

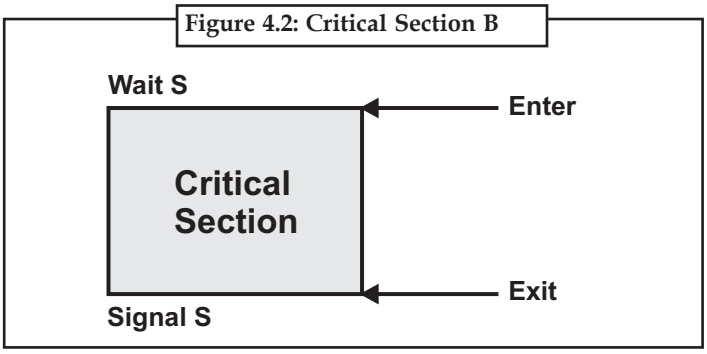


In general, there are a number of restrictions to the use of locks, although not all systems enforce the same set of restrictions. However, it would be very helpful for a programmer to know the possible restrictions. The first restriction is only **the owner can release the lock**. This is a very natural requirement. Imagine the following situation. Suppose thread **A** is the current owner of lock **L** and thread **B** is a second thread who wants to lock the lock. If a non-owner can unlock a lock, thread **B** can unlock the lock that thread **A** owns, and, hence, either both threads may be executing in the same critical section, or thread **B** preempts thread **A** and executes the instructions of the critical section. However, both are not very secured ways of protecting the shared items. Thus, in most systems, **Thread Mentor** included, only the owner of a lock can release the lock.

The second restriction is **recursive lock acquisition is not allowed**. This means the current owner of the lock is not allowed to acquire the same lock again. More precisely, if thread **A** currently owns lock **L**. If thread Awants to own lock **L** again, it must release lock **L** and re-acquire lock it again. Some systems permit a lock to be acquired recursively because this is useful in some applications; however, **Thread Mentor** does not allow this to happen because we believe that a programmer must know all the fundamentals before s/he starts to do something strange such as acquiring the same lock recursively.

Notes

Figure 4.2: Critical Section B



The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the Critical Section. To avoid race conditions and flawed results, one must identify codes in Critical Sections in each thread. The characteristic properties of the code that form a Critical Section are:

- Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in "read-updata-write" fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it is possibly in use by another thread.

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.


Did u know?

Part of the program where the shared memory is accessed is called the Critical Section.

4.2 The Critical Section Problem

n processes all competing to use some shared data. Each process has a code segment, called critical section, in which the shared data is accessed. Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section. Structure of process Pi repeat entry section critical section exit section remainder section until false.

4.3 Way to Handle Critical Section Problem

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Notes

3. **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. Assume that each process executes at a nonzero speed. No assumption concerning relative speed of the n processes.

Initial Attempts to Solve Problem

Only 2 processes, P0 and P1

General structure of process Pi (other process Pj)

repeat

entry section

critical section

exit section

remainder section

until false;

Processes may share some common variables to synchronize their actions.

Algorithm 1

Shared variables:

– var turn: (0..1);

initially turn = 0

– turn = i) Pi can enter its critical section

Process Pi

repeat

while turn =6 i do no-op;

critical section

turn := j;

remainder section

until false;

Satises mutual exclusion, but not progress.

Algorithm 2

Shared variables

– var ag: array [0..1] of boolean;

initially ag[0] = ag[1] = false.

– ag[i] = true) Pi ready to enter its critical section

Process Pi

repeat

ag[i] := true;

while ag[j] do no-op;

critical section

Notes

ag[i] := false;

remainder section

until false;

Does not satisfy the mutual exclusion requirement.

Algorithm 3

Combined shared variables of algorithms 1 and 2.

Process Pi

repeat

ag[i] := true;

turn := j;

while (ag[j] and turn=j) do no-op;

critical section

ag[i] := false;

remainder section

until false;

Meets all three requirements; solves the critical-section

problem for two processes.

Bakery Algorithm

Critical section for n processes Before entering its critical section, process receives a number.

Holder of the smallest number enters the critical section.

If processes Pi and Pj receive the same number, if i < j, then

Pi is served rst; else Pj is served rst. The numbering scheme always generates numbers in

increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Notation < lexicographical order (ticket #, process id #)

– (a,b) < (c,d) if a < c or if a = c and b < d

– max(a0, . . . , an1) is a number, k, such that k ai

for i = 0,

n 1 , . . .

Shared data

var choosing: array [0..n1] of boolean;

number: array [0..n1] of integer;

Data structures are initialized to false and 0, respectively

Bakery Algorithm (Cont.)

repeat

choosing[i] := true;

number[i] := max(number[0], number[1], ..., number[n 1])+1;

Notes

```
choosing[i] := false;
for j := 0 to n  1
do begin
while choosing[j] do no-op;
while number[j] =6 0
and (number[j],j) < (number[i], i) do no-op;
end;
critical section
number[i] := 0;
remainder section
until false;
```

Synchronization Hardware

Test and modify the content of a word atomically.

```
function Test-and-Set (var target: Boolean): Boolean;
begin
Test-and-Set := target;
target := true;
end;
```

Mutual Exclusion with Test-and-Set

```
Shared data: var lock: Boolean (initially false)
Process Pi
repeat
while Test-and-Set(lock) do no-op;
critical section
lock := false;
remainder section
until false;
```



Task Explain steps of Bakery Algorithm.

Self Assessment

Multiple choice questions:

1. Which of the following are(is) Language Processor(s)?
   (*a*) Assembles (*b*) Compilers
   (*c*) Interpreters (*d*) All of the above

Notes

2. To avoid the race condition, the number of processes that may be simultaneously inside their critical section is

   (*a*) 8 (*b*) 1

   (*c*) 16 (*d*) 0

3. A critical region

   (*a*) is a piece of code which only one process executes at a time

   (*b*) is a region prone to deadlock

   (*c*) is a piece of code which only a finite number of processes execute

   (*d*) is found only in Windows NT operation system

4. The solution to Critical Section Problem is: Mutual Exclusion, Progress and Bounded Waiting.

   (*a*) The statement is false. (*b*) The statement is true.

   (*c*) The statement is contradictory. (*d*) None of the above

4.4 Safety-Critical Operating Systems

The successful design of safety-critical systems is difficult and demands significant attention to detail. Fortunately, an operating system's emphasis on protection and resource guarantees can make the job of application developers less arduous.

Whether you are designing a telecom switch, a piece of medical equipment, or one of the many complex systems aboard an aircraft, certain critical parts of the application must be able to operate under all conditions. Indeed, given the steadily increasing speed of processors and the economically-driven desire to run multiple applications, at varying levels of criticality, on the same processor, the risks continue to grow. Consider a blood gas analyzer used in an intensive care unit. The analyzer may serve two distinct purposes. First, it monitors the level of oxygen and other gasses in the patient's bloodstream, in real time. If any monitored gas reaches a dangerously low or high level, the analyzer should produce an audible alarm or take some more direct, intervention, action. But the device may have a second use, offering a historical display of gas levels for "offline" analysis. In such a system, data logging, data display, and user interface threads may compete with the critical monitoring and alarm threads for use of the processor and other resources. In order for threads of varying importance to safely coexist in the same system, the operating system that manages the processor and other resources must be able to properly partition the software to guarantee resource availability. The key word here is guarantee. Post-design, post-implementation testing cannot be counted on. Safety-critical systems must be safe at all times.

4.4.1 Terminology

The following terms are used in this article:

- **Thread:** A lightweight unit of program execution
- **Process:** A heavyweight unit consisting primarily of a distinct address space, within which one or more threads execute
- **Kernel:** The portion of an operating system that provides core system services such as scheduling, thread synchronization, and interprocess communication

Notes

4.4.2 Memory Protection

Fault tolerance begins with memory protection. For many years, microprocessors have included on-chip memory management units (MMU) that enable individual threads of software to run in hardware-protected address spaces. But many commercial real-time operating systems never enable the MMU, even if such hardware is present in the system.

When all of an application's threads share the same memory space, any thread could-intentionally or unintentionally-corrupt the code, data, or stack of another thread. A misbehaved thread could even corrupt the kernel's own code or internal data structures. It is easy to see how a single errant pointer in one thread could easily bring down the entire system, or at least cause it to behave unexpectedly.

For safety and reliability, a process-based real-time operating system (RTOS) is preferable. To create processes with individual address spaces, the RTOS need only create some RAM-based data structures and enable the MMU to enforce the protections described therein. The basic idea is that a new set of logical addresses is "switched in" at each context switch. The MMU maps a logical address used during an instruction fetch or a data read or write to a physical address in memory through the current mapping. It also flags attempts to access illegal logical addresses, which have not been "mapped" to any physical address.

The cost of processes is the overhead inherent in memory access through a look-up table. But the payoff is huge. Careless or malicious corruption across process boundaries is rendered impossible. A bug in a user interface thread cannot corrupt the code or data of a more critical thread. It's truly a wonder that non-memory protected operating systems are still used in complex embedded systems where reliability, safety, or security are important.

Enabling the MMU has other benefits as well. One big advantage stems from the ability to selectively map and unmap pages into a logical address space. Physical memory pages are mapped into the logical space to hold the current process' code; others are mapped for data. Likewise, physical memory pages are mapped in to hold the stacks of threads that are part of the process. An RTOS can easily provide the ability to leave a page's worth of the logical addresses after each thread's stack unmapped. That way, if any thread overflows its assigned stack, a hardware memory protection fault will occur. The kernel will suspend the thread instead of allowing it to corrupt other important memory areas within the address space (like another thread's stack). This adds a level of protection between threads, even within the same address space.

Memory protection, including this kind of stack overflow detection, is often helpful during the development of an application. Programming errors will generate exceptions that are immediately detected and easily traceable to the source code. Without memory protection, bugs can cause subtle corruptions that are very difficult to track down. In fact, since RAM is often located at physical address zero in a flat memory model, even NULL pointer dereferences will go undetected! (Clearly, logical page zero is a good one to add to the "unmap list."). Another issue is that the kernel must protect itself against improper system calls.



Did u know?

The kernel must protect itself against improper system calls. Many kernels return the actual pointer to a newly created kernel object, such as a semaphore, to the thread that created it, as a handle. When that pointer is passed back to the kernel in subsequent system calls, it may be dereferenced directly. But what if the thread uses that pointer to modify the kernel object directly, or simply overwrites its handle with a pointer to some other memory. The results may be disastrous.

Notes

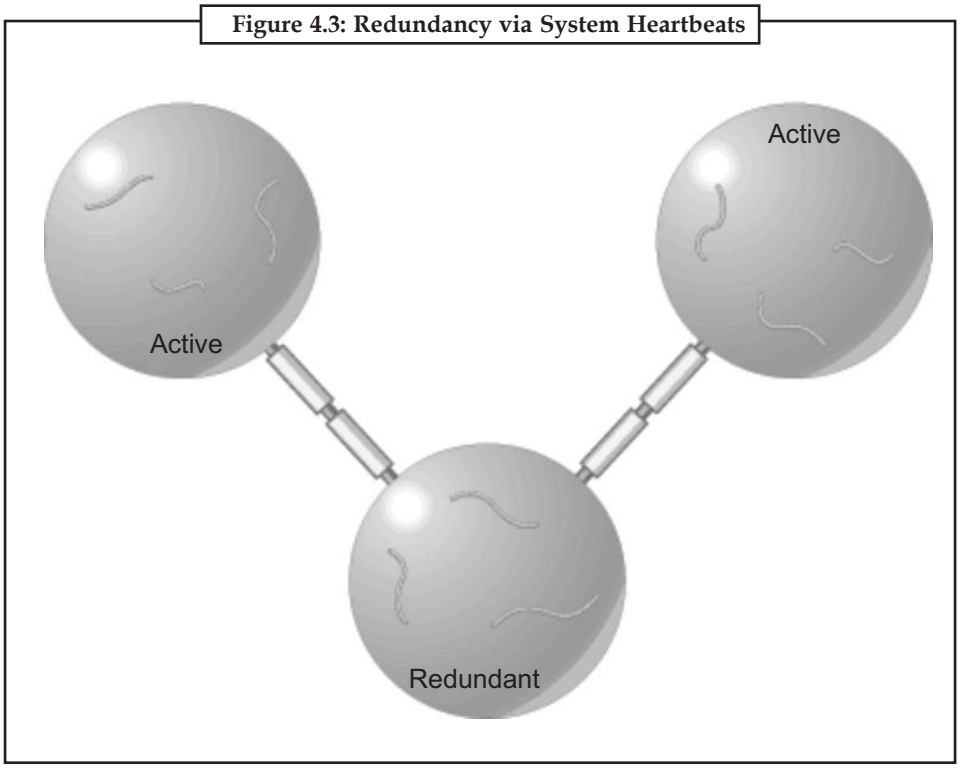
A bad system call should never be able to take down the kernel. An RTOS should, therefore, employ opaque handles for kernel objects. It should also validate the parameters to all system calls.

4.4.3 Fault Tolerance and High Availability

Even the best software has latent bugs. As applications become more complex, performing more functions for a software-hungry world, the number of bugs in fielded systems will continue to rise. System designers must, therefore, plan for failures and employ fault recovery techniques of course, the effect of fault recovery is application-dependent—a user interface can restart itself in the face of a fault, a flight-control system probably cannot. One way to do fault recovery is to have a supervisor thread in an address space all its own. When a thread faults (for example, due to a stack overflow), the kernel should provide some mechanism whereby notification can be sent to the supervisor thread. If necessary, the supervisor can then make a system call to close down the faulted thread, or the entire process, and restart it. The supervisor might also be hooked into a software "watchdog" setup, whereby thread deadlocks and starvation can be detected as well.

In many critical systems, high availability is assured by employing multiple redundant nodes in the system. In such a system, the kernel running on a redundant node must have the ability to detect a failure in one of the operating nodes. One method is to provide a built-in heartbeat in the interprocessor message passing mechanism of the RTOS. Upon system startup, a communications channel is opened between the redundant nodes and each of the operating nodes. During normal operation, the redundant nodes continually receive heartbeat messages from the operating nodes. If the heartbeat fails to arrive, the redundant node can take control automatically.

Figure 4.3: Redundancy via System Heartbeats



Notes

4.4.4 Mandatory vs. Discretionary Access Control

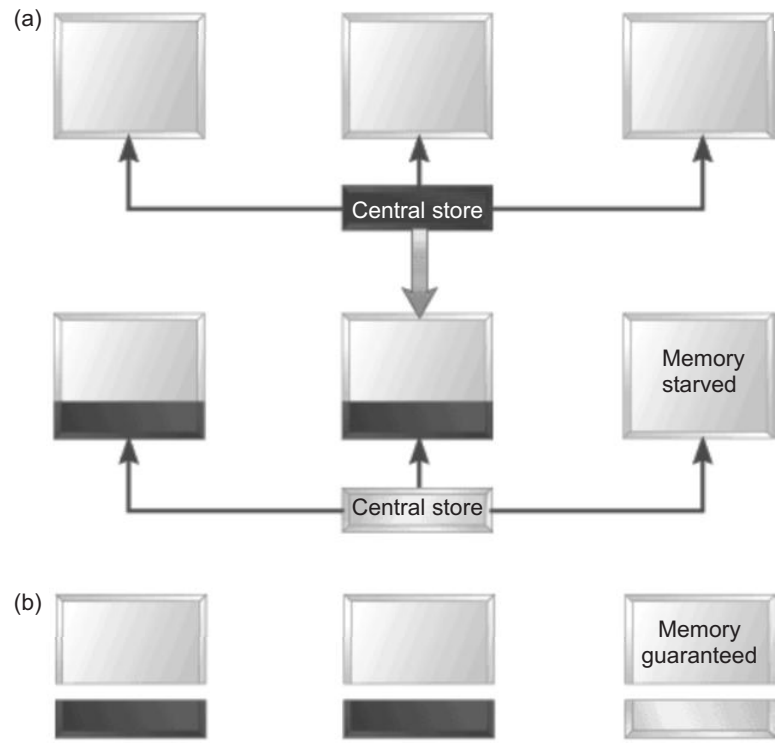
An example of a discretionary access control is a Unix file—a process or thread can, at its sole discretion, modify the permissions on a file, thereby permitting access to the file by another process in the system. Discretionary access controls are useful for some objects in some systems. An RTOS that is used in a safety- or security-critical system must be able to go one big step further and provide mandatory access control of critical system objects. For example, consider an aircraft sensor device, access to which is controlled by a flight control program. The system designer must be able to set up the system statically such that the flight control program and only the flight control program has access to this device. Another application in the system cannot dynamically request and obtain access to this device. And the flight control program cannot dynamically provide access to the device to any other application in the system. The access control is enforced by the kernel, is not circumventable by application code, and is thus mandatory. Mandatory access control provides guarantees. Discretionary access controls are only as effective as the applications using them, and these applications must be assumed to have bugs in them.

Task Differentiate between Mandatory vs. discretionary access control.

4.4.5 Guaranteed Resource Availability: Space Domain

In safety-critical systems, a critical application cannot, as a result of malicious or careless execution of another application, run out of memory resources. In most real-time operating systems, memory used to hold thread control blocks and other kernel objects comes from a central store.

Figure 4.4: (*a*) Before Memory Quotas (*b*) After



Notes

When a thread creates a new thread, semaphore, or other kernel object, the kernel carves off a chunk of memory from this central store to hold the data for this object. A bug in one thread could, therefore, result in a situation where this program creates too many kernel objects and the central store is exhausted (see Figure 4.4 a). A more critical thread could fail as a result, perhaps with disastrous effects.

In order to guarantee that this scenario cannot occur, the RTOS can provide a memory quota system wherein the system designer statically defines how much physical memory each process has (see Figure 4.4 b). For example, a user interface process might be provided a maximum of 128 KB and a flight control program a maximum of 196 KB. If a thread within the user interface process encounters the aforementioned failure scenario, the process may exhaust its own 128 KB of memory. But the flight control program and its 196 KB of memory are wholly unaffected. In a safety-critical system, memory should be treated as a hard currency—when a thread wants to create a kernel object, its parent process must provide a portion of its memory quota to satisfy the request. This kind of space domain protection should be part of the RTOS design. Central memory stores and discretionarily-assigned limits are insufficient when guarantees are required.

If an RTOS provides a memory quota system, dynamic loading of low criticality applications can be tolerated. High criticality applications already running are guaranteed to have the physical memory they will require to run. In addition, the memory used to hold any new processes should come from the memory quota of the creating process. If this memory comes from a central store, then process creation can fail if a malicious or carelessly written application attempts to create too many new processes. (Most programmers have either mistakenly executed or at least heard of a Unix "fork bomb," which can easily take down an entire system.) In most safety-critical systems, dynamic process creation will simply not be tolerated at all, and the RTOS should be configurable such that this capability can be removed from the system.

4.4.6 Guaranteed Resource Availability: Time Domain

The vast majority of RTOSes employ priority-based, preemptive schedulers. Under this scheme, the highest priority ready thread in the system always gets to use the processor (execute). If multiple threads are at that same highest priority level, they generally share the processor equally, via time slicing. The problem with this time slicing (or even run-to-completion) within a given priority level, is that there is no provision for guaranteeing processor time for critical threads.

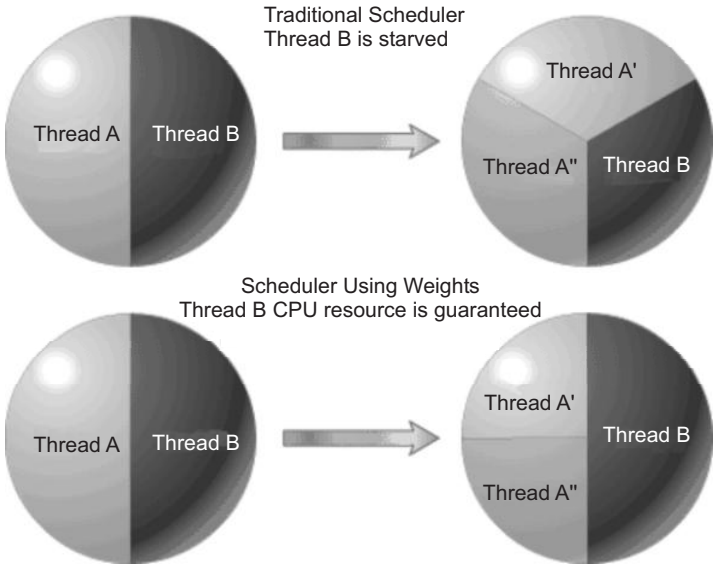
Consider the following scenario—the system includes two threads at the same priority level. Thread A is a non-critical, background thread. Thread B is a critical thread that needs at least 40% of the processor time to get its work done. Because Threads A and B are assigned the same priority level, the typical scheduler will time slice them so that both threads get 50% of the processor. At this point, Thread B is able to get its work done. Now suppose Thread A creates a new thread at the same priority level. Consequently, there are three highest priority threads sharing the processor. Suddenly, Thread B is only getting 33% of the processor and cannot get its critical work done. For that matter, if the code in Thread A has a bug or virus, it may create dozens or even hundreds of "confederate" threads, causing Thread B to get a tiny fraction of the runtime.

One solution to this problem is to enable the system designer to inform the scheduler of a thread's maximum "weight" within the priority level. When a thread creates another equal priority thread, the creating thread must give up part of its own weight to the new thread. In our previous example, suppose the system designer had assigned weight to Thread A and Thread B such that Thread A has 60% of the runtime and Thread B has 40% of the runtime. When Thread A creates the third thread, it must provide part of its own weight, say 30%. Now Thread A and

Notes

the new thread each get 30% of the processor time, but critical Thread B's 40% remains inviolate. Thread A can create many confederate threads without affecting the ability of Thread B to get its work done; Thread B's processor reservation is thus guaranteed. A scheduler that provides this kind of guaranteed resource availability in addition to the standard scheduling techniques is required in some critical embedded systems, particularly avionics.

Figure 4.5: Traditional Scheduler vs. Scheduler with Weights



The problem inherent in all schedulers is that they are ignorant of the process in which threads reside. Continuing our previous example, suppose that Thread A executes in a user interface process while critical Thread B executes in a flight control process. The two applications are partitioned and protected in the space domain, but not in the time domain. Designers of safety-critical systems require the ability to guarantee that the run-time characteristics of the user interface cannot possibly affect the run-time characteristics of the flight control system. Thread schedulers simply cannot make this guarantee. Consider a situation in which Thread B normally gets all the runtime it needs by making it higher priority than Thread A or any of the other threads in the user interface. Due to a bug or poor design or improper testing, Thread B may lower its own priority (the ability to do so is available in practically all kernels), causing the thread in the user interface to gain control of the processor. Similarly, Thread A may raise its priority above the priority of Thread B with the same effect.

A convenient way to guarantee that the threads in processes of different criticality cannot affect each other is to provide a process-level scheduler. Designers of safety critical software have noted this requirement for a long time. The process, or partition, scheduling concept is a major part of ARINC Specification 653, an Avionics Application Software Standard Interface.

The ARINC 653 partition scheduler runs partitions, or processes, according to a timeline established by the system designer. Each process is provided one or more windows of execution within the repeating timeline. During each window, all the threads in the other processes are not runnable; only the threads within the currently active process are runnable (and typically are scheduled according to the standard thread scheduling rules). When the flight control application's window is active, its processing resource is guaranteed; a user interface application cannot run and take away processing time from the critical application during this window.

Notes

Although not specified in ARINC 653, a prudent addition to the implementation is to apply the concept of a background partition. When there are no runnable threads within the active partition, the partition scheduler should be able to run background threads, if any, in the background partition, instead of idling. An example background thread might be a low priority diagnostic agent that runs occasionally but does not have hard real-time deadlines. Attempts have been made to add partition scheduling on top of commercial off-the-shelf operating systems by selectively halting all the threads in the active partition and then running all the threads in the next partition. Thus, partition switching time is linear with the number of threads in the partitions, an unacceptably poor implementation.



Did u know? The RTOS must implement the partition scheduler within the kernel and ensure that partition switching takes constant time and is as fast as possible.

4.4.7 Schedulability

Meeting hard deadlines is one of the most fundamental requirements of a real-time operating system and is especially important in safety-critical systems. Depending on the system and the thread, missing a deadline can be a critical fault.

Rate monotonic analysis (RMA) is frequently used by system designers to analyze and predict the timing behavior of systems. In doing so, the system designer is relying on the underlying operating system to provide fast and temporally deterministic system services. Not only must the designer understand how long it takes to execute the thread's code, but also any overhead associated with the thread must be determined. Overhead typically includes context switch time, the time required to execute kernel system calls, and the overhead of interrupts and interrupt handlers firing and executing.

All real-time operating systems incur the overhead of context switching. Lower context switching time implies lower overhead, more efficient use of available processing resources, and increased likelihood of meeting deadlines. A real-time operating system's context switching code is usually hand optimized for optimal execution speed.

4.4.8 Interrupt Latency

A typical embedded system has several types of interrupts resulting from the use of various kinds of devices. Some interrupts are higher priority and require a faster response time than others. For example, an interrupt that signals the kernel to read a sensor that is critical to an aircraft's flight control should be handled with the minimum possible latency. On the other hand, a typical timer tick interrupt frequency may be 60 Hz or 100 Hz. Ten milliseconds is an eternity in hardware terms, so interrupt latency for the timer tick interrupt is not as critical as for most other interrupts.

Most kernels disable all interrupts while manipulating internal data structures during system calls. Interrupts are disabled so that the timer tick interrupt cannot occur (a timer tick may cause a context switch) at a time when internal kernel data structures are being changed. The system's interrupt latency is directly related to the length of the longest critical section in the kernel.

In effect, most kernels increase the latency of all interrupts just to avoid a low priority timer interrupt. A better solution is to never disable interrupts in kernel system calls and instead to postpone the handling of an intervening timer tick until the system call completes. This strategy depends on all kernel system calls being short (or at least that calls that are not short are restartable), so that scheduling events can preempt the completion of the system call.

Notes

Therefore, the time to get back to the scheduler may vary by a few instructions (insignificant for a 60 Hz scheduler), but will always be short and bounded. It is much more difficult to engineer a kernel that has preemptible system calls in this manner, which is why most kernels do not do it this way.

4.4.9 Bounded Execution Times

In order to allow computation of the overhead of system calls that a thread will execute while doing its work, an RTOS should provide bounded execution times for all such calls. Two major problems involve the timing of message transfers and the timing of mutex take operations.

A thread may spend time performing a variety of activities of course, its primary activity is executing code. Other activities include sending and receiving messages. Message transfer times vary with the size of the data. How can the system designer account for this time? The RTOS can provide a capability for controlling whether transfer times are attributed to the sending thread or to the receiving thread, or shared between them. Indeed, the kernel's scheduler should treat all activities, not just the primary activity, as prioritized units of execution so that the system designer can properly control and account for them.

4.4.10 Priority Inversion

Priority inversion has long been the bane of system designers attempting to perform rate monotonic analysis, since RMA depends on higher priority threads running before lower priority threads. Priority inversion occurs when a high priority thread is unable to run because a mutex (or binary semaphore) it attempts to obtain is owned by a low priority thread, but the low priority thread is unable to execute and release the mutex because a medium priority thread is also runnable. The most common RTOS solution to the priority inversion problem is to support the priority inheritance protocol.

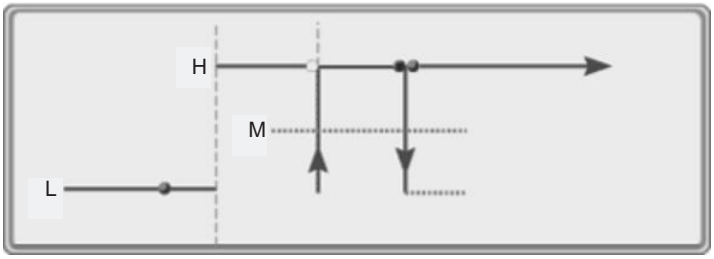
Figure 4.6: Priority Inversion



A mutex that supports priority inheritance works as follows—if a high priority thread attempts to take a mutex already owned by a low priority thread, the kernel automatically elevates the low priority thread to the priority of the high priority thread. Once the low priority thread releases the mutex, its priority will be returned to normal and the high priority thread will run. The dynamic priority elevation prevents a medium priority thread from running while the high priority thread is waiting; priority inversion is avoided. In this example, the critical section execution time (the time the low priority thread holds the mutex) is added to the overhead of the high priority thread.

Notes

Figure 4.7: Priority Inheritance

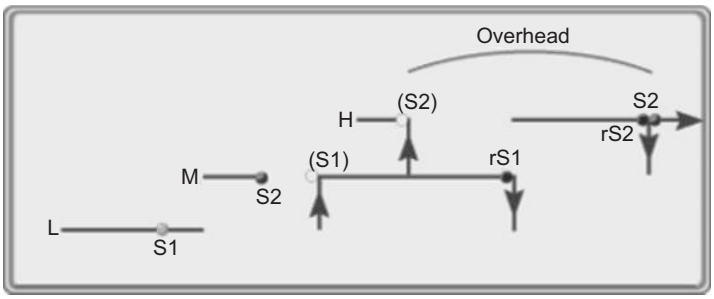


A weakness of the priority inheritance protocol is that it does not prevent chained blocking. Suppose a medium priority thread attempts to take a mutex owned by a low priority thread, but while the low priority thread's priority is elevated to medium by priority inheritance, a high priority thread becomes runnable and attempts to take another mutex already owned by the medium priority thread. The medium priority thread's priority is increased to high, but the high priority thread now must wait for both the low priority thread and the medium priority thread to complete before it can run again.

The chain of blocking critical sections can extend to include the critical sections of any threads that might access the same mutex. Not only does this make it much more difficult for the system designer to compute overhead, but since the system designer must compute the worst case overhead, the chained blocking phenomenon may result in a much less efficient system. These blocking factors are added into the computation time for tasks in the RMA analysis, potentially rendering the system unschedulable. This may force the designer to resort to a faster CPU or to remove functionality from the system.

Task Differentiate between Priority inheritance and Priority inversion.

Figure 4.8: Chained Blocking caused by Priority Inheritance

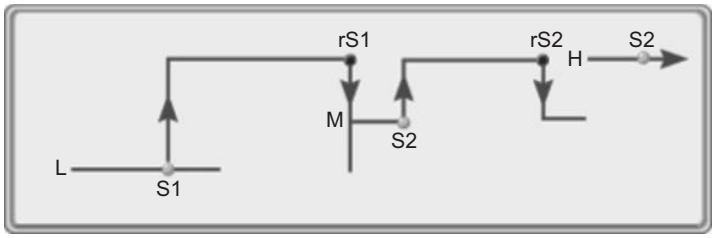


A solution called the priority ceiling protocol not only solves the priority inversion problem but also prevents chained blocking. In one implementation scheme (called the highest locker), each semaphore has an associated priority, which is assigned by the system designer to be the priority of the highest priority thread that might ever try to acquire that object. When a thread takes such a semaphore, it is immediately elevated to the priority of the semaphore. When the semaphore is released, the thread reverts back to its original priority. Because of this priority elevation,

Notes

no other threads that might contend for the same semaphore can run until the semaphore is released. It is easy to see how this prevents chained blocking.

Figure 4.9: Priority Ceilings



Several RTOSes provide support for both priority inheritance and priority ceilings, leaving the decision up to the system designer. Ada's protected objects implement the priority ceiling protocol.

4.4.11 Changing Requirements

Many of the real-time operating systems in use today were originally designed for software systems that were smaller, simpler, and ran on processors without memory protection hardware. With the ever-increasing complexity of applications in today's embedded systems, fault tolerance and high availability features have become increasingly important. Especially stringent are the requirements for safety-critical systems.

Fault tolerance begins with processes and memory protection, but extends to much more, especially the need to guarantee resource availability in the time and space domains. Kernel support for features like the priority ceiling protocol give safety-critical system designers the capabilities needed to maximize efficiency and guarantee schedulability in their systems.

4.5 Semaphores

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphoiinitislize'. Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only non-negative values. The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

P(S):IF S > 0

THEN S := S – 1

ELSE (wait on S)

The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

V(S):IF (one or more process are waiting on S)

THEN(let one of these processes proceed)

ELSE S := S +1

Notes

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operations has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within **P(S)** and **V(S)**.

If several processes attempt a P(S) simultaneously, only process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement. Semaphores solve the lost-wakeup problem.

4.5.1 Producer-Consumer Problem Using Semaphores

The solution to producer-consumer problem uses three semaphores, namely, full, empty and mutex.

Initialization

- Set full buffer slots to 0.
  i.e., semaphore full = 0.
- Set empty buffer slots to N.
  i.e., semaphore empty = N.
- For control access to critical section set mutex to 1.
  i.e., semaphore mutex = 1.

```
Producer ( )
WHILE (true)

produce-Item ( );
P (empty);
P (mutex);
enter-Item ( )
V (mutex)
V (full);

Consumer ( )
WHILE (true)
P (full)

P (mutex);

remove-Item ( );

V (mutex);

V (empty);

consume-Item (Item)
```



Did u know?

The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

4.6 Deadlock Concept and Handling

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a **deadlock**. Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part—"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Notes

In this unit, we describe methods that an operating system can use to prevent or deal with deadlocks. Most current operating systems do not provide deadlock-prevention facilities, but such features will probably be added soon. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multi-threaded programs, many more resources within a system, and the emphasis on long-lived file and database servers rather than batch systems.


Caution

Keep away your system from the deadlock condition for the best processing of the system.

4.6.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances. If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer. A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release:** The process releases the resource. The request and release of resources are system calls. Examples are the request and release device, open and close file, and allocate and free memory system calls. Request and release of other resources can be accomplished through the waif and signal operations on semaphores. Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in

Notes

deadlocks. To illustrate a deadlock state, we consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type. Deadlocks may also involve different resource types. For example, consider a system with one printer and one tape drive. Suppose that process Pi is holding the tape drive and process Pi is holding the printer. If Pi requests the printer and Pi requests the tape drive, a deadlock occurs.


Did u know?

A programmer who is developing multi-threaded applications must pay particular attention to this problem—Multi-threaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

4.6.2 Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual Exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular Wait:** A set {Po, PI, ..., P,) of waiting processes must exist such that Po is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, ..., PnPl is waiting for a resource that is held by P,, and P, is waiting for a resource that is held by Po. We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

4.6.3 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system. A directed edge from process Pi to resource type Rj is denoted by Pi -+ Rj; it signifies that process Pi requested an instance of resource type Ri and is currently waiting for that resource. A directed edge from resource type Ri to process Pi is denoted by Rj -+ Pi; it signifies that an instance of resource type Ri has been allocated to process Pi. A directed edge Pi -+ Rj is called a request edge; a directed

Notes

edge Rj + Pi is called an assignment edge. Pictorially, we represent each process Pi as a circle, and each resource type Ri as a square. Since resource type Ri may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square Xi, whereas an assignment edge must also designate one of the dots in the square. When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted. The resource-allocation graph shown in Figure depicts the following situation.

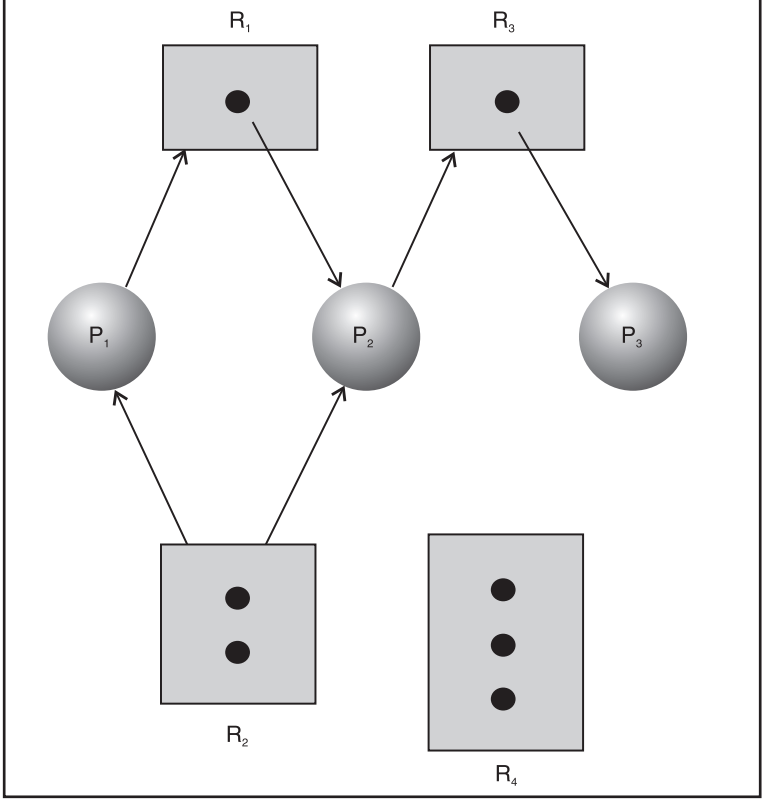
The sets **P**, R, and E:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Figure 4.10: Resource-Allocation Graphs



Notes

Process States

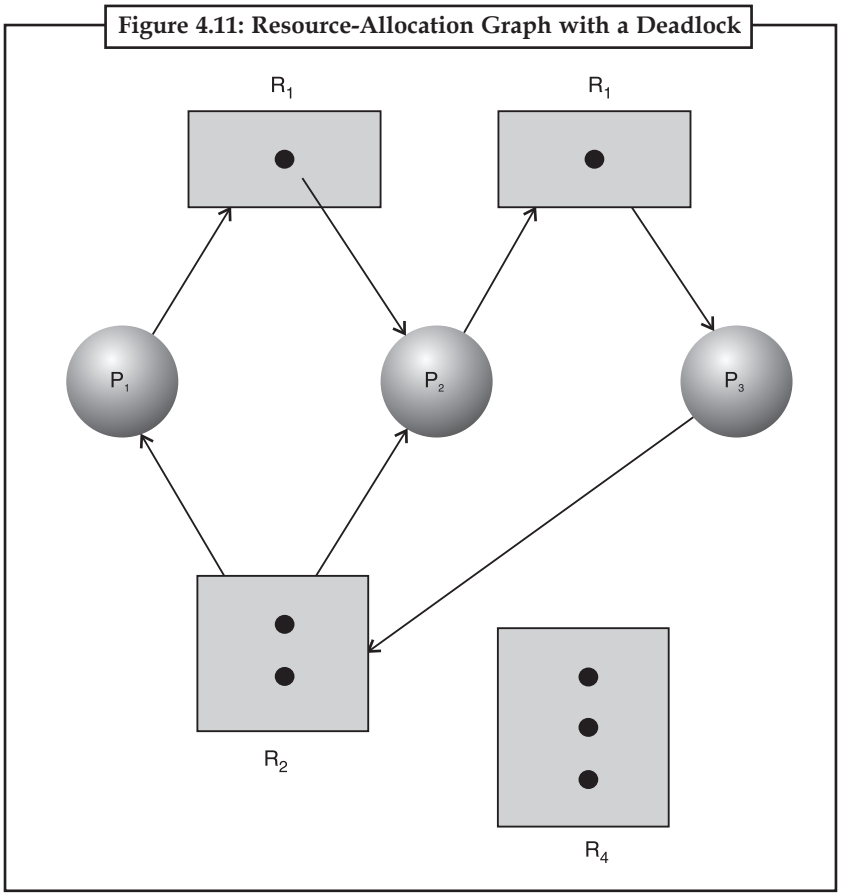
- Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock. To illustrate this concept, let us return to the resource-allocation graph depicted. Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 -+ **RL** is added to the graph. At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Figure 4.11: Resource-Allocation Graph with a Deadlock



Notes

Processes PI, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process PI or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1. Now consider the resource-allocation graph in Figure 4.11. In this example, we also have a cycle. However, there is no deadlock. Observe that process **P4** may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

4.6.4 Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state. We can allow the system to enter a deadlock state, detect it, and recover.

We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX. We shall elaborate briefly on each method. Then, we shall present detailed algorithms. To ensure that deadlocks never occur, the system can use either a deadlockprevention or a deadlock-avoidance scheme.

4.6.5 Deadlock Prevention

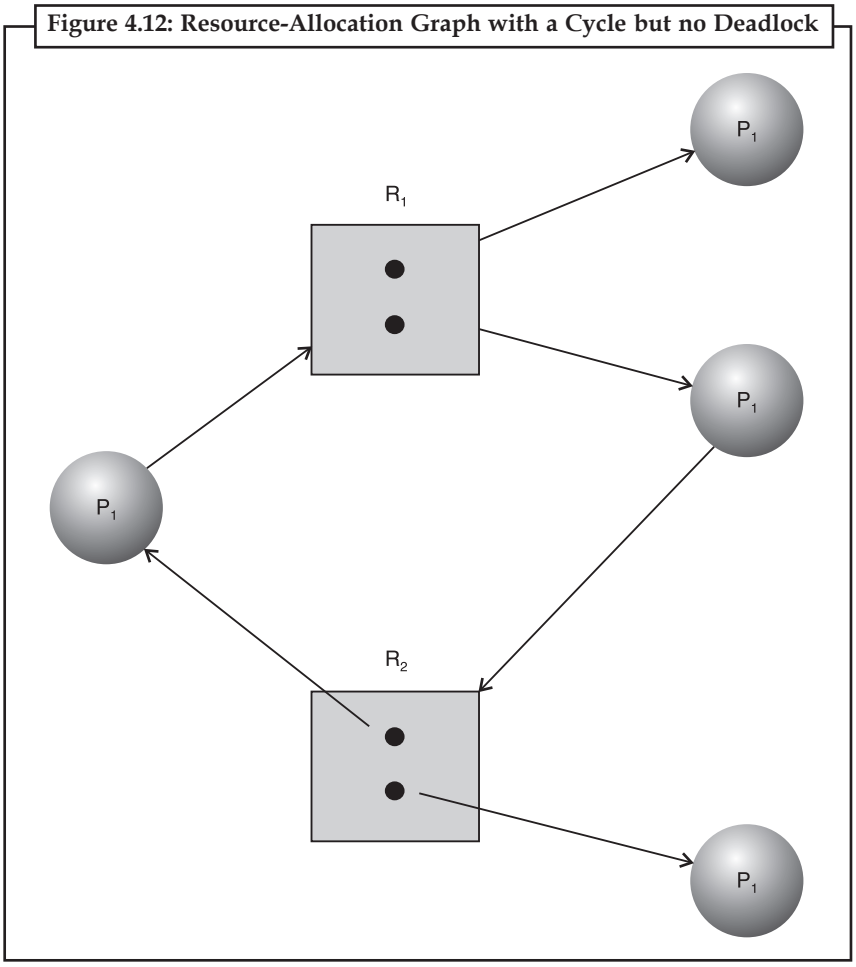
Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

4.6.6 Deadlock Avoidance

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in if a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in the deterioration of the system performance, because resources are being held by

Notes

Figure 4.12: Resource-Allocation Graph with a Cycle but no Deadlock



processes that cannot run, and because more and more processes, as they make requests for resources, enter a deadlock state. Eventually, the system will stop functioning and will need to be restarted manually. Although this method does not seem to be a viable approach to the deadlock problem, it is nevertheless used in some operating systems. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the costly deadlock-prevention, deadlock-avoidance, or deadlock-detection and recovery methods that must be used constantly. Also, in some circumstances, the system is in a frozen state but not in a deadlock state. As an example, consider a real-time process running at the highest priority (or any process running on a non-preemptive scheduler) and never returning control to the operating system. Thus, systems must have manual recovery methods for non-deadlock conditions, and may simply use those techniques for deadlock recovery.

4.6.7 Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition. Some resources are intrinsically non-sharable.

Notes

4.6.8 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated. To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end. The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates. These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

4.6.9 No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

4.6.10 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Notes

Let R = {XI, R2, ..., R,) be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R + N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1,

F(disk drive) = 5,

F(printer) = 12.

We can now consider the following protocol to prevent deadlocks. Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Xi. After that, the process can request instances of resource type Ri if and only if F(Rj) > F(Ri). If several instances of the same resource type are needed, a single request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) 2 F(Rj). If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {Po, PI, ..., P,), where Pi is waiting for a resource Xi, which is held by process Pi+l. (Modulo arithmetic is used on the indexes, so that P, is waiting for a resource R, held by Po.) Then, since process Pi+l is holding resource Ri while requesting resource Ri+l, we must have F(Ri) < F(Ri+1), for all i. But this condition means that F(Ro) < F(R1) < ... < F(R,) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible. Therefore, there can be no circular wait.



Notes The function F should be defined according to the normal order of usage of the resources in a system. For example, since the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

4.6.11 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of processes <PI, P2, ..., P,> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pi, with j < i. In this situation, if the resources that process Pi needs are not immediately available, then Pi can wait until all Pi have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe. A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs. The behaviour of the processes controls unsafe states. To illustrate, we consider a system with 12 magnetic tape drives and 3 processes—Po, PI, and P2. Process Po requires 10 tape drives, process PI may need as many as 4, and process P2 may need up to 9 tape drives. Suppose that, at time t0, process Po is holding 5 tape drives, process PI is holding 2, and process PZ is holding 2 tape drives. (Thus, there are 3 free tape drives.)

Notes

	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

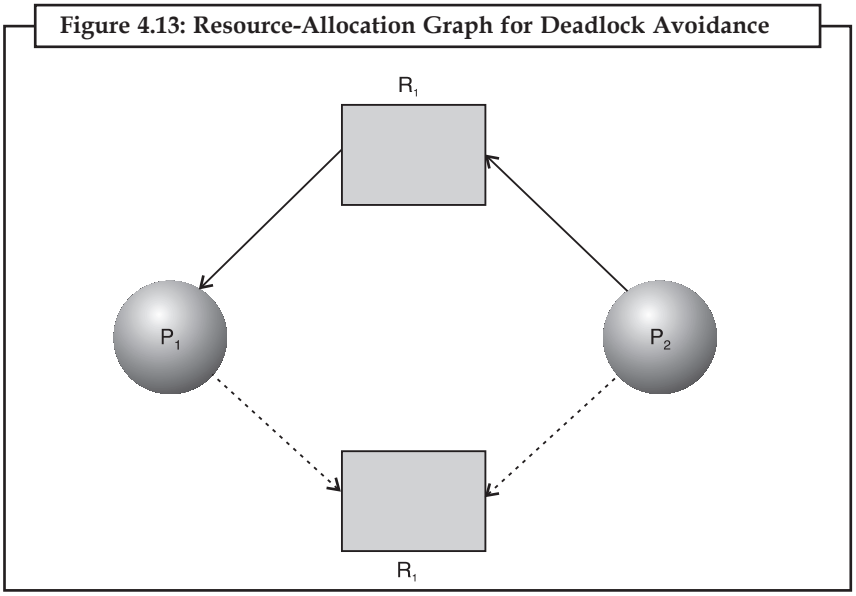
At time t0, the system is in a safe state. The sequence <PI, PO, P2> satisfies the safety condition, since process PI can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process Po can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P2 could get all its tape drives and return them (the system will then have all 12 tape drives available). A system may go from a safe state to an unsafe state. Suppose that, at time t1, process Pp requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process PI can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process Po is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process Po must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P2 for 1 more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock. Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state. In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock-avoidance algorithm.

4.6.12 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge Pi -+ Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi -+ Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj -+ Pi is reconverted to a claim edge Pi -+ Xi. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge Pi + Rj to be added to the graph only if all the edges associated with process Pi are claim edges. Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi -+ Rj to an assignment edge Rj + Pi does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system.

Notes

Figure 4.13: Resource-Allocation Graph for Deadlock Avoidance



If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for its requests to be satisfied. To illustrate this algorithm, we consider the resource-allocation graph of Figure 4.13. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If PI requests R2, and P2 requests R1, then a deadlock will occur.

4.6.13 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and **rn** be the number of resource types. We need the following data structures: Available: **a** vector of length **rn** indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

Max: An n x **rn** matrix defines the maximum demand of each process. If Max[i,j] = k, then process Pi may request at most k instances of resource type.

Ri . **Allocation:** An n x **rn** matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process **Pi** is currently allocated k instances of resource type Rj. Need: An n x **rn** matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process **Pi** may need k more instances of resource type Ri to complete its task. Note

Notes

that Need[i,j] = Max[i,j] - Allocafion[i,j]. These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and Y be vectors of length n. We say that X 5 Y if and only if X[i] 5 Y[i] for all i = 1, 2, ..., n. For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y 5 X. Y < X if Y IX and Y$X. We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocationi and Needi, respectively. The vector Allocation; specifies the resources currently allocated to process Pi; the vector Need, specifies the additional resources that process Pi may still request to complete its task. Safety Algorithm The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available and Finisk[i] :=false for i = 1,2, ..., n.
2. Find an i such that both a. Finisk[i] =false b. Needi 5 Work. If no such i exists, go to step 4.
3. Work := Work + Allocationi Finisk[i] := true go to step 2.
4. If Finish[i] = true for all i, then the system is in a safe state. This algorithm may require an order of rn x n2 operations to decide whether a state is safe.

4.6.14 Resource-Request Algorithm

Let Requesti be the request vector for process Pi. If Request;[j] = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Requesti 5 Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti 5 Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available := Available - Request,;

Allocationi := Allocation; + Request;;

Needi := Needi - Requesti;

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

Example: Consider a system with five processes Po through P4 and three resource types A, B, C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time To, the following snapshot of the system has been taken:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Notes

We claim that the system is currently in a safe state. Indeed, the sequence <PI, P3/ P4, P2/ Po> satisfies the safety criteria. Suppose now that process PI requests one additional instance of resource type A and two instances of resource type C, so Requestl = (1,0,2). To decide whether this request can be immediately granted, we first check that Requestl 5 Available (that is, (1,0,2) 5 (3,3,2)), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation Need			Available					
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <PI, P3, P4, POI P2> satisfies our safety requirement. Hence, we can immediately grant the request of process PI. You should be able to see, however, that when the system is in ths state, a request for (3,3,0) by **P4** cannot be granted, since the resources are not available. A request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

4.6.15 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide: An algorithm that examines the state of the system to determine whether a deadlock has occurred. An algorithm to recover from the deadlock. In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, let us note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

4.6.16 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges. More precisely, an edge from **Pi** to Pi in a wait-for graph implies that process Pi is waiting for process Pi to release a resource that Pi needs. An edge **Pi** -t Pi exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges **Pi** -+ Rq and Rq + Pj for some resource Rq. For example, we present a resource-allocation graph and the corresponding wait-for graph.

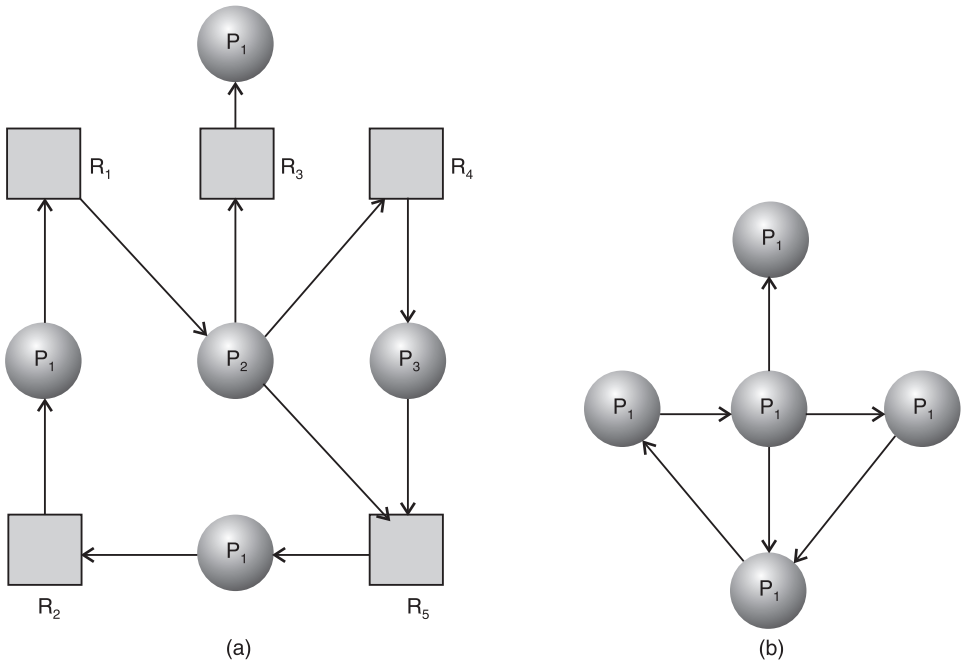
Notes



A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

Figure 4.14: (*a*) Resource-Allocation Graph and (*b*) Corresponding Wait-for Graph



4.6.17 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm that we describe next is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

Available: A vector of length rn indicates the number of available resources of each type.

Allocation: An n x rn matrix defines the number of resources of each type currently allocated to each process.

Request: An n x rn matrix indicates the current request of each process. If Request[i,j] = k, then process Pi is requesting k more instances of resource type Ri. The 5 relation between two vectors is defined. To simplify notation, we shall again treat the rows in the matrices Allocation and Request as vectors, and shall refer to them as Allocationi and Requesti, respectively. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available. For

i = 1, 2, ..., n, if Allocation $0, then Finish[i] :=false;

Notes

otherwise, Finish[i] := true.

2. Find an index i such that both

a. Finish[i] =false.

b. Requesti 5 Work.

If no such i exists, go to step 4.

3. Work := Work + Allocationi

Finish[i] := true go to step 2.

4. If Finish[i] = false, for some i, 1 _< i 5 n, then the system is in a deadlock state. Moreover, if Finish[i] =false, then process Pi is deadlocked. This algorithm requires an order of m x n2 operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process Pi (in step 3) as soon as we determine that Requesti _< Work (in step 2b). We know that Pi is currently not involved in a deadlock (since Requesti < Work). Thus, we take an optimistic attitude, and assume that Pi will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time that the deadlock-detection algorithm is invoked. To illustrate this algorithm, we consider a system with five processes Po through P4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation			**Need**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P_0	0	1	0	0	0	2	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <Po, P2, P3, PI, P4> will result in Finish[i] = true for all i. Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	Request				
	A	**B**	**C**		
P_0	0	1	0		
P_1	2	0	2		
P_2	0	0	1		
P_3	1	0	0		
P_4	0	0	2		

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes PI, P2, P3, and Pq.

Notes

4.6.18 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection-algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, we could invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may cause many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable process. Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent intervals—for example, once per hour, or whenever CPU utilization drops below 40 per cent. (A deadlock eventually cripples system throughput and will cause CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. We would generally not be able to tell which of the many deadlocked processes caused the deadlock.

4.6.19 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

4.6.20 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

4.6.21 Abort All Deadlocked Processes

This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

4.6.22 Abort One Process at a Time Until the Deadlock Cycle is Eliminated

This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the

Notes

deadlock. This determination is a policy decision, similar to CPU-scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine which process is chosen, including:

1. What the priority of the process is?
2. How long the process has computed, and how much longer the process will compute before completing its designated task?
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)?
4. How many more resources the process needs in order to complete?
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch?

4.6.23 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a Victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of pre-1 emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state. Since, in general, it is difficult to determine what a safe state is, them simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to rollback the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.
3. **Starvation:** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.



Case Study

Write Code that Acquires More Than One Lock

```
Lock *l1, *l2;
void p() {
   l1->Acquire();
   l2->Acquire();
   code that manipulates data that l1 and l2 protect
   l2->Release();
```

Contd...

Notes

```
   l1->Release();
}
void q() {
   l2->Acquire();
   l1->Acquire();
   code that manipulates data that l1 and l2 protect
   l1->Release();
   l2->Release();
}
```

If `p` and `q` execute concurrently, consider what may happen. First, `p` acquires `l1` and `q` acquires `l2`. Then, `p` waits to acquire `l2` and `q` waits to acquire `l1`. How long will they wait? Forever. This case is called deadlock.

Questions:

1. What are conditions for deadlock?
2. How can `p` and `q` avoid deadlock? Order the locks, and always acquire the locks in that order. Eliminates the circular wait condition.



Lab Exercise

1. Consider four processes arrived at different times and needed CPU time differ from process to other according to the next table:

Process	Arrived Time	CPU time (minute)
P_1	10:00	4 m
P_2	10:10	3 m
P_3	10:15	2 m
P_4	10:25	2 m

2. Calculate the termination time for each process, If the probability of I/O=0.7 in each of the following cases:

   (*a*) Mono-programming.

   (*b*) Multi-programming.

Self Assessment

Multiple choice questions:

5. ..... is a high level abstraction over Semaphore.

   (*a*) Shared memory (*b*) Message passing

   (*c*) Monitor (*d*) Mutual exclusion

6. The process related to process control, file management, device management, information about system and communication that is requested by any higher level language can be performed by ..... .

   (*a*) Editors (*b*) Compilers

   (*c*) System Call (*d*) Caching

Notes

7. Which is not the state of the process?
   (*a*) Blocked (*b*) Running
   (*c*) Ready (*d*) Privileged

Fill in the blanks:

8. ..... is a lightweight unit of program execution.
9. An example of a discretionary acess control is a ..... .
10. A system is in a safe state only if there exists a ..... .

4.7 Summary

- If a data item is shared by a number threads, race conditions could occur if the shared item is not protected properly.
- The first restriction is only the owner can release the lock. The second restriction is recursive lock acquisition is not allowed.
- Each process has a code segment, called critical section, in which the shared data is accessed.
- For many years, microprocessors have included on-chip memory management units (MMU).
- As applications become more complex, performing more functions for a software-hungry world, the number of bugs in fielded systems will continue to rise.
- A typical embedded system has several types of interrupts resulting from the use of various kinds of devices.
- Priority inversion has long been the bane of system designers attempting to perform rate monotonic analysis, since RMA depends on higher priority threads running before lower priority threads.
- A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphoiinitislize'.
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

4.8 Keywords

Memory Management Unit (MMU): Memory management units that enable individual threads of software to run in hardware-protected address spaces.

Real Time Operating System (RTOS): A Real-Time Operating System is a computing environment that reacts to input within a specific time period.

Rate Monotonic Analysis (RMA): Rate monotonic analysis is frequently used by system designers to analyze and predict the timing behavior of systems.

Deadlock: A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

Resource Allocation Graph: Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

Deadlock Avoidance: A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

Claim Edge: Resource-allocation graph can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge.

Notes

4.9 Review Questions

1. State a simple rule for avoiding deadlocks in this system.
2. Consider the deadlock situation that could occur in the dining-philosophers Problem when the philosophers obtain the chopsticks one at a time. Discuss How the four necessary conditions for deadlock indeed hold in this Setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.
3. What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
4. The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of n - 1 turns was presented by Eisenberg and McGuire. The processes share the following variables enum pstate {idle, want-in, in-cs);

   pstate flagCnl ; i n t turn;

```
do {
flag[i] = true;
while (flag[j]) {
         if (turn == j) {
              flag[i] = false;
              while (turn == j);
              flag[i] = true;
         }
}
```

   critical section

```
turn = j;

flag[i] = false;
```

   remainder section

   while (1);

The structure of process *Pi* in Dekker's algorithm.

All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and n-1). The structure of process *Pi* is shown below.

```
do {

while(1) {
        flag[i] = want-in;
        j = turn;
        while (j ! = i) {
           if (flag[j] != idle)
              j = turn;
           else
              j = (j+1) % n;
}
```

Notes

```
flag[i] = in-cs;
j = 0;
while ((j < n) && (j == i || flag[j] != in_cs))
   j++;
if ((j >= n) && (turn ++ i || flag[turn) == idle)) break;
}
turn = i;
```

critical section

```
j — (turn + 1) % n:
while (flag [j] == idle)
        j = (j+1) % n;
turn = j;
flag[i] = idle;
```

remainder section

}while (1)

The structure of process *Pi* in Eisenberg and McGuire's algorithm.

Prove that the algorithm satisfies all three requirements for the critical section problem.

5. Demonstrate that monitors, conditional critical regions, and semaphore are all equivalent, insofar as the same types of synchronization problems can be implemented with them.
6. Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
7. List three examples of deadlocks that are not related to a computer-system environment.
8. Is it possible to have a deadlock involving only one process? Explain your answer.
9. Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.
10. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.
11. Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.
12. Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0012	1520	
P_1	1000	1750	
P_2	1354	2356	
P_3	0632	0652	
P_4	0014	0656	

Notes

13. Discuss about Banker's Algorithm with the example.
14. What is deadlock condition? Explain with the example.
15. Write the definition of Semaphores.

Answers to Self Assessment

1. (*d*) 2. (*b*) 3. (*a*) 4. (*b*)
5. (*c*) 6. (*c*) 7. (*d*) 8. Thread
9. Unix file 10. Safe sequence

4.10 Further Readings



Books

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.



Online link

wiley.com/coolege.silberschatz

Unit 5: Memory Management

Notes

CONTENTS

Objectives
Introduction
5.1 Address Binding
5.2 Logical versus Physical-Address Space
5.3 Swapping
5.4 Contiguous Memory Allocation
5.5 Paging
5.5.1 Basic Method
5.5.2 Hardware Support
5.5.3 Protection
5.5.4 Hierarchical Paging
5.5.5 Shared Pages
5.6 Segmentation
5.6.1 Segmentation with Paging
5.7 Virtual Memory
5.7.1 Paging and Swapping
5.7.2 Demand Paging — Lazy Evaluation
5.7.3 Swapping and Paging Algorithms
5.7.4 FIFO — First In First Out
5.7.5 Second Chance
5.7.6 LRU — Least Recently Used
5.8 Thrashing
5.9 Demand Paging
5.9.1 Basic Concepts
5.9.2 Performance of Demand Paging
5.10 Overview of Page Replacement
5.10.1 Basic Scheme
5.10.2 FIFO Page Replacement
5.10.3 Optimal Page Replacement
5.10.4 Page-replacement Algorithm
5.11 LRU Page Replacement
5.11.1 Counters
5.11.2 Stack
5.11.3 LRU Approximation Page Replacement
5.11.4 Additional-Reference-Bits Algorithm
5.11.5 Second-chance Algorithm
5.11.6 Enhanced Second-chance Algorithm
5.11.7 Counting-based Page Replacement
5.11.8 Page-buffering Algorithm

Notes

5.12 Thrashing
5.12.1 Cause of Thrashing
5.12.2 Working-set Model
5.12.3 Page-fault Frequency
5.13 Summary
5.14 Keywords
5.15 Review Questions
5.16 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain memory management
- Understand swapping
- Discuss contiguous-memory allocation
- Explain paging
- Discuss thrashing
- Explain overview of page replacement
- Understand LRU page replacement

Introduction

In this unit, we discuss various ways to manage memory. The memory management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, many algorithms require hardware support, although recent designs have closely integrated the hardware and operating system.

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

5.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk, that is waiting to be brought into memory for execution forms the **input queue.**

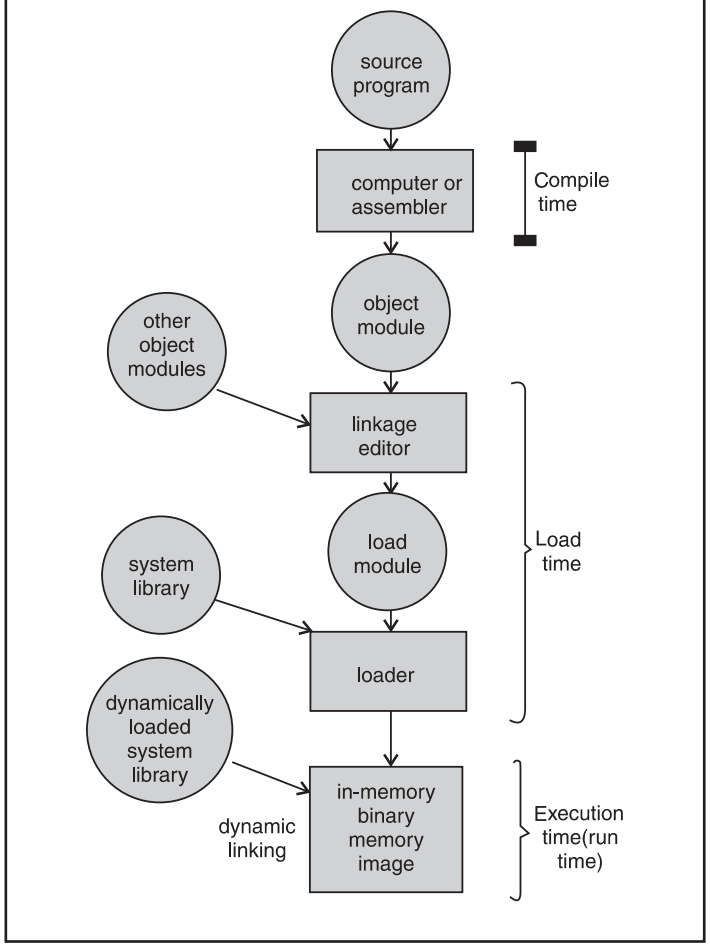
The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Notes

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as count). A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

1. **Compile Time:** If you know at compile time where the process will reside in memory, then,
2. **Absolute Code** can be generated. For example, if you know a priori that a user process resides starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS. COM-format programs are absolute code bound at compile time.
3. **Load Time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code.** In this case, final binding is delayed until load

Figure 5.1: Multistep Processing of a user Program



Notes

time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

4. **Execution Time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method. A major portion of this unit is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

5.2 Logical versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory is commonly referred to as a **physical address.**

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address.** We use logical address and virtual address interchangeably in this text. The set of all logical addresses generated by a program is a **logical-address space;** the set of all physical addresses corresponding to these logical addresses is a **physical-address space.** Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

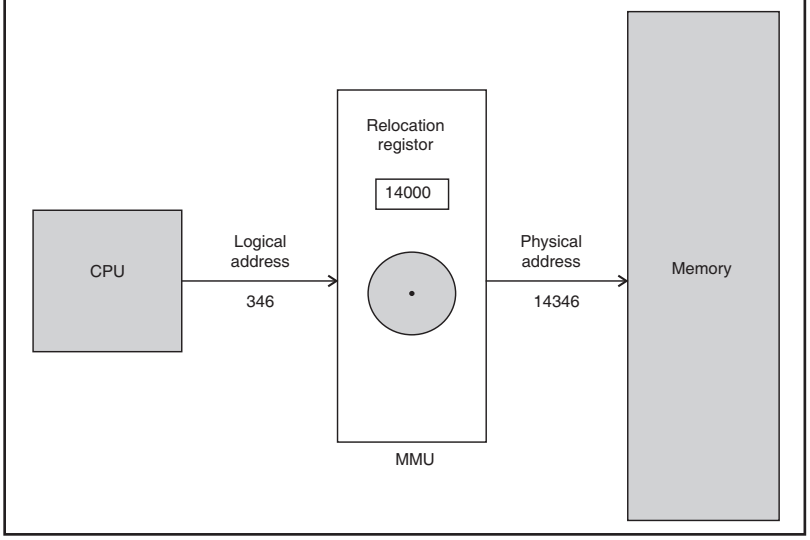
This method requires hardware support slightly different from the hardware configuration. The base register is now called a **relocation register.** The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses-all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in section given above. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a bas value R). The user generates only logical addresses and thinks that the process runs in locations 0 to max. The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used. The concept of a logical-address space that is bound to a separate physical address space is central to proper memory management.

Notes

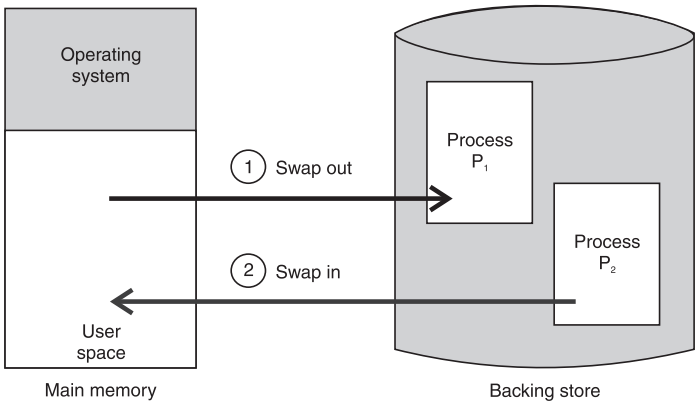
Figure 5.2: Dynamic Relocation using a Relocation Register



5.3 Swapping

A process needs to be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure 5.3). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

Figure 5.3: Swapping of Two Processes using a Disk as a Backing Store



Notes

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be moved to different locations. If execution-time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time. Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let us assume that the user process is of size 1MB and the backing store is a standard hard disk with a transfer rate of 5 MB per second. The actual transfer of the 1 MB process to or from memory takes 1000 KB/5000 KB per second = 1 /5 second = 200 milliseconds.

Assuming that no head seeks are necessary and an average latency of 8 milliseconds, the swap time takes 208 milliseconds. Since we must both swap out and swap in, the total swap time is then about 416 milliseconds. For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.416 seconds. Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. If we have a computer system with 128 MB of main memory and a resident operating system taking 5 MB, the maximum size of the user process is 123 MB. However, many user processes may be much smaller than this size-say, 1 MB. A 1 MB process could be swapped out in 208 milliseconds, compared to the 24.6 seconds for swapping 123 MB. Therefore, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using. Then, we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (request memory and release memory) to inform the operating system of its changing memory needs.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up its memory. However, if the 1/0 is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the 1/0 operation was queued because the device was busy. Then, if we were to swap out process PI and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2. The two main solutions to this problem are never to swap a process with pending I/O, or to execute 1/0 operations only into operating-system buffers. Transfers between operating-system buffers and process memory then occur only when the process is swapped in. The assumption that swapping requires few, if any, head seeks needs further explanation. We postpone discussing this issue until where secondary-storage structure is covered. Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible. Currently, standard swapping is used in few systems. It requires too much swapping time and provides too little execution time to be a reasonable

Notes

memory-management solution. Modified versions of swapping, however, are found on many systems. A modification of swapping is used in many versions of UNIX. Swapping was normally disabled, but would start if many processes were running and were using a threshold amount of memory. Swapping would again be halted if the load on the system were reduced. Early PCs lacked sophisticated hardware (or operating systems that take advantage of the hardware) to implement more advanced memory management methods, but they were used to run multiple large processes by a modified version of swapping. A prime example is the Microsoft Windows 3.1 operating system, which supports concurrent execution of processes in memory. If a new process is loaded and there is insufficient main memory, an old process is swapped to disk. This operating system, however, does not provide full swapping, because the user, rather than the scheduler, decides when it is time to preempt one process for another. Any swapped-out process remains swapped out (and not executing) until the user selects that process to run. Follow-on Microsoft operating systems, such as Windows NT, take advantage of advanced MMU features now found even on PCs.


Did u know?

Swapping is a simple memory/process management technique used by the operating system(os) to increase the utilization of the processor by moving some blocked process from the main memory to the secondary memory(hard disk); thus forming a queue of temporarily suspended process and the execution continues with the newly arrived process. After performing the swapping process,the operating system has two options in selecting a process for execution.

5.4 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible. This section will explain one common method, contiguous memory allocation. The memory is usually divided into two partitions—one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we shall discuss only the situation where the operating system resides in low memory. The development of the other situation is similar. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

5.5 Paging

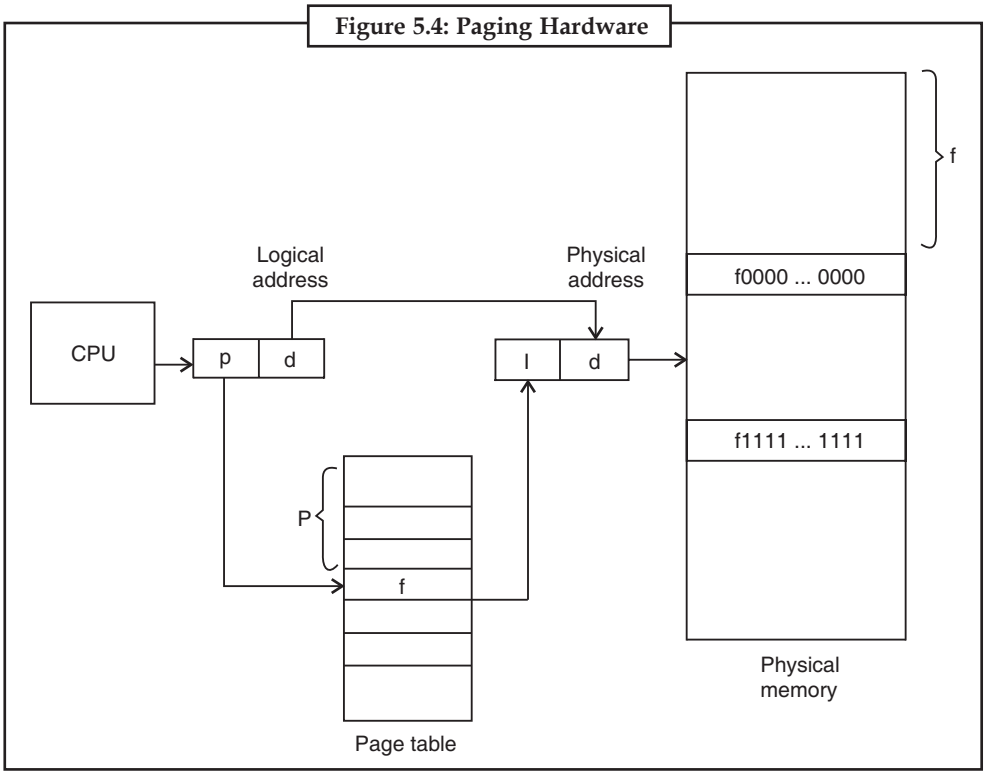
Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection

Notes

with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in most operating systems.

Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Figure 5.4: Paging Hardware



5.5.1 Basic Method

Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames. Every address generated by the CPU is divided into two parts: a page number **(p)** and a page offset **(d).** The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical-address space is 2m, and a page size is 2n addressing units (bytes or words), then the high-order **rn—**n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

Notes

Figure 5.5: Paging Model of Logical and Physical Memory

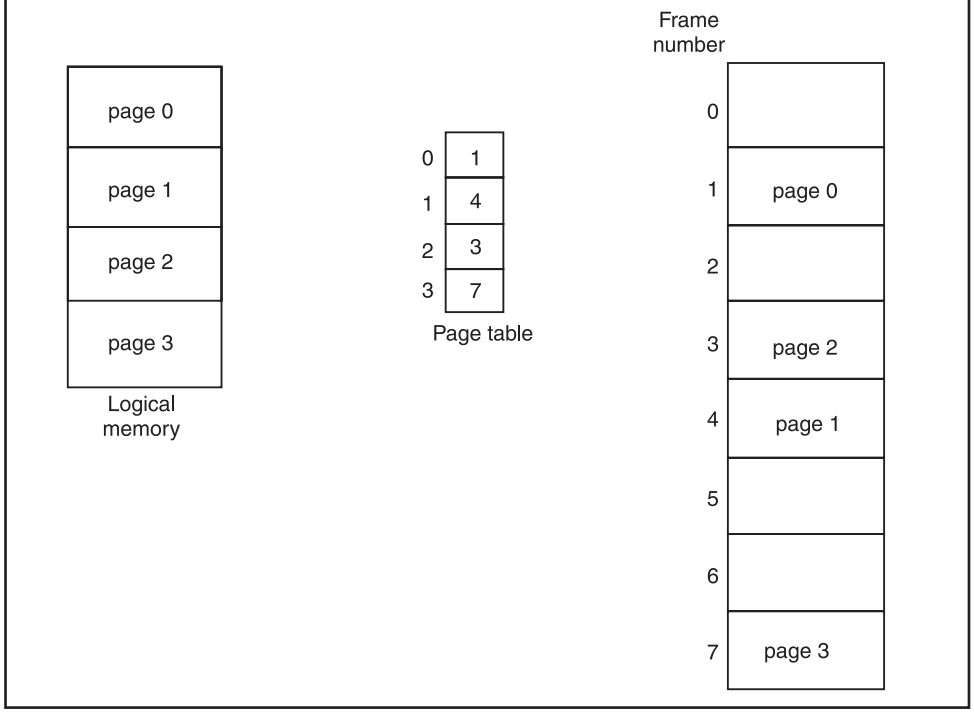
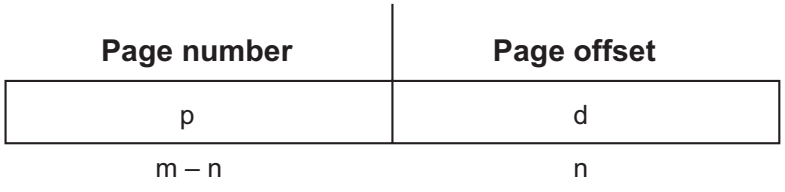


Figure 5.6: Where p is an index into the page table and d is the displacement within the page

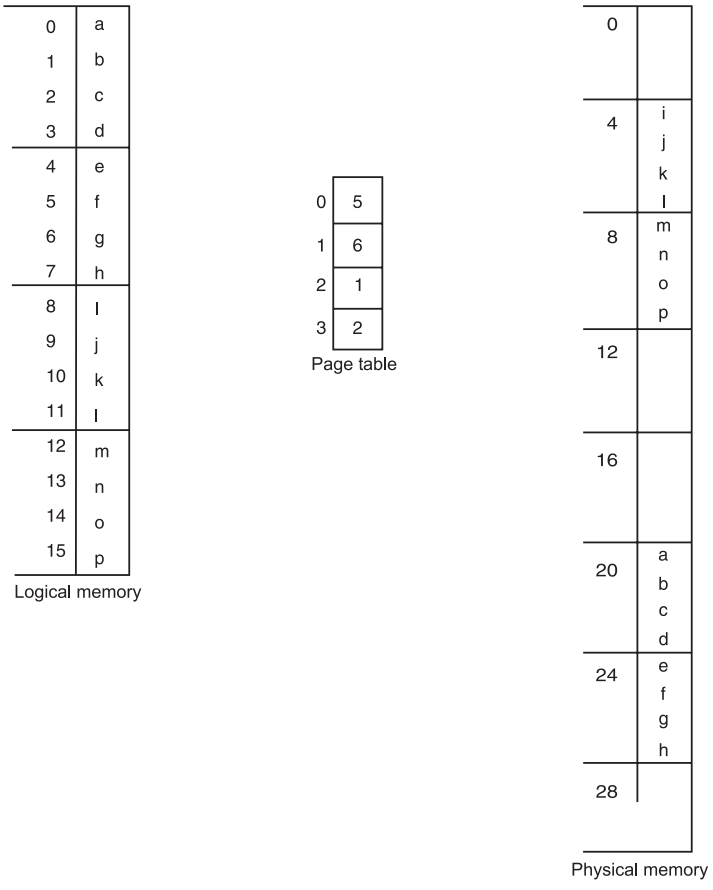


As a concrete (although minuscule) example, consider the memory in Figure 5.7. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 × 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 × 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 × 4) + 0). Logical address 13 maps to physical address 9. You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to fall on page boundaries, the last frame allocated may not be completely full. For example, if pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages

Notes

plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2048 — 1086 = 962 bytes. In the worst case, a process would need n pages plus one byte. It would be allocated n + 1 frames, resulting in an internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today pages typically are between KB and 8 KB, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses 8 KB and 4 MB page sizes, depending on the data stored by the pages. Researchers are now developing variable on-the-fly page-size support. Each page-table entry is usually 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If a frame is 4 KB, then a system with 4-byte entries can address 236 bytes (or 64 GB) of physical memory. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.

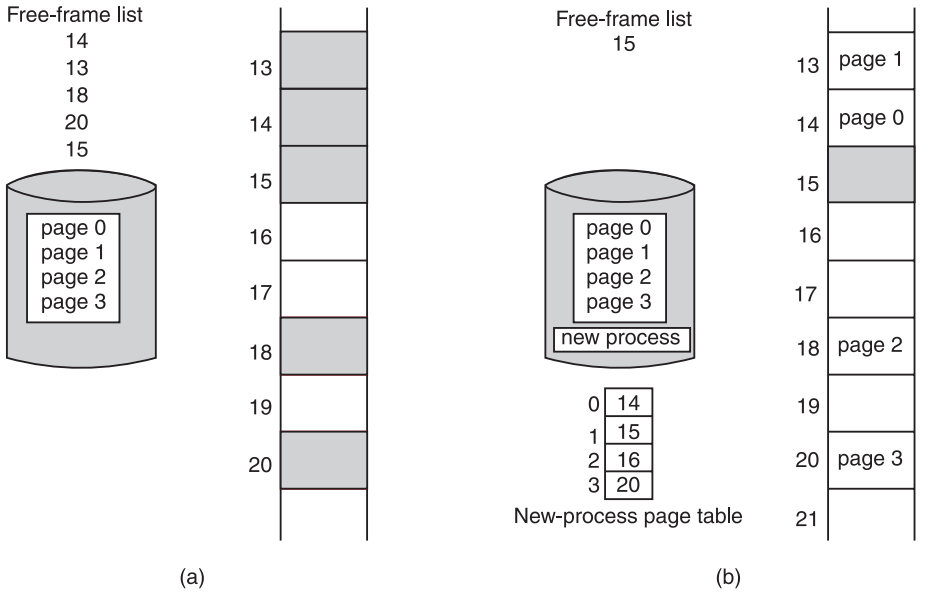
Figure 5.7: Paging Examples for a 32-byte Memory with 4-byte Pages



Notes

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program.

Figure 5.8: Free Frames: (*a*) Before Allocation and (*b*) After Allocation



In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns. Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes. In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

Notes



Did u know?

Paging used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

5.5.2 Hardware Support

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers. The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a page-table base register **(PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for **i.** This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, **two** memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

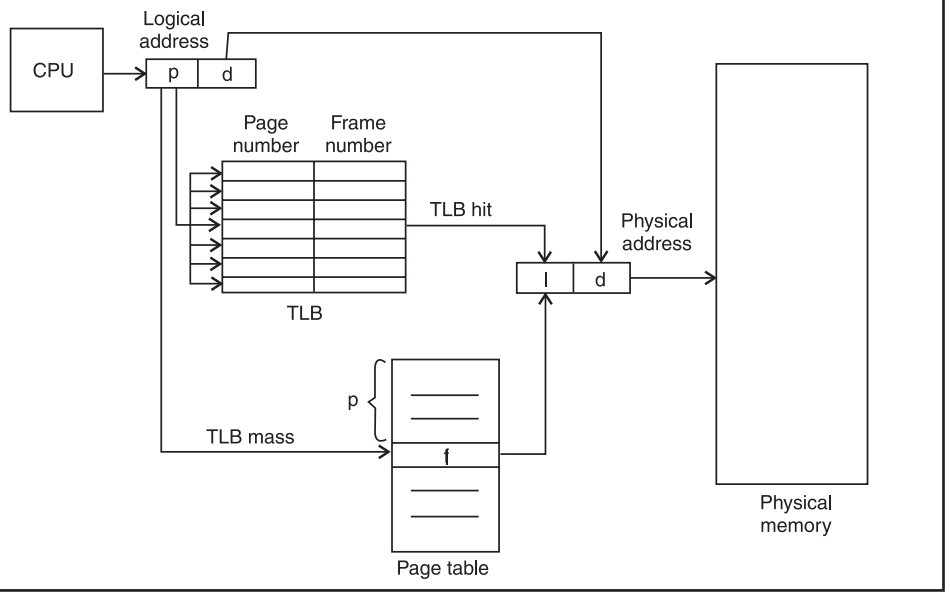
The standard solution to this problem is to use a special, small, fast lookup hardware cache, called translation look-aside buffer **(TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024. The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 per cent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (known as a **TLB** miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

Notes

Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are often wired down. Some TLBs store address-space identifiers **(ASIDs)** in each entry of the TLB. An ASID uniquely identifies each process and is used to provide address space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, they are treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

Figure 5.9: Paging Hardware with TLB



If the TLB does not support separate ASIDs, every time a new page table is selected (for instance, each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process. The percentage of times that a particular page number is found in the TLB is called the **hit ratio.** An 80-per cent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the TLB.

If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the **effective memory-access time,** we must weigh each case by its probability: effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds. In this example, we suffer a 40-per cent slowdown in memory access time (from 100 to 140 nanoseconds). For a 98-per cent hit ratio, we have effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds. This increased hit rate produces only a 22-per cent slowdown in access time.

Task How to uses paging in the RAM and CPU?

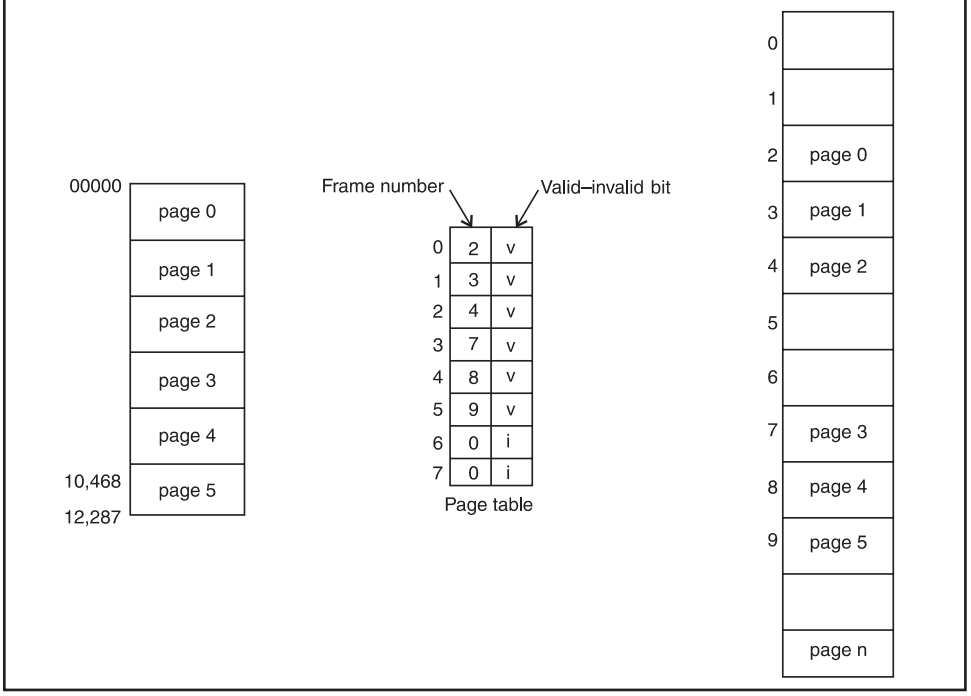
Notes

5.5.3 Protection

Memory protection in a paged environment is accomplished by protection bits 11 that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation). We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses; illegal attempts will be trapped to the operating system. One more bit is generally attached to each entry in the page table: a **validin valid** bit. When this bit is set to "valid," this value indicates that the associated page is in the process logical-address space, and is thus a legal (or valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process logical-address space. Illegal addresses are trapped by using the validin valid bit. The operating system sets this bit for each page to allow or disallow accesses to that page. For example, in a system with a 14-bit address space (0 to 16383), we may have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, however, finds that the validin valid bit is set to invalid, and the computer will trap to the operating system. Because the program extends to only address 10468, any reference beyond that address is illegal. This problem is a result of the 2 KB page size and reflects the internal fragmentation of paging. Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries with entries for every page in the address range. Most of this table would be unused, but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR),** to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

Structure of the Page Table: In this section we explore some of the most common techniques for structuring the page table.

Figure 5.10: Valid (v) or Invalid (i) Bit in a Page Table

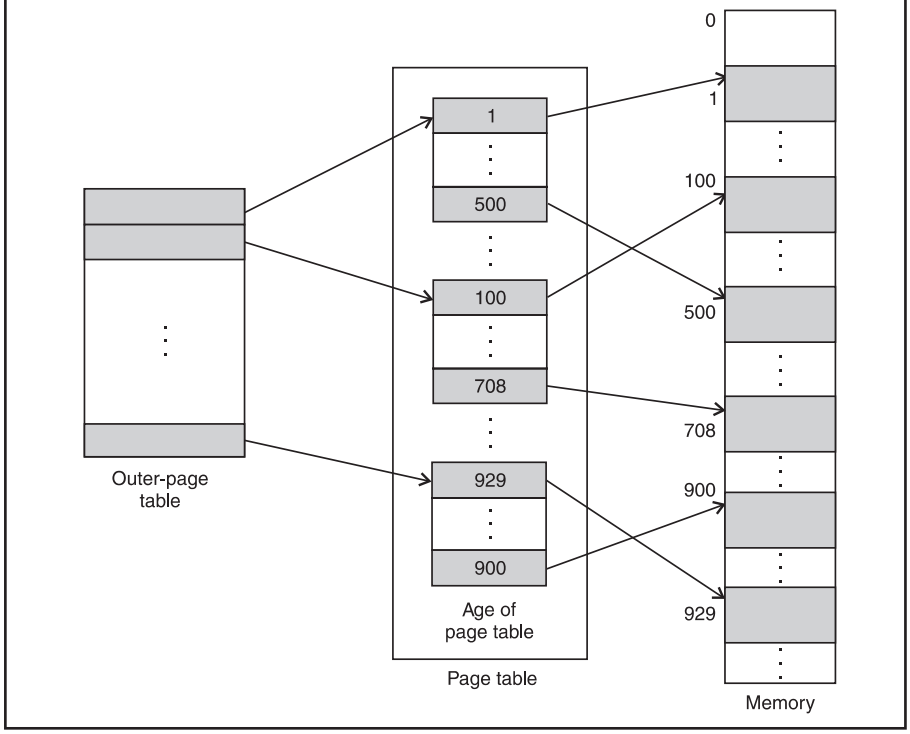


Notes

5.5.4 Hierarchical Paging

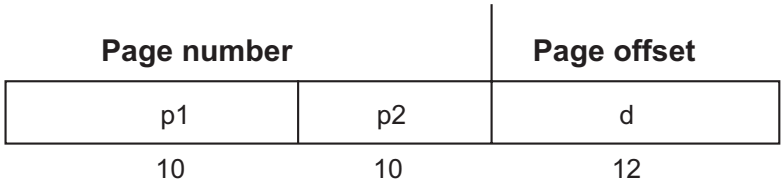
Most modern computer systems support a large logical-address space (Z3 to 2@). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical-address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical-address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. There are several ways to accomplish this division.

Figure 5.11: A Two-level Page-table Scheme



Remember our example to our 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows—where p1 is an index into the outer page table and p2 is the

Figure 5.12: Logical Address



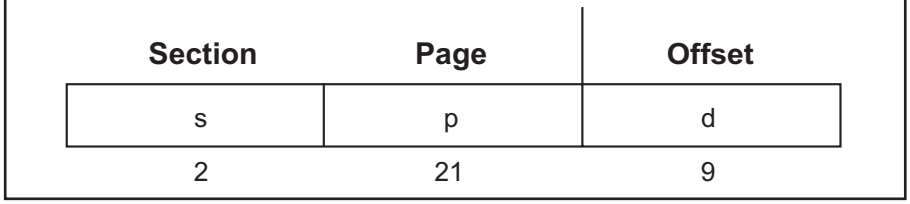
displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure 5.14. Because address translation works from the outer page table inwards, this scheme is also known as a **forward-mapped page table.** The Pentium-I1 uses this architecture. The **VAX** architecture also supports a variation of two-level paging. The **VAX** is

Notes

a 32-bit machine with page size of 512 bytes. The logical-address space of a process is divided into four equal sections, each of which consists of 230 bytes.

Each section represents a different part of the logical-address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. An address on the **VAX** architecture is as follows:

Figure 5.13: VAX Architecture



where s designates the section number, p is an index into the page table, and d is the displacement within the page. The size of a one-level page table for a **VAX** process using one section still is 2″ bits * 4 bytes per entry = 8 MB. So that main-memory use is reduced even further, the **VAX** pages the user-process page tables. For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (212). In this case, the page table will consist of up to 252 entries. If we use a two-level paging scheme, then the inner page tables could conveniently be one page long, or contain 21 4-byte entries. The addresses would look like:

Figure 5.14: Address Translation for a Two-level 32-bit Paging Architecture

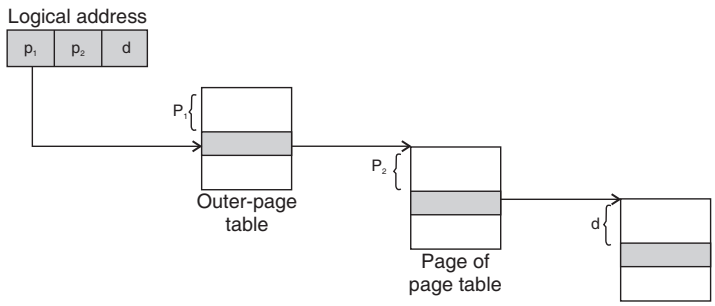
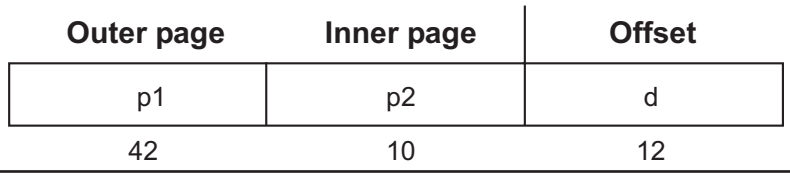


Figure 5.15: 32-bit Paging



The outer page table will consist of 242 entries, or 244 bytes. The obvious method to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency. We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (21 entries, or 2 bytes); a 64-bit address space is still daunting:

Notes

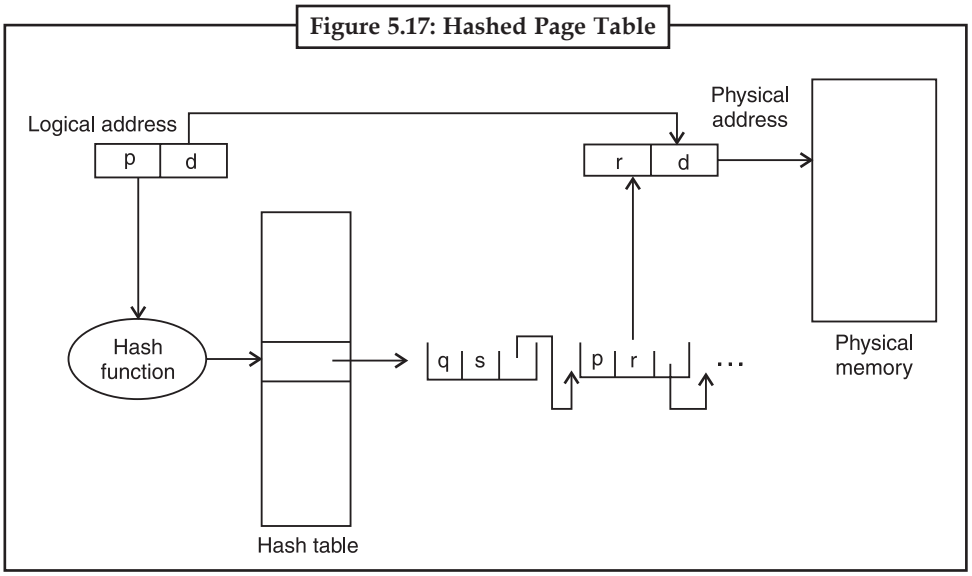
Figure 5.16: 64-bit Address

2nd outer page	outer page	inner page	offset
p1	p2	p3	d
32	10	10	12

The outer page table is still **234** bytes large. The next step would be a four-level paging scheme, where the second level outer page table itself is also paged. The SPARC architecture (with 32-bit addressing) supports a three-level paging scheme, whereas the 32-bit Motorola 68030 architecture supports a four-level paging scheme.

However, for 64-bit architectures, hierarchical page tables are generally considered inappropriate. For example, the 64-bit UltraSPARC would require seven levels of paging-a prohibitive number of memory accesses to translate each logical address. 42 10 12 inner page P2 2nd outer page PI **Hashed Page Tables.** A common approach for handling address spaces larger than 32 bits is to use a **hashed page table,** with the hash value being the virtual-page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (a) the virtual page number, (b) the value of he mapped page frame, and (c) a pointer to the next element in the linked list. The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. A variation to this scheme that is favorable for 64-bit address spaces has been proposed. **Clustered page tables** are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables

Figure 5.17: Hashed Page Table



are particularly useful for **sparse** address spaces where memory references are noncontiguous and scattered throughout the address space.

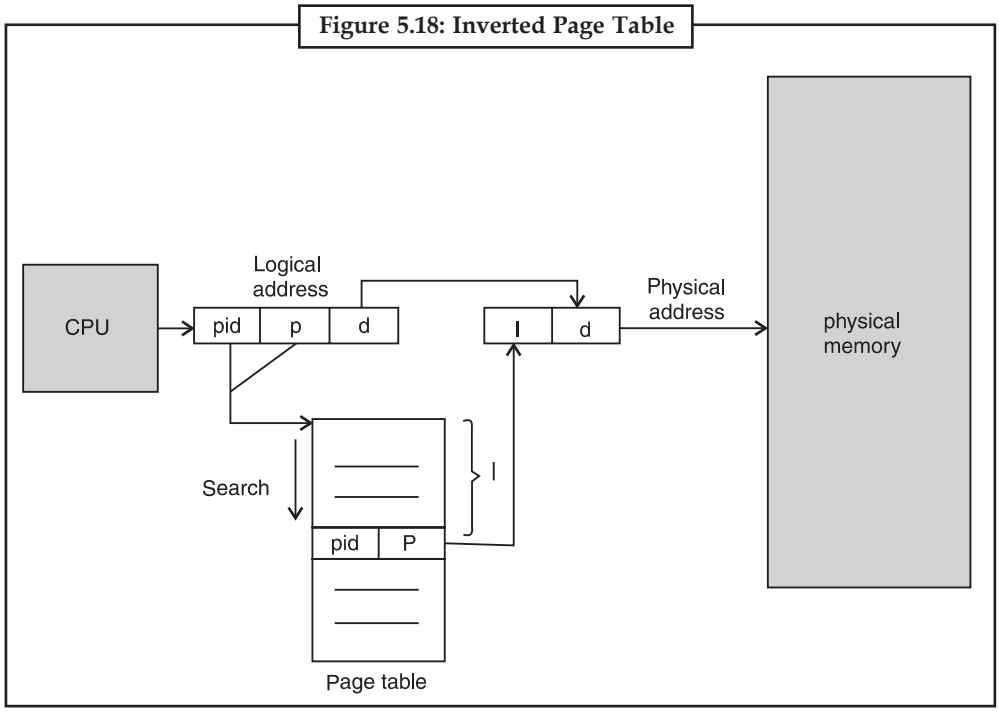
Inverted Page Table: Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless

Notes

of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system t of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.

To solve this problem, we can use an **inverted page table.** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. which depicts a standard page table in operation. Because only one page table is in the system yet there are usually several different address spaces mapping physical memory, inverted page tables often require an address-space identifier stored in each entry of the page table.

Figure 5.18: Inverted Page Table



Storing the address-space identifier ensures the mapping of a logical page for a particular process to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC. To illustrate this method, we describe a simplified version of the implementation of the inverted page table used in the IBM RT. Each virtual address in the system consists of a triple <process-id, page-number, offset>. Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found-say, at entry i-then the physical address **<i,** offset> is generated. If no match is found, then an illegal address access has been attempted.

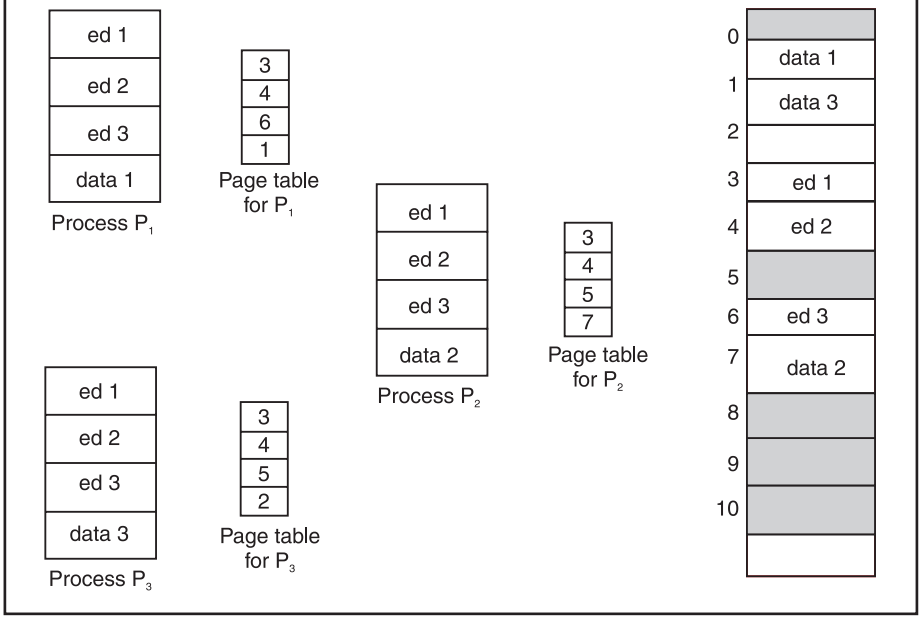
Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by a physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual-memory reference requires at least two real-memory reads one for the hash-table entry and one for the page table. To improve performance, recall that the TLB is searched first, before the hash table is consulted.

Notes

5.5.5 Shared Pages

Another advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we would need 8,000 KB to support the 40 users. If the code is **reentrant code**, Here we see a three-page editor-each page of size 50 KB; the large page size is used to simplify the figure-being shared among three processes. Each process has its own data page. Reentrant code (or **pure code)** is non-self-modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process execution. The data for two different processes will, of course, vary for each process.

Figure 5.19: Sharing of Code in a Paging Environment



Only one copy of the editor needs to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB, instead of 8,000 KB-a significant savings. Other heavily used programs can also be shared-compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have two (or more) shared virtual addresses. Organizing memory according to pages provides numerous other benefits in addition to allowing several processes to share the same physical pages.

Notes



The free page queue is a list of page frames that are available for assignment after a page fault. Some operating systems support page reclamation; if a page fault occurs for a page that had been stolen and the page frame was never reassigned, then the operating system avoids the necessity of reading the page back in by assigning the unmodified page frame.

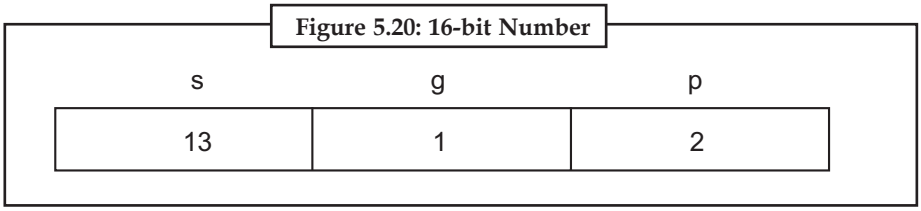
5.6 Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory.

5.6.1 Segmentation with Paging

Both paging and segmentation have advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat-address space, whereas the Intel 80 × 86 and Pentium family are based on segmentation. Both are merging memory models toward a mixture of paging and segmentation. We can combine these two methods to improve on each. This combination is best illustrated by the architecture of the Intel 386. The IBM OS/2 32-bit version is an operating system running on top of the Intel 386 (and later) architecture. The Intel 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes. The page size is 4 KB. We shall not give a complete description of the memory-management structure of the 386 in this text. Rather, we shall present the major ideas. The logical-address space of a process is divided into two partitions. The first partition consists of up to 8 KB segments that are private to that process. The second partition consists of up to 8 KB segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table (LDT),** information about the second partition is kept in the **global descriptor table (GDT).** Each entry in the LDT and GDT consists of 8 bytes, with detailed information about a particular segment including the base location and length of that segment. The logical address is a pair (selector, offset), where the selector is a 16-bit number:

Figure 5.20: 16-bit Number



in which s designates the segment number, g indicates whether the segment is in the GDT or LDT, and **p** deals with protection. The offset is a 32-bit number specifying the location of the byte (or word) within the segment in question. The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It has six 8-byte micro program registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the 386 avoid having to read the descriptor from memory for every memory reference. The physical address on the 386 is 32 bits long and is formed as follows.

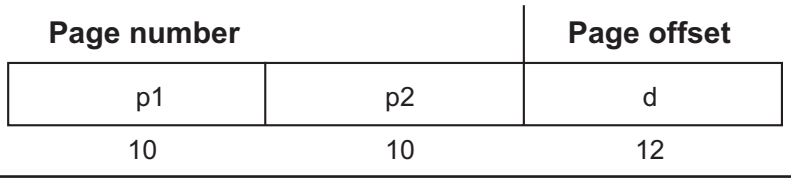
The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question are used to generate a **linear address.** First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated,

Notes

resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address. As pointed out previously, each segment is paged, and each page is 4 KB.

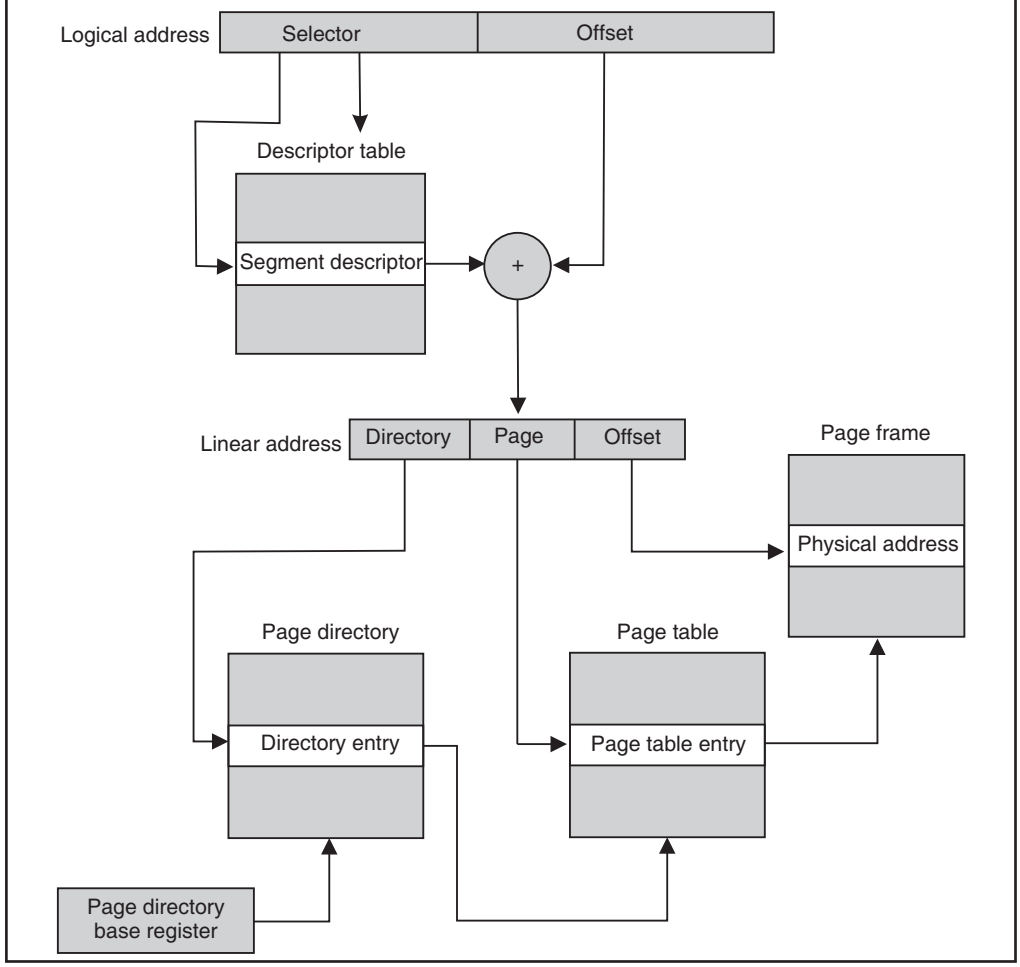
A page table may thus consist of up to 1 million entries. Because each entry consists of 4 bytes, each process may need up to 4 MB of physical-address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. The solution adopted in the 386 is to use a two-level paging scheme. The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows:

Figure 5.21: Logical Address



To improve the efficiency of physical-memory use, Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which

Figure 5.22: Intel 80386 Address Translation



Notes

the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.


Caution

All CPU address is differed one another.


Did u know?

Segmentation is one approach to memory management and protection in the operating system. It has been superseded by paging for most purposes, but much of the terminology of segmentation is still used, "segmentation fault" being an example. Some operating systems still have segmentation at some logical level although paging is used as the main memory management policy.

Self Assessment

Multiple choice questions:

1. Primary memory stores ..... .
   (*a*) Data alone
   (*b*) Programs alone
   (*c*) Results alone
   (*d*) All of these
2. Memory is made up of ..... .
   (*a*) Set of wires
   (*b*) Set of circuits
   (*c*) Large number of cells
   (*d*) All of these
3. The principal of locality of reference justifies the use of
   (*a*) re-enterable (*b*) non-reusable
   (*c*) virtual memory (*d*) cache memory

5.7 Virtual Memory

5.7.1 Paging and Swapping

Virtual memory is a way of making the physical memory of a computer system effectively larger than it really is. Rather than using mirrors, the system does this by determining which parts of its memory are often sitting idle, and then makes a command decision to empty their contents onto a disk, thereby freeing up useful RAM.

As we noted earlier, it is quite seldom that every byte of every program is in use all of the time. More often programs are large and contain sections of code which are visited rarely if ever at all by the majority of users – so if they are not used, why keep them in RAM?

Notes

Virtual memory uses two methods to free up RAM when needed.

- **Swapping:** An entire process, including code segment and data segments is expunged from the system memory.
- **Paging:** Only single pages are swapped out.

Of course, the simplest way to clear a space in RAM is to terminate some processes, but virtual memory is more subtle than that. The idea is to free RAM only temporarily, with the intention of copying the data back again later. All of this should happen in such a way that the user of the system do not realize that it is happening.

Swapping and paging dump the system memory in special disk caches. Normally these disk areas are not part of the usual file system structure, since the overhead of maintaining a file system is inappropriate when only the system needs to use the disk. Instead, the system stores swap files in large contiguous blocks, sacrificing utilization of space for speed. Some systems also allow swapping to a special file in the normal filesystem, which has a reserved size.

In UNIX, there both methods are available. On BSD systems, normally a whole disk partition (see next section) is reserved for swapping and paging. (This is called the swap partition for historical reasons.) If this fails to provide enough space, under SunOS the system administrator can either add other partitions, or use the mkfile command to create a swap file on a normal in a part of the file system where there is sufficient space. In the system 5 based HPUX operating system, the normal swap area is invisible to the user. Additional swap space can simply be grabbed from some part of the filesystem, by the kernel, if the system goes short. Eventually this can lead to a paradoxical situation in which the user sees nothing on the disk, but the OS declares that the disk is full!

Early versions of UNIX used swapping exclusively when RAM was in short supply. Since BSD 4.3, all systems which have learned something from the BSD project use paging as their main method of virtual memory implementation.

5.7.2 Demand Paging — Lazy Evaluation

You might ask — if a program has a lot of pages which do not get used, what is the purpose of loading them in the first place and then swapping them out? One could simply make a rule that no page should be brought into memory until it were needed. Such a scheme is possible, but few systems allow a program to run if it cannot be loaded fully into memory on start-up. One argument against this extreme form of paging is that, it could be dangerous to start a program which was unable to complete because it was too large to run on the system, under the conditions of the moment. If it started to run and then crashed or exited, it could compromise important data. (The BSD UNIX system allocates sufficient space in its swap area to swap or page out each entire process as it begins. That way, none of them will ever run out of swap during execution.)

On the other hand, if a program can be loaded in, it is most likely safe — so if we then discover that large parts of the program are never used, we can page them out and never bother to page them in again.

This is an example of what is called lazy evaluation. A lazy pager never brings a page back into memory until is has to, i.e. until someone wants to use it. This can save a considerable amount of I/O time. Another name for this is demand paging, since it only occurs on demand from user processes.

It is now easy to see how the paging concept goes hand in hand with the logical memory concept: each time the system pages out a frame of physical memory, it sets a flag in the page table next to the logical page that was removed. If a process attempts to read from that page of logical memory the system first examines the flag to see if the page is available and, if it is not, a page fault occurs.

Notes

A page fault is a hardware or software interrupt (depending on implementation) which passes control to the operating system. The OS proceeds to locate the missing page in the swap area and move it back into a free frame of physical memory. It then binds the addresses by updating the paging table and, when control returns to the waiting process, the missing page is automatically restored, as if it had never been gone.

Notice, that the location of the physical frame is completely irrelevant to the user process. A frame does not have to be moved back into the same place that it was removed from, because the runtime binding of addresses takes care of its relocation.

5.7.3 Swapping and Paging Algorithms

How does the system decide what pages or processes to swap out? This is another problem in scheduling. A multitude of schemes is available. Here we shall only consider some examples.

Consider the UNIX system a moment. Before paging was introduced, the only way that memory segments could increase their size was to:

1. Try to look for free memory at the end of the current segment and add it to the current segment.
2. Try to allocate a new, larger segment, copy the data to the new segment and deallocate the old one.
3. Swap out the process, reallocate and swap in again.

In this use of swap space, it is clear that a process is swapped out while it is waiting for a suitable hole in to appear in the memory. This might take a long time and it might be immediate. Another case for swapping out a job is if it has been idle (sleeping) for a long time.

On a BSD-like UNIX system, the first three processes to be started are the swapper, init, the and the page daemon. The page daemon is responsible for examining the state of the page-table and deciding which pages are to be moved to disk. Normally the swapper will not swap out processes unless they have been sleeping for a long time, because the pager will first strip them of their inactive pages. It will begin to swap out processes however, if the average load on the system is very high. (The load average is a number based on the kernel's own internal accounting and is supposed to reflect the state of system activity.) This gives 'cheap' processes a chance to establish themselves. It is the page daemon which makes the paging decisions. By copying read-only segments to the swap area at load time, the running overhead of paging out read-only data is removed, since the data are always where we need them in swap space and never change. In modernized versions of UNIX, such as the Solaris systems by Sun Microsystems, read only pages from the code segment are thrown away when they are selected for swap out and then read in from the filesystem if needed again. Moreover, data pages are only allocated swap space when they are forced out of physical memory. These optimizations reflect the fact that modern systems have more physical memory than previously; also disks are getting faster.

Let us now look more generally at how paging decisions are made. The most important aspect of paging is that pages can still be accessed even though they are physically in secondary storage (the disk). Suppose a page fault occurs and there are no free frames into which the relevant data can be loaded. Then the OS must select a victim: it must choose a frame and free it so that the new faulted page can be read in. This is called (obviously) page replacement. The success or failure of virtual memory rest on its ability to make page replacement decisions. Certain facts might influence these algorithms. For instance, if a process is receiving I/O from a device, it would be foolish to page it out — so it would probably I/O locked into RAM. Here are some viable alternatives for page replacement.

Notes

5.7.4 FIFO — First In First Out

The simplest way of replacing frames is to keep track of their age (by storing their age in the frame table). This could either be the date, as recorded by the system clock, or a sequential counter. When a new page fault occurs, we can load in pages until the physical memory is full — thereafter, we have to move out pages. The page which has been in memory longest is then selected as the first to go.

This algorithm has the advantage of being very straightforward, but its performance can suffer if a page is in heavy use for a long period of time. Such a page would be selected even though it was still in heavy use.


Did u know?

FIFOs are used commonly in electronic circuits for buffering and flow control which is from hardware to software. In hardware form a FIFO primarily consists of a set of read and write pointers, storage and control logic.


Caution

Data sent via FIFO is not persisted in memory and can be an unreliable method for data sources. To ensure your data is not lost.

5.7.5 Second Chance

A simple optimization we can add to the FIFO algorithm is the following. Suppose we keep a reference bit for each page in the page table. Every time the memory management unit accesses a page it sets that bit to. When a page fault occurs, the page replacement algorithm looks at that bit and - if it is set to — sets the bit to but jumps over it and looks for another page.

The idea is that pages which are frequently use will have their bits set often and will therefore not get paged out. Of course, this testing incurs an overhead. In the extreme case that all pages are in heavy use the page algorithm must cycle through all the pages setting their bits to zero before finding the original page again. Even then, it might not find a page to replace, if the bit was set again while it was looking through the others. In such a case, the paging system simply fails.

5.7.6 LRU—Least Recently Used

The best possible solution to paging would be to replace the page that will not be used for the longest period of time—but unfortunately, the system has no way of knowing what that is. A kind of compromise solution is to replace the page which has not been used for the longest period. This does not require a crystal ball, but it does require some appropriate hardware support to make it worthwhile. As with all good ideas, it costs the system quite a lot to implement it.

Two possibilities for such an implementation are the following:

- We record the time at which each page was last referenced. Unlike the FIFO scheme above, this means that we have to update the time-stamp every single time memory is referenced, instead of only each time a page is replaced. If the copying operation takes, say, five CPU instructions (jump to update routine, locate page table entry, load system clock time, store system clock time, return), this means — roughly speaking — that the system is slowed down by a factor of around five. This is an unacceptable loss, so unless the memory management unit can do something fancy in hardware, this scheme is not worth the system's time.
- We keep a stack of page addresses, so that the page number of the most recently accessed page is always on the top of the stack. Although this sounds cheaper in principle, since the

Notes

page replacement algorithm never has to search for a replacement — it just looks on top of the stack — it still results in a large system overhead to maintain the stack. We must update a data structure which requires process synchronization and therefore waiting. Again, without special hardware, this is not economical. In practice, many systems use something like the second-chance algorithm above. The UNIX page daemon uses this approach.

5.8 Thrashing

Swapping and paging can lead to quite a large system overhead. Compared to memory speeds, disk access is quite slow — and, in spite of optimized disk access for the swap area, these operations delay the system markedly. Consider the sequence of events which takes place when a page fault occurs:

1. Interrupt/trap and pass control to the system interrupt handler.
2. Save the process control block.
3. Determine cause of interrupt — a page fault.
4. Consult MMU — is the logical address given inside the process' segment, i.e. legal.
5. Look for a free frame in the frame table. If none is found, free one.
6. Schedule the disk operation to copy the required page and put the process into the waiting state.
7. Interrupt from disk signals end of waiting.
8. Update the page table and schedule the process for running.
9. (On scheduling) Restore the process control block and resume executing the instruction that was interrupted.

Such a sequence of operations could take of the order or milliseconds under favorable conditions (although technology is rapidly reducing the timescale for everything). It is possible for the system to get into a state where there are so many processes competing for limited resources that it spends more time servicing page faults and swapping in and out processes than it does executing the processes. This sorry state is called thrashing.

Thrashing can occur when there are too many active processes for the available memory. It can be alleviated in certain cases by making the system page at an earlier threshold of memory usage than normal. In most cases, the best way to recover from thrashing is to suspend processes and forbid new ones, to try to clear some of the others by allowing them to execute. The interplay between swapping and paging is important here too, since swapping effectively suspends jobs.



Did u know?

Thrashing can occur when there are too many active processes for the available memory. It can be alleviated in certain cases by making the system page at an earlier threshold of memory usage than normal. In most cases, the best way to recover from thrashing is to suspend processes and forbid new ones, to try to clear some of the others by allowing them to execute. The interplay between swapping and paging is important here too, since swapping effectively suspends jobs.

Notes

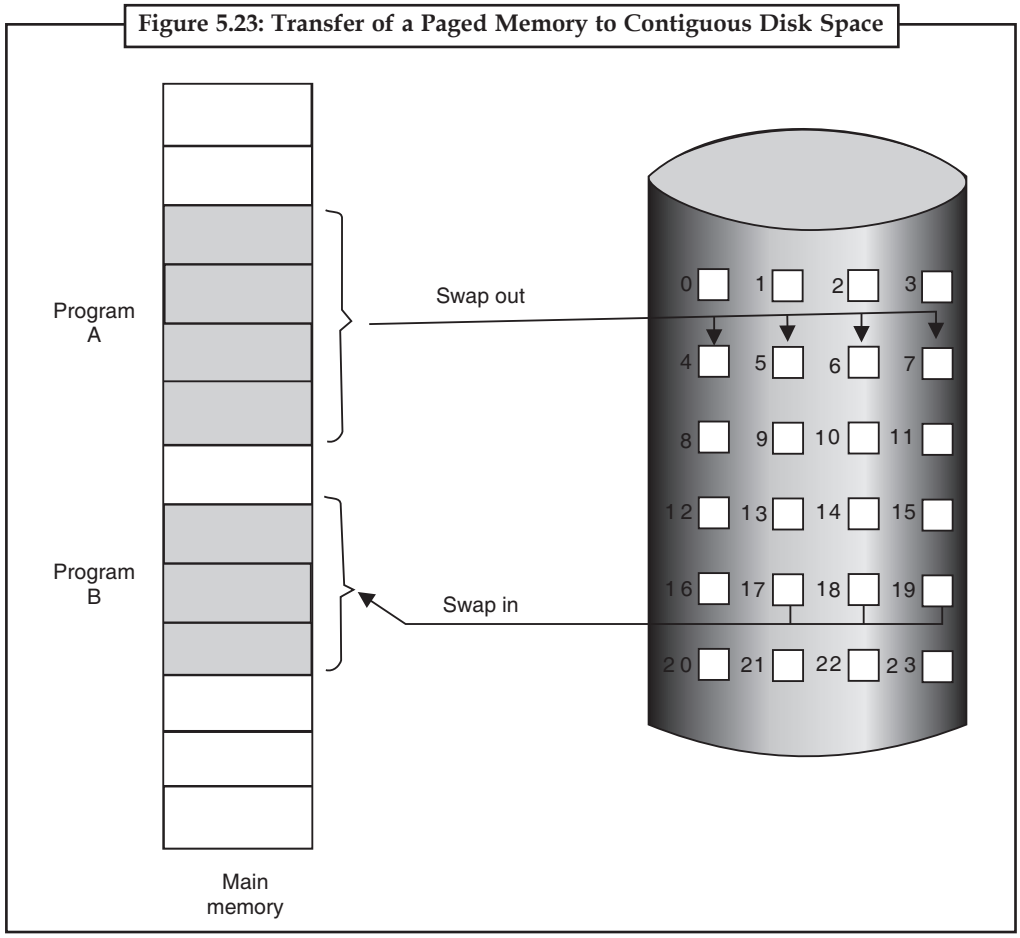
5.9 Demand Paging

A demand-paging system is similar to a paging system. Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of swap is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

5.9.1 Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

Figure 5.23: Transfer of a Paged Memory to Contiguous Disk Space

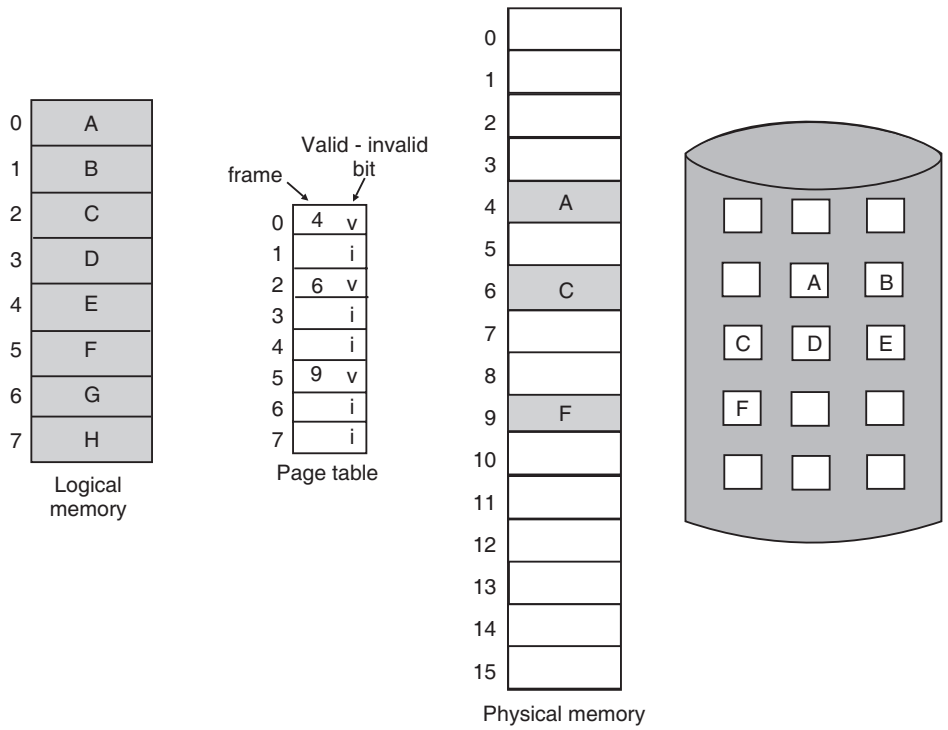


With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme already described and can be used for this purpose. This time, however, when this bit is set to "valid,"

Notes

this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in figure given below. Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are **memory resident,** execution proceeds normally. But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page-fault trap.** The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk-transfer overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore correct this oversight. The procedure for handling this page fault is straightforward (Figure 5.24).

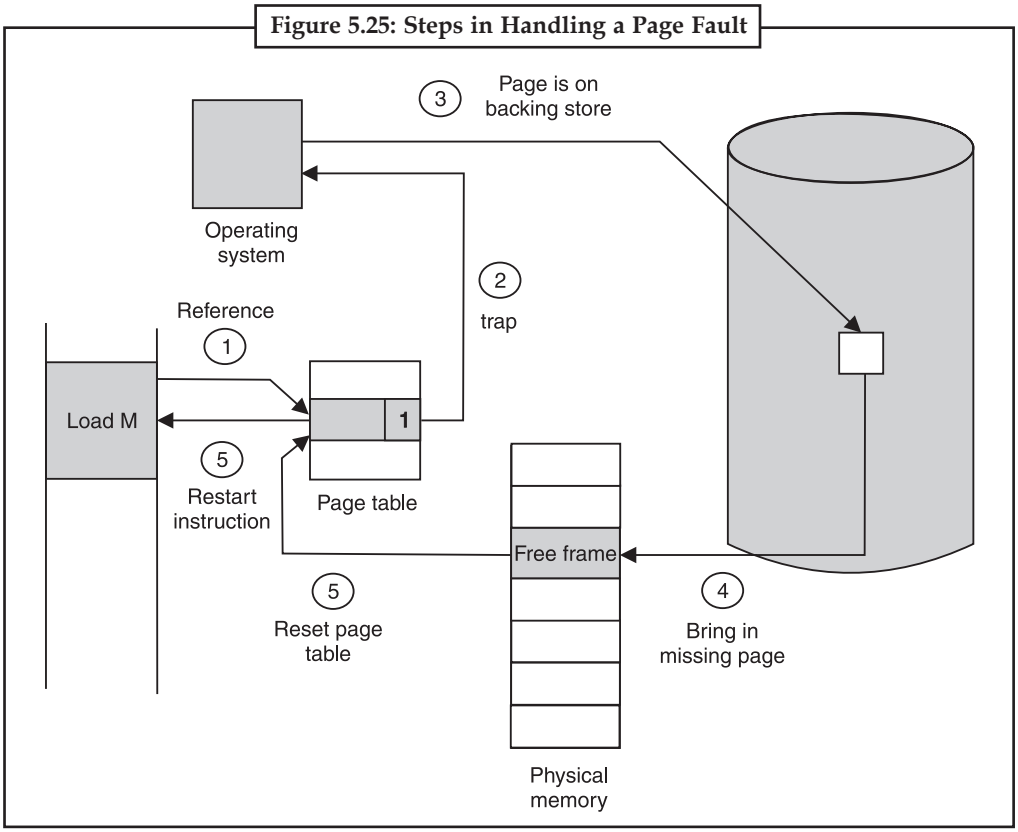
Figure 5.24: Page Table when Some Pages are not in Main Memory



1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).

Notes

Figure 5.25: Steps in Handling a Page Fault



4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory. In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging:** Never bring a page into memory until it is required.

Theoretically, some programs may access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference** which results in reasonable performance from demand paging. The hardware to support demand paging is the same as the hardware for paging and swapping.

Notes

Page Table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

Secondary Memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space.** In addition to this hardware support, considerable software is needed, as we shall see. Additional architectural constraints must be imposed. A crucial one is the need to be able to restart any instruction after a page fault. In most cases, this requirement is easy to meet. A page fault could occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again, and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we faulted when we tried to store in C (because C is in a page not currently in memory), we would have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart would require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs. The major difficulty occurs when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place, as we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

A similar architectural problem occurs in machines that use special addressing modes, including auto decrement and auto increment modes (for example, the PDP-11). These addressing modes use a register as a pointer and automatically decrement or increment the register as indicated. Auto decrement automatically decrements the register before using its contents as the operand address; auto increment automatically increments the register after using its contents as the operand address. Thus, the instruction **MOV (R2) +, - (R3)** copies the contents of the location pointed to by register2 into the location pointed to by register3. Register2 is incremented (by two for a word, since the PDP-11 is a byte-addressable computer) after it is used as a pointer; register3 is decremented (by two) before it is used as a pointer. Now consider what will happen if we get a fault when trying to store into the location pointed to by register3. To restart the instruction, we must reset the two registers to the values they had before we started the execution of the

Notes

instruction. One solution is to create a new special status register to record the register number and amount modified for any register that is changed during the execution of an instruction. This status register allows the operating system to undo the effects of a partially executed instruction that causes a page fault. These are by no means the only architectural problems resulting from adding paging to an existing architecture to allow demand paging, but they illustrate some of the difficulties. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to the user process. Thus, people often assume that paging could be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

5.9.2 Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the **effective access time** for a demand paged memory. For most computer systems, the memory-access time, denoted ma, now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk, and then access the desired word. Let p be the probability of a page fault (0 5 p 5 1). We would expect p to be close to zero; that is, there will be only a few page faults. The **effective access time** is then effective access time = (1 – p) x ma + p x page fault time. To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   (a) Wait in a queue for this device until the read request is serviced.
   (b) Wait for the device seek and/or latency time.
   (c) Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling; optional).
7. Interrupt from the disk (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service time:

Notes

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek of 15 milliseconds, and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device-queuing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

If we take an average page-fault service time of 25 milliseconds and a memory-access time of 100 nanoseconds, then the effective access time in nanoseconds is effective access time = (1 – p) x (100) + p (25 milliseconds) = (1 – p) x 100 + p x 25,000,000 = 100 + 24,999,900 x p. We see then that the effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging. If we want less than 10-percent degradation, we need:

$110 > 100 + 25{,}000{,}000 \times p,$
$10 > 25{,}000{,}000 \times p,$
$p < 0.0000004,$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault. It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically. One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used It is therefore possible for the system to gain better paging throughput, by copying an entire file image into the swap space at process startup, and then performing demand paging from the swap space. Another option is to demand pages from the file system initially, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.

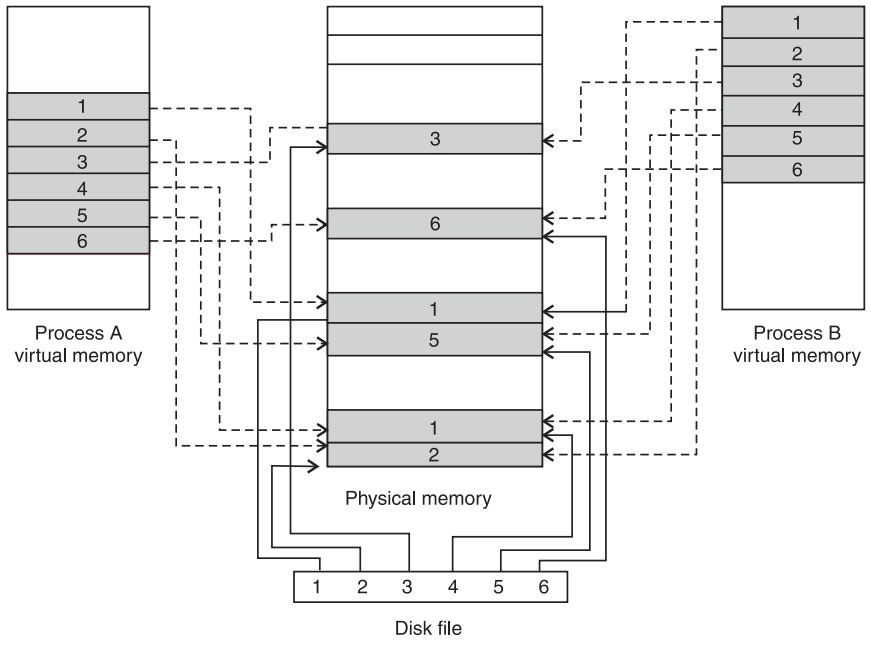
Some systems attempt to limit the amount of swap space when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these pages can simply be overwritten (because they are never modified) and read in from the file system again if needed. Using this approach, the file system itself serves as the baclung store. However, swap space must still be used for pages not associated with a file; these pages include the **stack** and **heap** for a process. This technique is used in several systems including Solaris 2. This method appears to be a good compromise; it is used in BSD UNIX.

Notes

5.10 Overview of Page Replacement

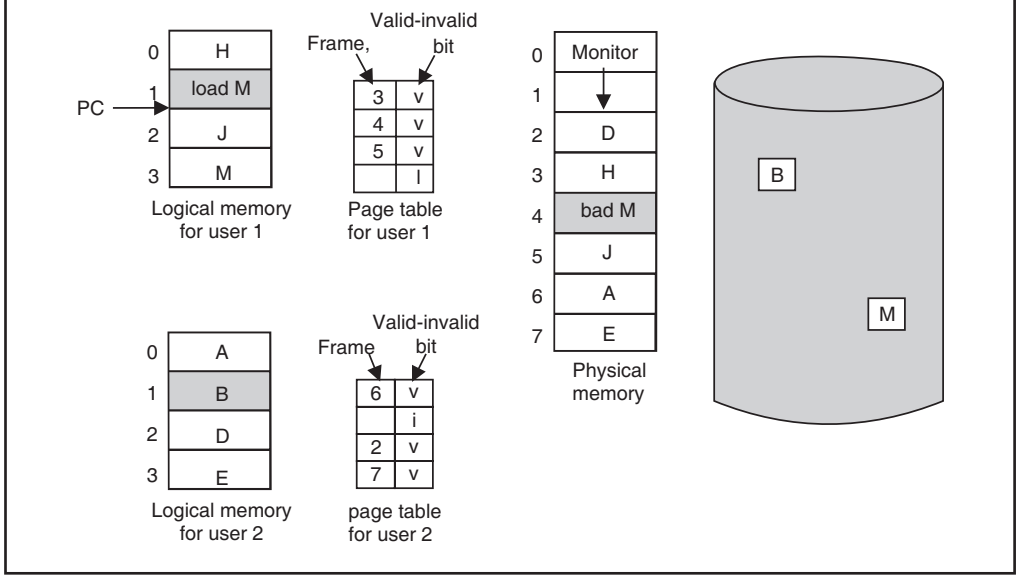
In our presentation so far, the page-fault rate has not been a serious problem, because each page has faulted at most once, when it is first referenced. This representation is not strictly accurate. If a process of ten pages actually uses only one-half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used). If we increase our degree of multiprogramming, we are over-allocating memory. If we run six processes, each of which is ten pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for 60 frames, when only 40 are available. Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory. (In our example, why stop at a multiprogramming level of six, when we can move to a level of seven or eight?) Further, consider that system memory is not used only for holding program pages. Buffers for 1/0 also consume a significant amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory. Over-allocation manifests itself as follows. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free-frame list: All memory is in use. The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput.

Figure 5.26: Memory-mapped Files



Notes

Figure 5.27: Need for Page Replacement



Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So this option is not the best choice. The operating system could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good one in certain circumstances; Here, we discuss a more intriguing possibility—**page replacement.**

5.10.1 Basic Scheme

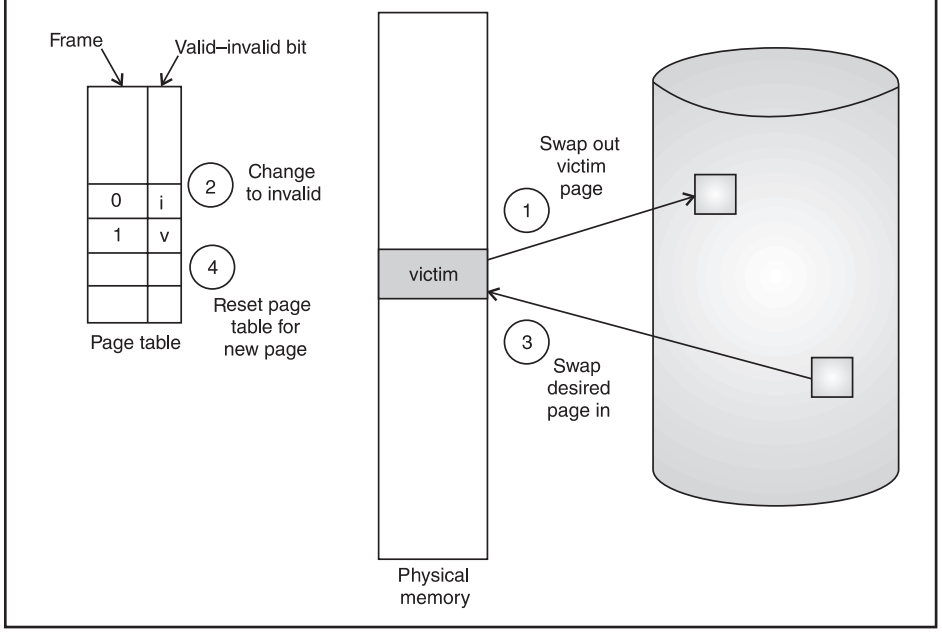
Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   (a) If there is a free frame, use it.
   (b) If there is no free frame, use a page-replacement algorithm to select a victim frame.
   (c) Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify bit (or dirty bit). Each page or frame may have a modify bit associated with it in the hardware. The modify m bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), then we can avoid writing the

Notes

Figure 5.28: Page Replacement



memory page to the disk—it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time required to service a page fault, since it reduces I/O time by one-half if the page is not modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in ten frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a **frame-allocation algorithm** and a **page-replacement algorithm.** If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk 1/0 is so expensive.

Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.** We can generate reference strings artificially (by a random-number generator, for example) or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts. First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Notes

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,

0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string.

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

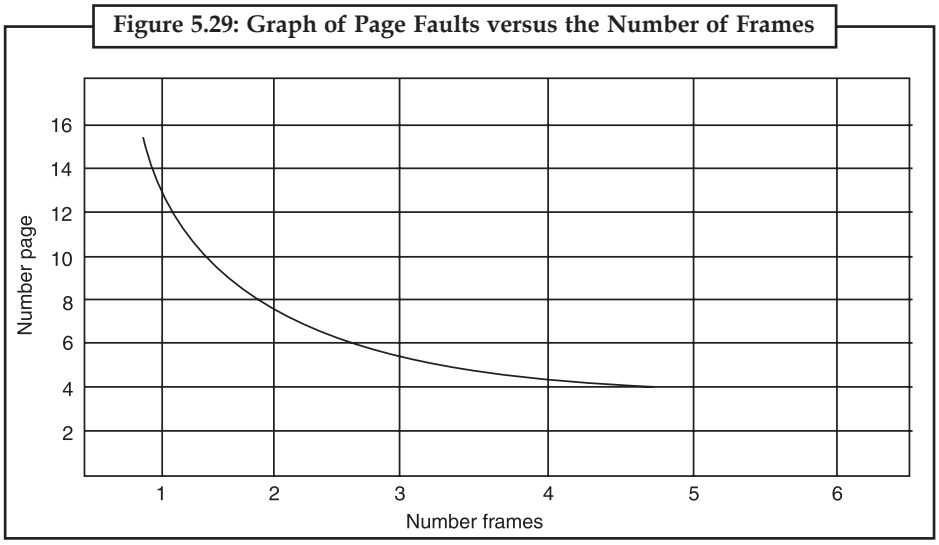
To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

Figure 5.29: Graph of Page Faults versus the Number of Frames



5.10.2 FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue. For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to **3** results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether. The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use. Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to

Notes

retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string page frames.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Figure 5.30: FIFO Page-replacement Algorithm

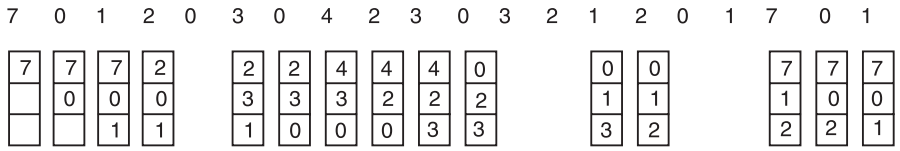
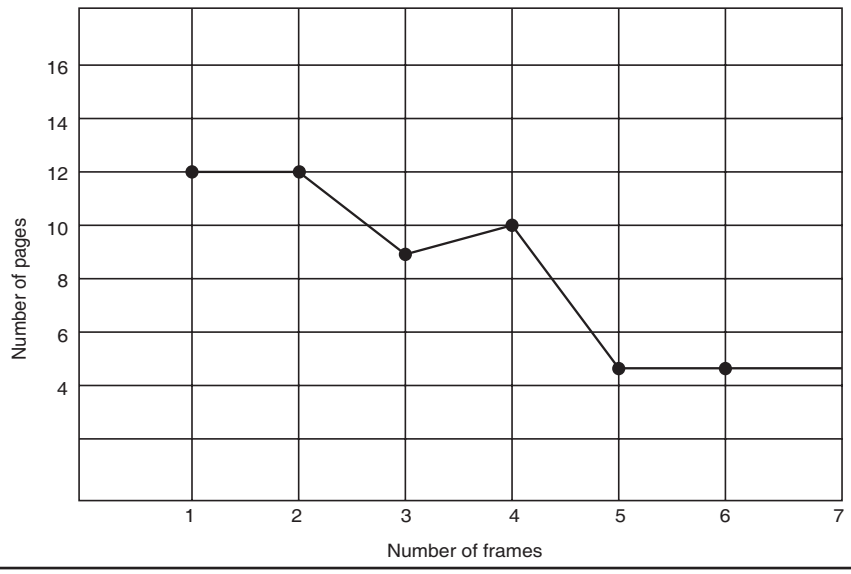


Figure 5.31: Page-fault Curve for FIFO Replacement on a Reference String



5.10.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal.**

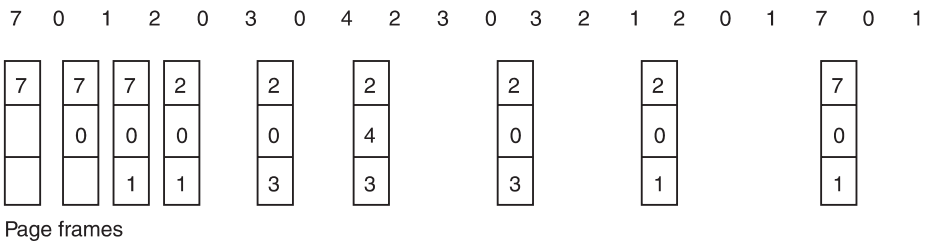
5.10.4 Page-replacement Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults. The first three references cause faults that fill the three empty frames. The reference to page reference string 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the

Notes

reference string. As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 per cent of optimal at worst, and within 4.7 per cent on average.

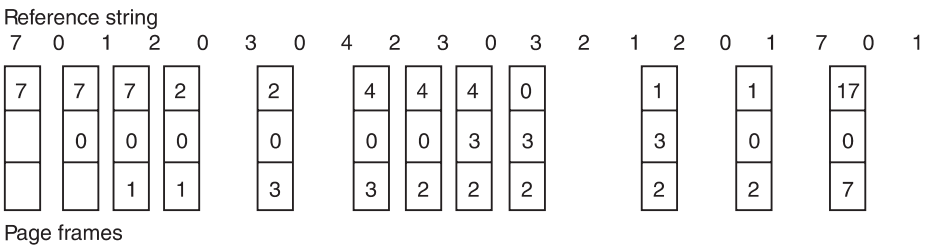
Figure 5.32: Optimal Page-replacement Algorithm



5.11 LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least-recently-used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. Strangely, if we let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R.

Figure 5.33: LRU Page-replacement Algorithm



Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on SR. The result of applying LRU replacement to our example reference string is shown in the Figure 5.33. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15. The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement.



Did u know?

An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible.

Notes

5.11.1 Counters

In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

5.11.2 Stack

Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (figure). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory. Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for every memory reference. If we were to use an interrupt for every reference, to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every user process by a factor of ten. Few systems could tolerate that level of overhead for memory management.



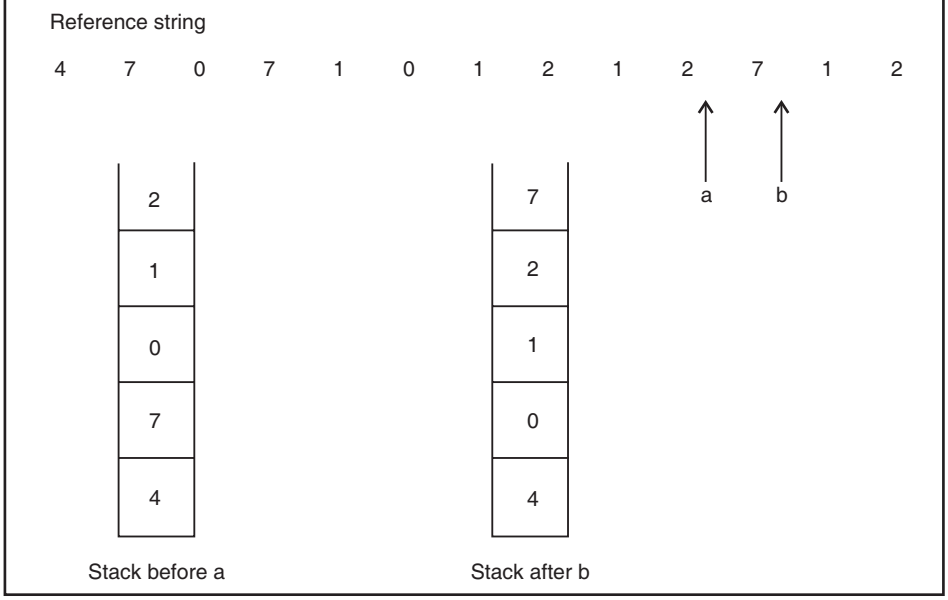
Task How to use stack in memory give steps?

5.11.3 LRU Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the order of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

Notes

Figure 5.34: Use of a Stack to Record the most recent Page References



5.11.4 Additional-Reference-Bits Algorithm

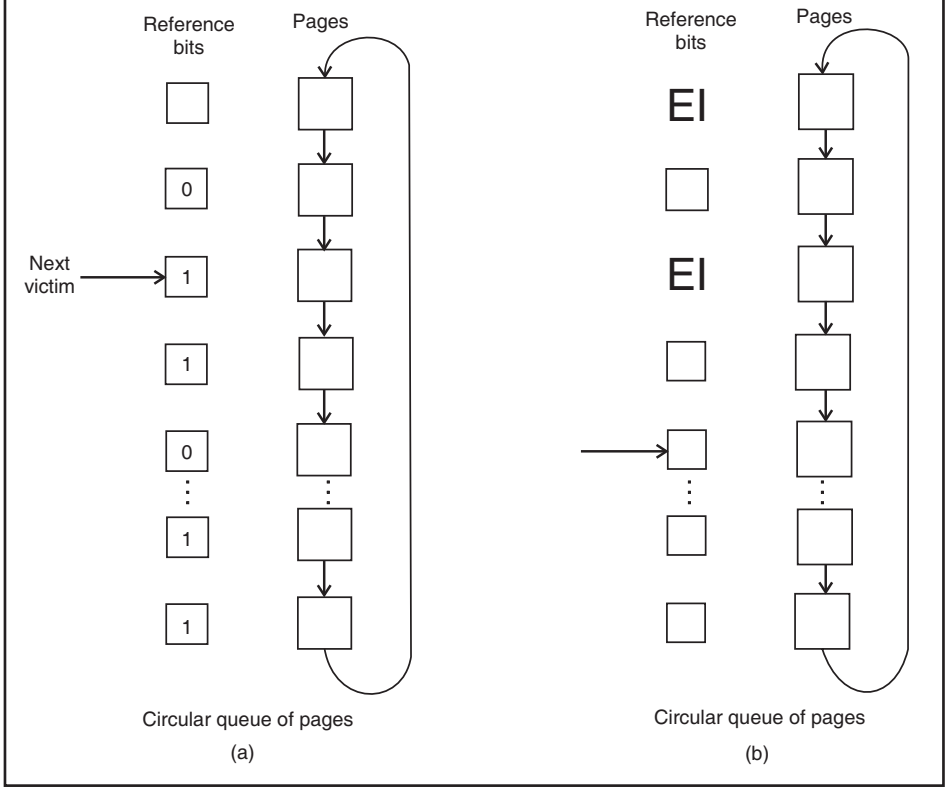
We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These &bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value, or use a FIFO selection among them. The number of bits of history can be varied, of course, and would be selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance page replacement algorithm.

5.11.5 Second-chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced. One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

Notes

Figure 5.35: Second-chance (clock) Page-replacement Algorithm



5.11.6 Enhanced Second-chance Algorithm

We can enhance the second-chance algorithm by considering both the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified—best page to replace
2. (0,l) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1,0) recently used but clean—it probably will be used again soon
4. (1,l) recently used and modified—it probably will be used again soon, and the page will be need to be written out to disk before it can be replaced when page replacement is called for, each page is in one of these four classes.

We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

5.11.7 Counting-based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we could keep a counter of the number of references that have been made to each page, and develop the following two schemes.

Notes

The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used. As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

5.11.8 Page-buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool. An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written out. Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No 1/0 is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

This technique is used in the VAX/VMS system, with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not implement the reference bit correctly.

5.12 Thrashing

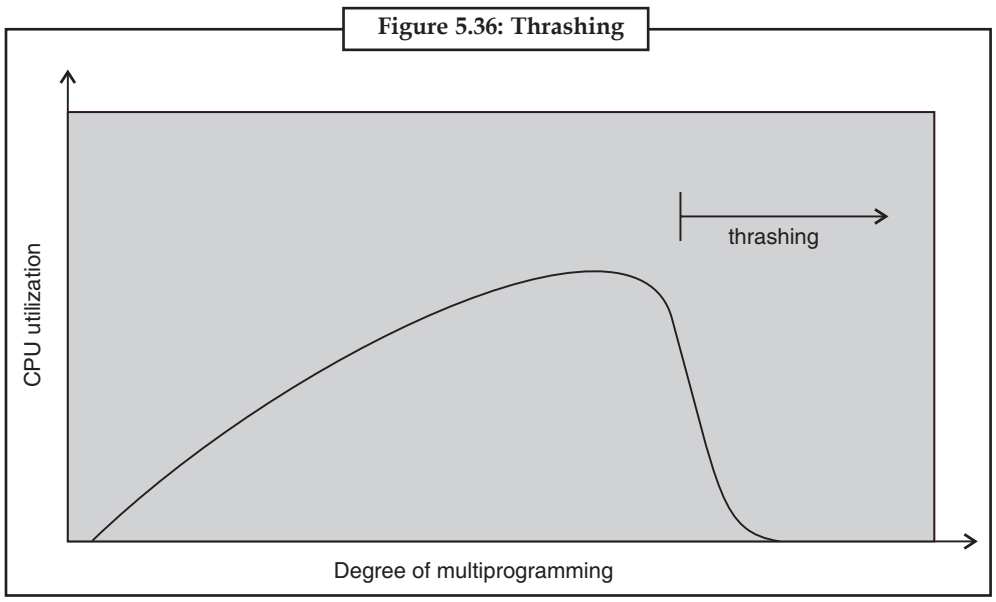
If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling. In fact, look at any process that does not have "enough" frames. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages in active use. If the process does not have this number of frames, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

Notes

5.12.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behaviour of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases. The CPU scheduler sees the decreasing CPU utilization, and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults, and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory access time increases. No work is getting done, because the processes are spending all their time paging. This phenomenon is illustrated in the Figure 5.36, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming. We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part. However, if processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase, due to the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing. To prevent thrashing, we must provide a process as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

Figure 5.36: Thrashing



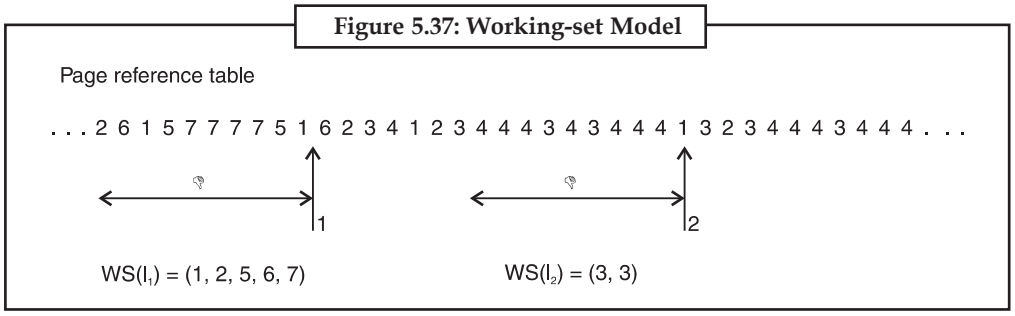
Notes

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap. For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless. Suppose that we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we allocate fewer frames than the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

5.12.2 Working-set Model

The working-set model is based on the assumption of locality. This model uses a parameter, A, to define the working-set window. The idea is to examine the most recent A page references. The set of pages in the most recent A page references is the working set If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set A time units after its last reference. Thus, the working set is an approximation of the program's locality. For example, given the sequence of memory references shown in if A = 10 memory references, then the working set at time tl is (1, 2, 5, 6, 7). By time t2, the working set has changed to {3, 4). The accuracy of the working set depends on the selection of A. If A is too small, it will not encompass the entire locality; if A is too large, it may overlap several localities. In the extreme, if A is infinite, the working set is the set

Figure 5.37: Working-set Model



of pages touched during the process execution. The most important property of the working set is its size. If we compute the working-set size, WSSi, for each process in the system, we can then consider where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames. Use of the working-set model is then simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process pages are written out

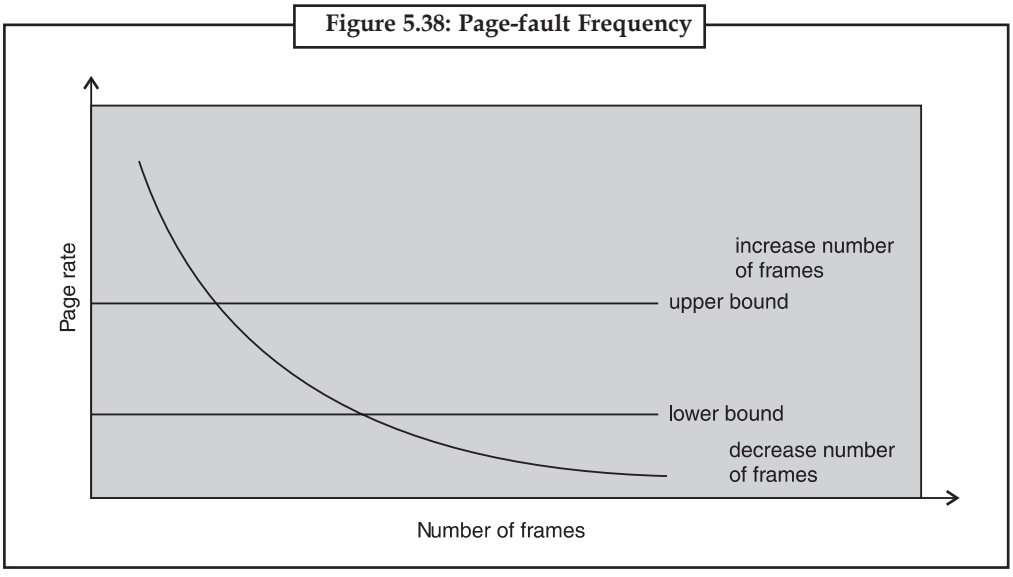
Notes

and its frames are reallocated to other processes. The suspended process can be restarted later. This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window. We can approximate the working-set model with a fixed interval timer interrupt and a reference bit. For example, assume A is 10,000 references and we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and 2 in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least 1 of these bits will be on. If it has not been used, these bits will be off. Those pages with at least 1 bit on will be considered to be in the working set. Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of our history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

5.12.3 Page-fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging but it seems a clumsy way to control thrashing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach.

Figure 5.38: Page-fault Frequency



The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Similarly, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate that process another frame; if the page-fault rate falls below the lower limit, we remove a frame from that process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

Notes

As with the working-set strategy, we may have to suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.



Case Study

Write a program for memory management

```
#include <memory>
#include <iostream>
int main() {
  std::auto_ptr<int> a(new int(3));
  // a.get() returns the raw pointer of a
  std::cout << "a loc: " << a.get() << '\n';
  std::cout << "a val: " << *a << '\n';

  std::auto_ptr<int> b;

  b = a; // now b points to the int, a is null

  std::cout << "b loc: " << b.get() << '\n';
  std::cout << "b val: " << *b << '\n';
  std::cout << "a loc: " << a.get() << '\n';

  return 0;
}
```

Questions:

1. Give the brief about the pointer in memory.
2. Explain the Garbage collection (GC) in memory management.



Lab Exercise

1. Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as a decimal numbers):
   (*a*) 2375
   (*b*) 19366
   (*c*) 30000
   (*d*) 256
   (*e*) 16385
2. C Program for First In First Serve Algorithm.

Self Assessment

Notes

Multiple choice questions:

4. Virtual memory is ..... .
   (*a*) An extremely large main memory
   (*b*) An extremely large secondary memory
   (*c*) An illusion of extremely large main memory
   (*d*) A type of memory used in super computers.
5. The problem of thrashing is effected scientifically by ..... .
   (*a*) Program structure
   (*b*) Program size
   (*c*) Primary storage size
   (*d*) None of the above
6. The mechanism that bring a page into memory only when it is needed is called ..... .
   (*a*) Segmentation
   (*b*) Fragmentation
   (*c*) Demand Paging
   (*d*) Page Replacement
7. The Memory Buffer Register (MBR)
   (*a*) is a hardware memory device which denotes the location of the current instruction being executed.
   (*b*) is a group of electrical circuits (hardware), that performs the intent of instructions fetched from memory.
   (*c*) contains the address of the memory location that is to be read from or stored into.
   (*d*) contains a copy of the designated memory location specified by the MAR after a "read" or the new contents of the memory prior to a "write".
8. The LRU algorithm
   (*a*) pages out pages that have been used recently.
   (*b*) pages out pages that have not been used recently.
   (*c*) pages out pages that have been least used recently.
   (*d*) pages out the first page in a given area.
9. Thrashing
   (*a*) is a natural consequence of virtual memory systems.
   (*b*) can always be avoided by swapping.
   (*c*) always occurs on large computers.
   (*d*) can be caused by poor paging algorithms.

Notes

10. Thrashing can be avoided if

(*a*) the pages, belonging to the working set of the programs, are in main memory.

(*b*) the speed of CPU is increased.

(*c*) the speed of I/O processor is increased.

(*d*) all of the above.

5.13 Summary

- The memory management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies.
- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible.
- Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous.
- Hardware Support Each operating system has its own methods for storing page tables.
- Most allocate a page table for each process.
- Memory protection in a paged environment is accomplished by protection bits 11 that are associated with each frame. Normally, these bits are kept in the page table.
- Another advantage of paging is the possibility of sharing common code.
- Swapping and paging can lead to quite a large system overhead.
- A demand-paging system is similar to a paging system.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.

5.14 Keywords

Compile Time: It refers to either the operations performed by a compiler (the "compile-time operations"), programming language requirements that must be met by source code for it to be successfully compiled (the "compile-time requirements"), or properties of the program that can be reasoned about at compile time.

Fragmentation: A multiprogrammed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

Global Descriptor Table (GDT): Is specific to the IA32 architecture. It contains entries telling the CPU about memory segments. A similar Interrupts Descriptor Table exists containing tasks and interrupts descriptors. Read the GDT Tutorial.

Hashed Page Tables: A common approach for handling address spaces larger than 32 bits.

Local Descriptor Table (LDT): Is a memory table used in the x86 architecture in protected mode and containing memory segment descriptors: start in linear memory, size, executability, writability, access privilege, actual presence in memory.

Notes

***Memory-Management Unit (MMU)*:** The run-time mapping from virtual to physical addresses is done by a hardware device.

Relocation: One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory without the program noticing the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

Secondary Memory: This memory holds those pages that are not present in main memory.

Translation Look-Aside Buffer (TLB): Standard solution to this problem is to use a special, small, fast look up hardware cache.

5.15 Review Questions

1. Describe how the Swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.
2. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?
3. Most systems allow programs to allocate more memory to its address space during execution. Data allocated in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes:
   (a) contiguous-memory allocation
   (b) pure segmentation
   (c) pure paging
4. On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?
5. Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.
6. Why are segmentation and paging sometimes combined into one scheme?
7. Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.
8. Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

Notes

9. What is the purpose of paging the page tables?
10. Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when an user program executes a memory load operation?
11. Compare the segmented paging scheme with the hashed page tables scheme for handling large address spaces. Under what circumstances is one scheme preferrable over the other?
12. Discuss the hardware support required to support demand paging.
13. What is the copy-on-write feature and under what circumstances is it beneficial to use this feature? What is the hardware support required to implement this feature?
14. A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 218 bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.
15. Discuss situations under which the least frequently used page-replacement algorithm generates fewer page faults than the least recently used page replacement algorithm. Also discuss under what circumstance does the opposite holds.
16. Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

17. Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discarding that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?
18. A page-replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

    (*a*) Define a page-replacement algorithm using this basic idea. Specifically address the problems of (1) what the initial value of the counters is, (2) when counters are increased, (3) when counters are decreased, and (4) how the page to be replaced is selected.

    (*b*) How many page faults occur for your algorithm for the following reference string, for four page frames?

    1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

    (*c*) What is the minimum number of page faults for an optimal page replacement strategy for the reference string in part b with four page frames?

Notes

19. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
20. Is it possible for a process to have two working sets? One representing data and another representing code? Explain.

Answers to Self Assessment

1. (*d*) 2. (*d*) 3. (*d*) 4. (*c*) 5. (*a*)
6. (*c*) 7. (*d*) 8. (*c*) 9. (*d*) 10. (*b*)

5.16 Further Readings



Books

Operating Systems, by Staurt E. Madnick, John J. Doonovan.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.



Online link

wiley.com/coolege.silberschatz

Notes

Unit 6: File Management

CONTENTS

Objectives

Introduction

6.1 Managing Files in Windows

6.1.1 Managing Files from within a Program

6.1.2 Using My Computer

6.1.3 Using Windows Explorer

6.1.4 Working with More than One File

6.1.5 Locating Lost Files

6.2 File Concepts

6.2.1 Basic File Operations

6.3 File Access

6.4 Access Methods

6.4.1 Sequential Access

6.4.2 Direct Access

6.4.3 Mapped Access

6.4.4 Structured Files

6.4.5 Binding of Access Methods

6.5 Directory Structures

6.5.1 Single-level Directory

6.5.2 Two-level Director

6.5.3 Tree-structured Directory

6.5.4 Acrylic-structured Directory

6.5.5 Paths

6.6 WebSphere Portal Directory Structure

6.6.1 Directories for Languages

6.7 File Sharing

6.7.1 Multiple Users

6.7.2 Remote File Systems

6.7.3 The Client-Server Model

Notes

6.7.4 Distributed Information Systems

6.7.5 Failure Modes

6.8 Consistency Semantics

6.8.1 UNIX Semantics

6.8.2 Session Semantics

6.8.3 Immutable-Shared-Files Semantics

6.9 Protection Mechanisms

6.9.1 Protection of Memory

6.9.2 User-oriented Access Control

6.9.3 Protection Based on an OS Mode

6.10 Allocation Methods

6.10.1 Contiguous Allocation

6.10.2 File Allocation Methods—Chained

6.10.3 File Allocation Methods—Indexed

6.11 Free-Space Management

6.11.1 Bit Vector

6.11.2 Linked List

6.11.3 Grouping

6.11.4 Counting

6.12 Directory Implementation

6.12.1 Simple Directories

6.12.2 Hierarchical Directory Systems

6.12.3 Path Names

6.12.4 Directory Operations

6.13 Summary

6.14 Keywords

6.15 Review Questions

6.16 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Explain file concepts
- Understand file access
- Discuss access methods
- Explain directories for languages
- Understand file sharing
- Discuss protection mechanisms
- Explain allocation methods
- Understand free-space management
- Discuss directory implementation

Introduction

A file is a collection of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, data transfer rate and access methods.

Drive: The piece of hardware that holds and runs disks; used as a top-level location criterion for a file. Your "hard disk" or "hard drive" is usually designated with the letter "C," while your floppy disk/drive is usually named "A."

File: One document, one image, one something. In the world of computing, the terms "folder" and "file" are entirely separate, distinct, and no interchangeable. Folders contain files; files cannot contain folders. Files are represented by various icons that indicate which program is used to open them:

File extension: The two or three or four letters after the dot in a filename. The file extension

indicates what kind of file it is: its "format" or "type." For instance, the file extension .exe refers

to an "executable" file-in other words, an application. The file extension .html indicates a

Hypertext Markup Language file—in other words, a web page. In My Computer or Windows

Explorer, double-clicking on a file will open it if the file extension is correct. Some common file extensions:.

doc – Microsoft Word document

wpd – WordPerfect Document

txt – Plain text document

htm, .html – A plain text document with added code that enables it to be read on the World Wide Web

jpg – An image file

gif – An image file

Notes

exe – An executable file, meaning an application/program/piece of software

Filename: The name of a file, including or not including its file extension.

File Size: The size of a file measured in bytes. A floppy disk holds about 1.5 Mb; a Zip disk holds 100 Mb or 250 Mb; a CD holds about 800 Mb; a DVD holds about 4,700 Mb.

1,000 bytes = 1 kilobyte (Kb)

1,000 kilobytes = 1 megabyte (Mb)

1,000 megabytes = 1 gigabyte (Gb)

1,000 gigabytes = 1 terabyte (Tb)

Folder: Also "directory." A division of a drive into which you put files or further folders (which are then called subdirectories). In both Windows and Mac, a folder subfolder is represented by an icon that looks like a manila folder:

Path: The exact location of a file, including drive letter, directory, subdirectory, and filename, as in the following: C:\My Documents\TTSP\Basic File Management.doc .


Notes

Six main major activities of an operating system in regard to file management are:

1. The creation and deletion of files.
2. The creation and deletion of directions.
3. The support of primitives for manipulating files and directions.
4. The mapping of files onto secondary storage.
5. The back up of files on stable storage media
6. A file is a collection of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, data transfer rate and access methods.

6.1 Managing Files in Windows

There are three ways of managing files in Windows operating systems:

- From within a program;
- By using My Computer; and
- By using Windows Explorer.

6.1.1 Managing Files from within a Program

When you choose "File" → "Save As" from within a program such as Microsoft Word, a dialogue box appears with three important features:

"Save in" (near the top of the box).

Notes

A dropdown box that brings up your computer's directory structure, to allow you to choose where to save your file.

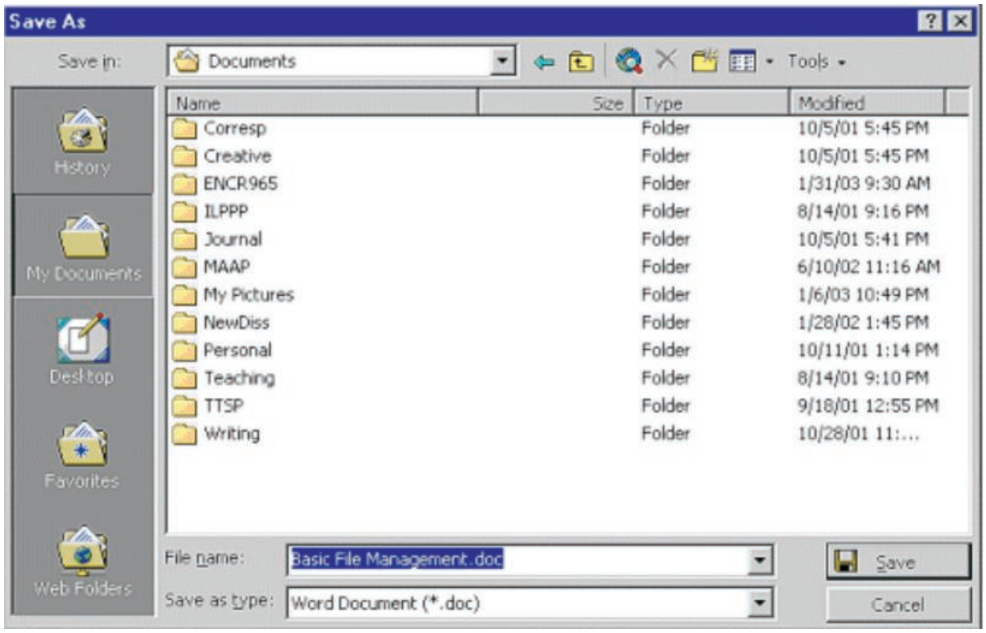
"File name" (near the bottom of the box).

Allows you to type in a name for your file.

"Save as type" (at the bottom of the box).

A dropdown box that allows you to choose a format (type) for your file. The default file format will appear with the default file extension.

Figure 6.1: File Management in Window



These three options also appear when you choose "File" "Open" from within Microsoft Word, but they have slightly different names. Other programs will have the same three options, which again might have slightly different names.

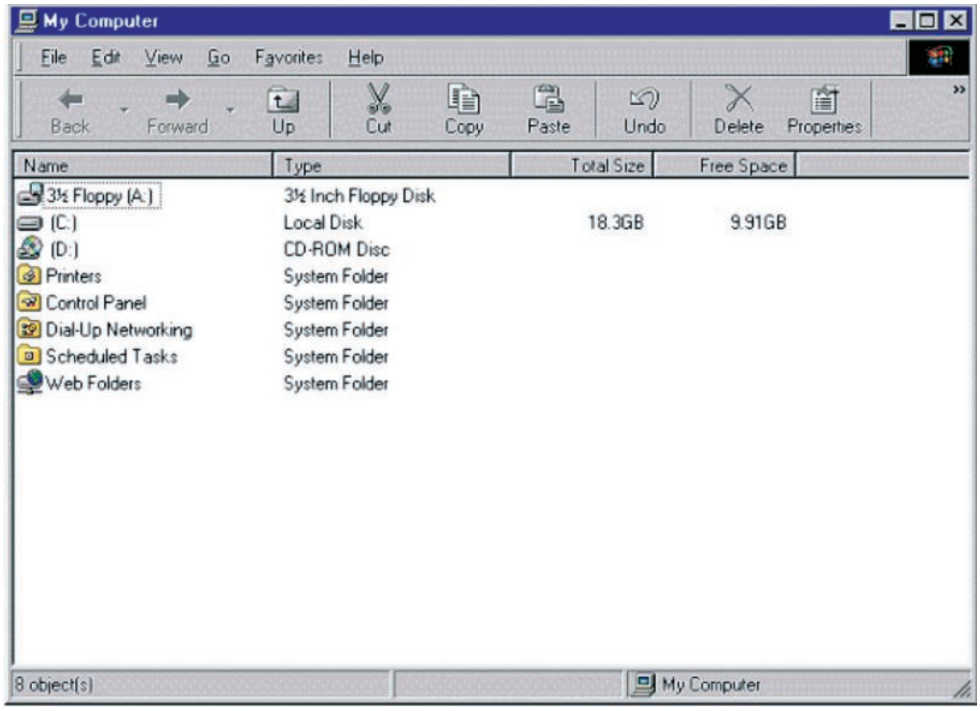


6.1.2 Using My Computer

Double-clicking on the My Computer icon, which should be located in the upper left-hand corner of your desktop, will open a window labeled My Computer. From within this window, you can open, move, copy, and delete files; you can also create, move, copy, and delete folders. Double clicking on any folder icon also opens My Computer, but you will see the contents of that directory rather than the contents of your computer.

Notes

Figure 6.2: Double Clicking Process



At the "top" level of the directory structure are the drives, differentiated by letters:

A:\ is your floppy disk drive

C:\ is your hard disk

D:\ is your Zip, CD, or DVD drive

J:\ is your Home Directory (your blue.unix.Virginia.edu account). This drive only appears when you have logged in to your Home Directory.

Go to "View" at the top of the window to change the way files and folders are displayed within the window. There are four ways to view files and folders:

Large icons

Small icons

List: Choose this when you want to work with several files or folders at a time.

Details: This is a good mode to work in when you want to see when the file was created, its size, and other important information.

- The toolbar has several buttons that enable you to work with files and folders:



Notes

Up: Choosing "Up" enables you to navigate through the computer's directory structure quickly. Clicking on this button will change the contents of the current window, taking you "up" in the directory structure until you get to the highest level, at which only the drives are shown.

Cut: When you single-click on a file or folder to select it, it will be highlighted in the window. Choosing "Cut" will delete the file or folder from its current location and copy it to the clipboard so that it can be pasted elsewhere.

Copy: Choosing "Copy" will copy a selected file or folder into the clipboard so that it can be pasted elsewhere, but will not remove the file or folder from its current location.

Paste: Choosing "Paste" will paste a file or folder that is stored in the clipboard into the current location.

Undo: Choosing "Undo" allows you to undo an action that you have just performed. This is particularly useful when you have just deleted something you didn't mean to delete.

Delete: Choosing "Delete" will delete a selected file or folder without copying it to the clipboard.

Properties: Choosing "Properties" will bring up a box that gives you information about a particular file or folder.

To create a new folder in the current window, you can do one of two things:

Go to "File" → "New" → "Folder." A new folder appears in the current window,

and the folder name is highlighted that will allow you to name it.

Right-click anywhere in the current window (not on an icon or filename) and

choose "New" → "Folder."

Right-clicking on a selected file or folder will allow you to do several useful things, among which are the following:

Rename a file or folder by choosing "Rename." A blinking cursor will appear in the file or folder name.

Create a desktop shortcut by choosing "Send To" → "Desktop as Shortcut."

Copy the file or folder to a floppy disk by choosing "Send To" → "3 ½ Floppy (A:)."

Cut, copy, paste, or print a file.



6.1.3 Using Windows Explorer

In Windows Explorer, the entire directory structure is always available at all times in the left-hand pane. In this respect it differs from My Computer.

Another difference between Windows Explorer and My Computer is that Windows Explorer allows you to drag-and-drop files and folders with the mouse.

Notes

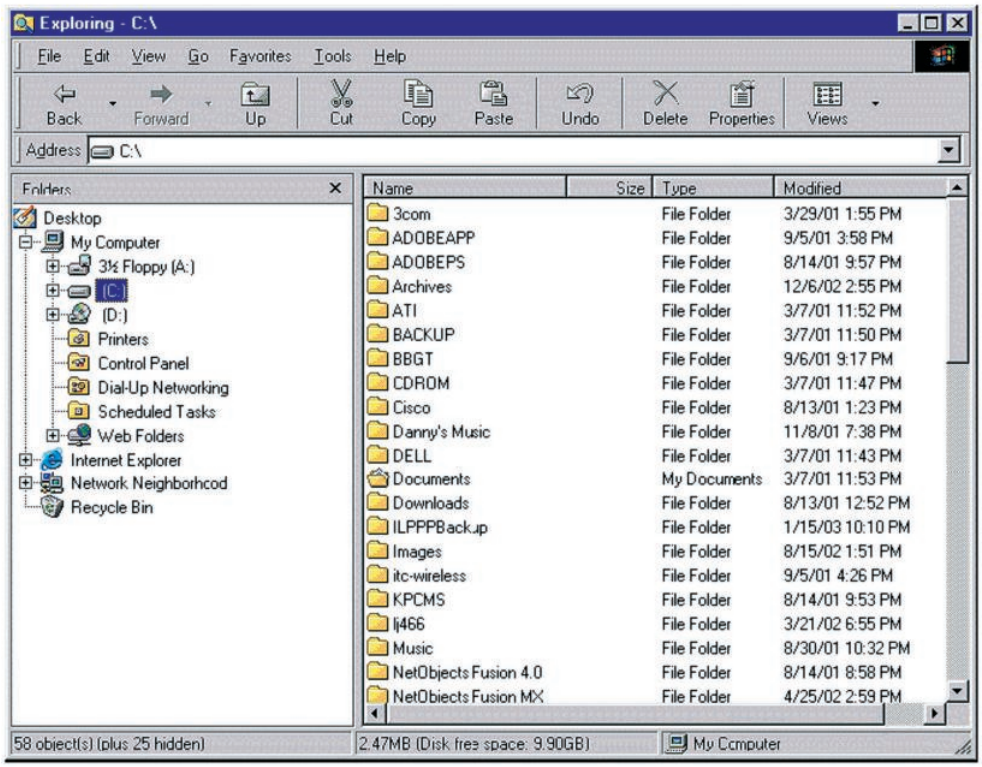
In the left-hand pane, drives, directories, and subdirectories are visible. To expand your view of the contents of a drive or directory, click on the + sign next to the directory name. To collapse your view of the contents of a drive or directory, click on the – sign next to the directory name.

To see the contents of a drive or directory, click once on it (i.e., select it). In the right hand pane, the contents of the selected drive or directory are then displayed. The right hand pane functions just like the windows in My Computer. In the Figure 6.3, the drive C:\ is selected, and its contents are shown in the right-hand pane.


Did u know?

Another difference between Windows Explorer and My Computer is that WindowsExplorer allows you to drag-and-drop files and folders with the mouse.

Figure 6.3: Windows Explorer

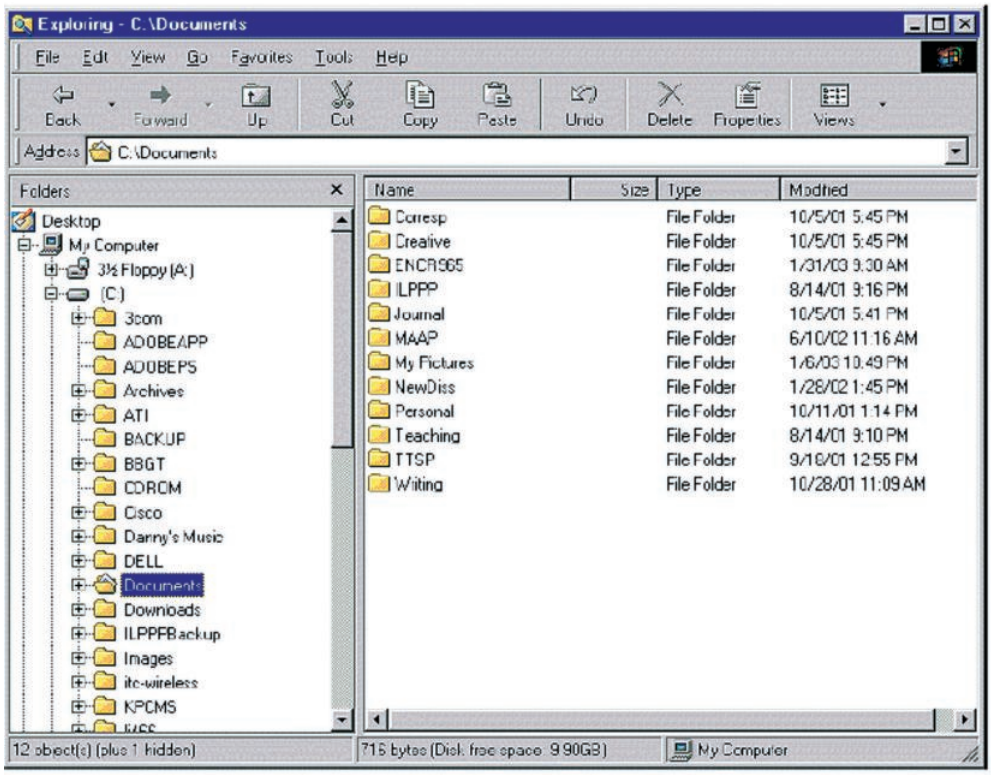


In the next example, the drive C:\ has been expanded, and the directory "Documents" has been selected. Its contents are displayed in the right-hand pane.


Task How to save file another drive in computer system?

Notes

Figure 6.4: Explorering Document



Sorting Files

You can sort files in My Computer and Windows Explorer by clicking once on the Name, Size, Type, or Modified header buttons. To sort the files with the most recent listed first, for instance, click once on "Modified." To re-sort them with the earliest listed first, click again on "Modified."



6.1.4 Working with More than One File

To select two or more separate files, hold down the "Ctrl" key and click on each filename. To select a contiguous group of files in a list, click on the first filename, then hold down the "Shift" key and click on the last filename. All files in between will also be selected. You can then perform cut, copy, and delete functions on all the selected files.

6.1.5 Locating Lost Files

Use the "Find File" facility of your operating system by going to "Start" → "Find" → "Files or Folders." A box will appear that will allow you to search for a file by name, by part of its name (use * as a wildcard character), by location, by date, by type, or by other criteria.

Notes

6.2 File Concepts

A file is an abstract data type, a "thing", which is defined and implemented by the operating system. The main task of the operating system is to map the logical file concept onto physical storage devices, such as disks or tapes.

A file

Has a **type** (.com,.bat,exe,…) which can either be known and managed by the operating system, or leaving the interpretation to the application process.

Some systems allow different file operations based on type.

For general purpose systems it is more effective to implement only basic types and grant maximum freedom to the processes.

For specialized systems, i.e. a database system, it "may" be more efficient to implement the logic in the operating system.

Consists of a **sequence of logical records**, which can be of type byte, a fixed or variable length line, or a complex data type (structure).

The O/S becomes bulky and complex if it supports too many file structures.

Assuming a file to be an array of bytes and deferring interpretation to the applications ensures that the O/S is simplified.

6.2.1 Basic File Operations

- **Create:** find space for the file and make an entry in the directory.
- **Open:** find file and determine if it has already been opened. If not open search directory, cache information, add entry in per-process open-file table. If open check lock and cache information if lock can be acquired. Increment the open count.
- **Close:** decrement the open count and remove the file's entry from the open-file table if count reaches zero.
- **Read:** read data from the file.
- **Write:** write data to the file.
- **Delete:** search directory, release file space and erase directory entry.
- **Reposition:** reposition the file position pointer. This is more commonly known as seek.
- **Truncate:** delete content of a file, but keep file properties.
- **Lock:** file locks provide concurrency control. A shared lock allows multiple "readers" to acquire a lock concurrently, while exclusive lock ensures only one "writer" can modify a file.With mandatory locking the operating system ensures locking integrity, while with **advisory locking** the application process ensures that the correct locking strategy is followed. The Windows operating system uses the mandatory locking strategy.

Notes

Self Assessment

Multiple choice questions:

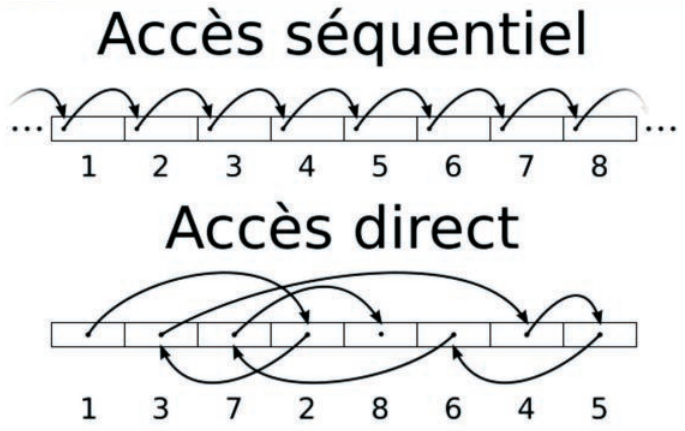
1. Which of the following systems software does the job of merging the records from two files into one?
   (*a*) Security software (*b*) Utility program
   (*c*) Networking software (*d*) Documentation system
   (*e*) None of the above
2. Which directory implementation is used in most Operating System?
   (*a*) Single level directory structure (*b*) Two level directory structure
   (*c*) Tree directory structure (*d*) Acyclic directory structure

Fill in the blanks:

3. A database system is more efficient to implement the ..... in the operating system.
4. To select two or more separate file, hold down the ..... key and click on each filename.
5. We can rename a file or folder by choosing ..... option.

6.3 File Access

- **Sequential:** Process files from beginning to end, in order, one record after the other. For example, if we want to read a document from beginning to end, we typically start at the beginning and read page, by page until we reach the end.
- **Direct:** Process file by accessing the content in no specific order. For example, if we only want to read page 1013 it makes sense to reposition (seek) to page 1013 and read the page.



6.4 Access Methods

An access method defines the way processes read and write files. We study some of these below.

6.4.1 Sequential Access

Under this access method, the entire file is read or written from the beginning to the end sequentially. Files in popular programming languages such as Pascal and Ada provide such access. The file is associated with a **read/write mark**, which is advanced on each access. If several

Notes

processes are reading or writing from the same file, then the system may define one read/write mark or several.



In the former case, the read/write mark is kept at a central place, while in the latter case it is kept with the process table entry. In Unix, a combination of the two schemes is provided, as we shall see later.

6.4.2 Direct Access

This access allows a user to position the read/write mark before reading or writing. This feature is useful for applications such as editors that need to randomly access the contents of the file.

6.4.3 Mapped Access

The Multics operating systems provide a novel form of access which we shall call **mapped access**. When a process opens a file, it is mapped to a segment. The open call returns the number of this segment. The process can thus access the file as part of its virtual store. The Close Segment call may be used to close the file.

6.4.4 Structured Files

So far, we have treated files as byte streams. Database applications often wish to treat them as records that may be accessed by some key. To accommodate these applications, some systems support typed or structured files that are considered streams of records. If a file is structured, the owner of the file describes the records of the file and the fields to be used as keys. OS/360 for IBM computers and DEC VMS provide such files.

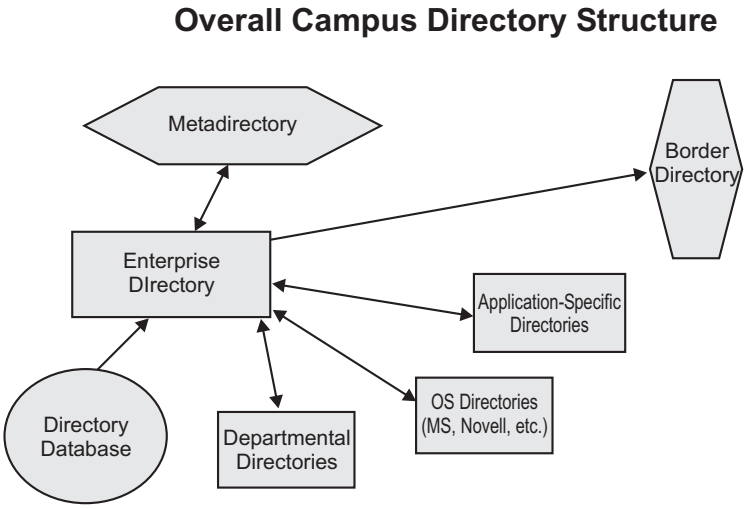
6.4.5 Binding of Access Methods

An access method may be specified at various times:

When the operating system is designed. In this case, all files use the same method. When the file is created. Thus, every time the file is opened, the same access method will be used. When the file is opened. Several processes can have the same file open and access it differently.

6.5 Directory Structures

A disk is typically partitioned, also known as slices and minidisks. The device directory or volume table of contents records and maintains the file properties such as name, size, type and location.



Notes

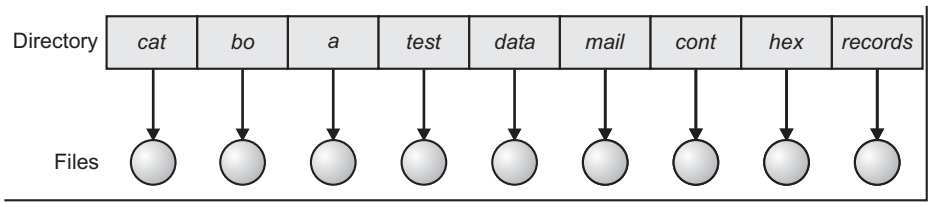
6.5.1 Single-level Directory

All files are contained in the same directory, which can be a daunting directory structure as number of files increases. Using long filenames and using the character "." we could (but really should not) simulate multi-level directory:

mail.willy.inbound.mailB.txt

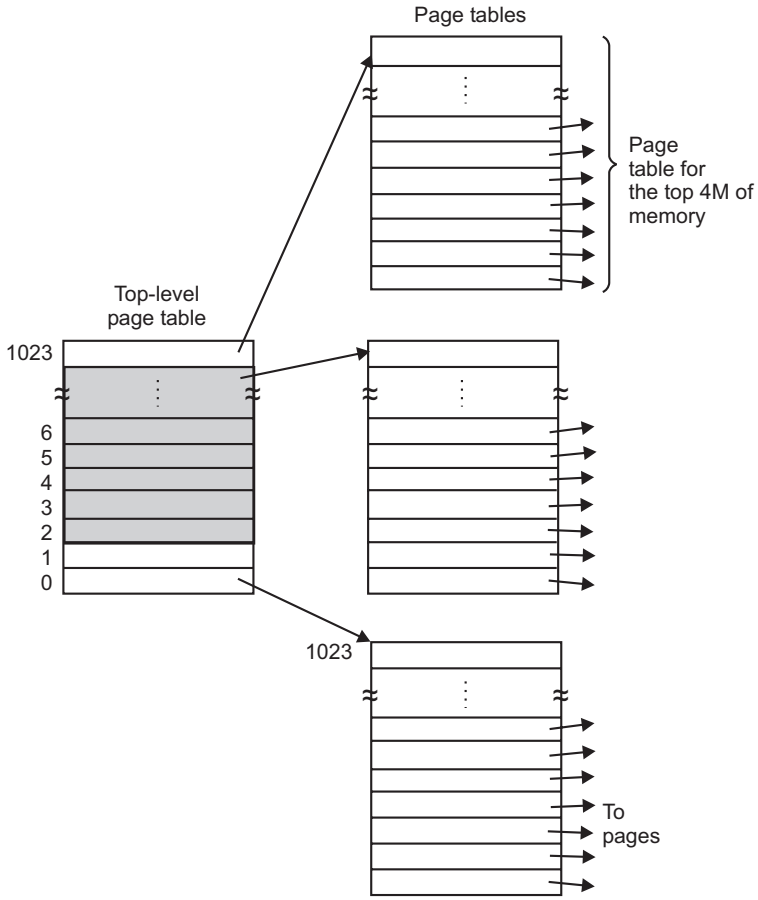
mail.willy.inbound.mailC.txt

mail.willy.deleted.mailA.txt



6.5.2 Two-level Directory

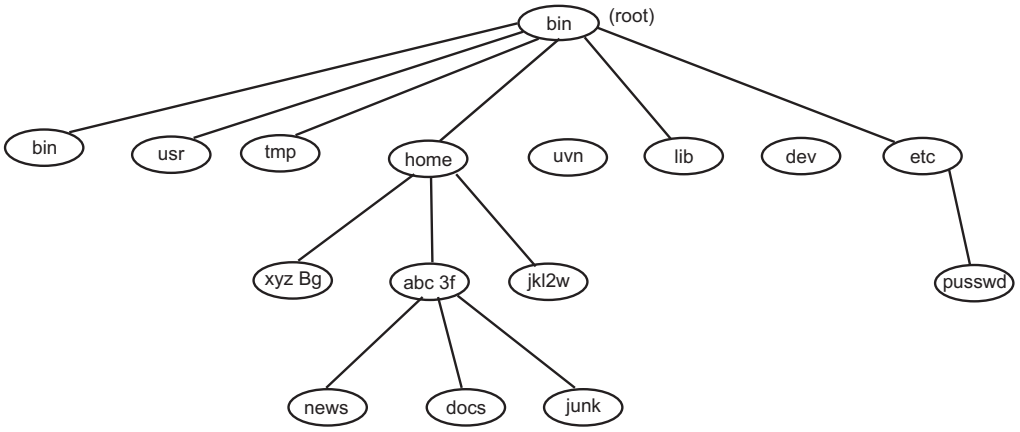
Each user has his own user file directory, which contains the files for each user.



Notes

6.5.3 Tree-structured Directory

An extension to the two-level directory to a tree of arbitrary height. Each directory can contain one or more directories and/or files.



6.5.4 Acrylic-structured Directory

Allows directories to share sub-directories and files. The sharing is achieved by creating links that point to another file or directory, implemented as an absolute or relative path.

6.5.5 Paths

- **Absolute path** begins at the root and follows a path down to a specified file. Example: c:\temp\x\demo\testing\hello.cpp.
- **Relative path** defines a path from the relative path. Example: if we are located in c:\temp\x and relative path is \test\log.txt, then its absolute path would be c:\temp\test\log.txt. This topic shows the naming conventions used to denote the location of files on the portal server and the types of resources you can find in those directories.

6.6 WebSphere Portal Directory Structure

Throughout this documentation, the install location for the portal server component of WebSphere Portal is noted as wp_root, which is the config root.

The following table shows the default location if it is not otherwise specified during installation:

Operating system	Location
z/OS	/usr/lpp/zPortalServer/V5R1M0 (install root) /PortalServer/V5R1M0/Portal1 (config root)

z/OS: The portal server has the following directory structure on z/OS after installation. All directories are r/w on the config root. The content, however, might be r/w (files are copied) or r/o (files are symlinked).

wp_root	Root directory for WebSphere Portal.
|	
| +— bin (mixed)	WebSphere Portal tools (scripts must be r/w; all

Notes

```
|                          other files r/o).
|
+— cloudscape (r/w)        Cloudscape database files.
|
+— config                  Portal configuration files.
| |
| + actions (r/o)
| |
| + helpers (r/w)
| |
| + includes (r/w)
| |
| + scripts
| |
| + templates (r/o)
| |
| + was (r/w)
| |
| + work (r/w)
|
|— content
|
+— deployed (r/w)          Copies of the .ear and .war files for installed portlet applications.
|
|
+— doc (r/o)               WebSphere Portal Information Center and Javadoc.
|
+— IBMTrans (r/w)          Transcoding component (these subdirectories are linked r/o: bin, lib,
                           plugins, toolkit, XMLConfigRules, and xmlconfig_stage).
|
|
|
|
|
+— install (r/o)          Contains wps.ear.
|
+— installableApps (r/w)WAR files prior to deployment.
|
+— installableConnectors
|
+— installedApps (r/w)    Active portlet applications extracted to the WAR file directory
                           structure
|(created during installation).
|
|
+— itlm                   IBM Tivoli License Manager files
```

Notes

|	
+— jcr	Java Content Repository files.
|	
+— ldap	LDAP configuration files.
|	
+— log (r/w)	WebSphere Portal log files (created during installation).
|	
|	
+— migration (r/w)	Scripts used to assist in migrating from previous releases of WebSphere Portal.
|	
|	
+— odc (mixed)	On-demand client files (./com is r/w; all other subdirectories are r/o).
|	
|	
+— portletscripts (r/w)	XML files for installing portlets individually.
|	
+— pzn (r/w)	Personalization runtime and resources (This component copies files during installation to a WebSphere subdirectory).
|	
|	
|	
+— shared	
| |	
| +— ext (r/o)	Contains collaborator.jar.
| |	
| +— app (r/w)	WebSphere Portal runtime JARs, TLDs, and other resources.
| |	
| |	
| +— config r/w)	Portal configuration files.
| | |	
| | +- services	Portal services configuration files (*.properties).
| |	
| +— nls (r/o)	WebSphere Portal NLS files.
|	
+— service	Contains fix packages.
|	
+— version (r/w)	Version information for various components.
|	
+— wcm	Source Web application files for Web Content Management
|	
+— wmm (mixed)	Member Manager configuration, including attributes of portal users (code will be r/o).

WebSphere Application Server Installation Directory

Throughout this documentation, the install location for WebSphere Application Server is noted aswas_root.

Notes

The following table shows the default WebSphere Application Server installation location if it is not otherwise specified during installation:

Default location of WebSphere Application Server

Operating system	Location
z/OS	/WebSphere/V5R1M0/AppServer

WebSphere Application Server Profile Directory

Throughout this documentation, the location for the WebSphere Application Server profiles is noted aswa profile root.

The following table shows the default WebSphere Application Server location for profiles if it is not otherwise specified during installation.

Default location of WebSphere Application Server profiles

Operating system	Location
z/OS	/WebSphere/V5R1M0/AppServer

WebSphere Application Server Configuration Directory

Throughout this documentation, the location for the WebSphere Application Server configuration files is noted as was_config_root.

The following table shows the default WebSphere Application Server location if it is not otherwise specified during installation.

Default location of WebSphere Application Server configuration files

Operating system	Location
z/OS	/WebSphere/V5R1M0/AppServer/config

wps.war directory

The WebSphere Portal enterprise application is installed to the following location within WebSphere Application Server's path:

was_root/installedApps/cell_name/wps.ear/wps.war

The WAR file directory structure for the WebSphere Portal enterprise application contains the following resources.

```
wps.war
 |
 +— c2a                Cooperative portlet resources
 |
 +— doc                Portal-level help and readme
 |
 +— DocEditor          JSPs for the Rich Text Editor
 |
 +— dtd                XML DTDs and schema definitions
 |
 +— html               License and privacy HTML files for the portal.
 |
```

Notes

+— images	Common images for the portal
|	
+— menu	Resources for MenuService applet (Collaborative Components API)
|	
+— META-INF	Standard Java MANIFEST.MF for the portal Web application
|	
+— peopleawareness	Resources for the people awareness features
|	
+— screens	Screen JSPs for the portal
| |	
| +— *markup_name*	Subdirectory for each markup type
|	
|— skins	Skin JSPs for the portal
| |	
| +— *markup_name*	Subdirectory for each markup type
|	
|— themes	Theme JSPs for the portal
| |	
| +— *markup_name*	Subdirectory for each markup type
|	
+— virtualportal	XMLAccess file for setting up a virtual portal
|	
+— WEB-INF	Protected resources for the portal Web application
|	
+— wts	JavaScript resources for Transcoding Technology

The following directories contain resources for customization. Resources for all other directories in the portal Web application directory structure must not be modified.

- /doc
- /html
- /images
- /screens
- /skins
- /themes
- /virtualportal

6.6.1 Directories for Languages

The following shows the languages supported by WebSphere Portal and the directories used for storing locale-specific resources. These directories are used in portlet Web application directories and in theWebSphere Portal enterprise application (themes, skins, and other Web application resources).

Notes

Language (locale)	Directory		Language (locale)	Directory		Language (locale)	Directory
Arabic	/ar		Hungarian	/hu		Brazilian Portuguese	/pt_BR
Czech	/cs		Italian	/it		Romanian	/ro
Danish	/da		Hebrew	/iw		Russian	/ru
German	/de		Japanese	/ja		Swedish	/sv
English	/en		Korean	/ko		Thai	/th
Greek	/el		Dutch	/nl		Turkish	/tr
Spanish	/es		Norwegian	/no		Ukrainian	/uk
Finnish	/fi		Polish	/pl		Simplified Chinese	/zh
French	/fr		Portuguese	/pt		Traditional Chinese	/zh_TW

6.7 File Sharing

In the previous sections, we explored the motivation for file sharing and some of the difficulties involved in allowing users to share files. Such file sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. First is the topic of multiple users and the sharing methods possible. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems. Finally, there can be several interpretations of conflicting actions occurring on shared files. For instance, if multiple users are writing to the file, should all the writes be allowed to occur, or should the operating system protect the user actions from each other.

6.7.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system either can allow a user to access the files of other users by default, or it may require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered below. To implement sharing and protection, the system must maintain more file and directory attributes than on a single-user system. Although there have been many approaches to this topic historically, most systems have evolved to the concepts of file/directory owner (or user) and group. The owner is the user who may change attributes, grant access, and has the most control over the file or directory. The group attribute of a file is used to define a subset of users who may share access to the file. For example, the owner of a file on a UNIX system may issue all operations on a file, while members of the file's group may execute one subset of those operations, and all other users may execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner. More details on permission attributes are included in the next section. Most systems implement owner attributes by managing a list of user names and associated **user identifiers (user IDS)**. In Windows NT parlance, this is a **Security ID (SID)**. These numerical IDS are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID

Notes

is associated with all of the user's processes and threads. When they need to be user readable, they are translated back to the user name via the user name list. Likewise, group functionality can be implemented as a system-wide list of group names and **group identifiers**. Every user can be in one or more groups, depending upon operating system design decisions. The user's group IDS are also included in every associated process and thread. The owner and group IDS of a given file or directory are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared to the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDS can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation, and allows or denies it. The user information within a process can be used for other purposes as well. One process may attempt to interact with another process, and user information can dictate the result, based on the design of the operating system. For example, a process may attempt to terminate, background, or lower the priority of another process. If the owner of each process is the same, then the command may succeed, or else it may be denied. It may also be allowed to succeed if it is owned by the privileged user. Many systems have multiple local file systems, including partitions of a single disk or multiple partitions on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted.


Did u know?

The user information within a process can be used for other purposes as well. One process may attempt to interact with another process, and user information can dictate the result, based on the design of the operating system. For example, a process may attempt to terminate, background, or lower the priority of another process. If the owner of each process is the same, then the command may succeed, or else it may be denied.

6.7.2 Remote File Systems

The advent of networks allowed communication between remote computers. Networking allows the sharing of resources spread within a campus or even around the world. One obvious resource to share is data, in the form of files. Through the evolution of network and file technology, file-sharing methods have changed. In the first implemented method, users manually transfer files between machines via programs like ftp. The second major method is a **distributed file system (DFS)** in which remote directories are visible from the local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files. ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involve a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, which we describe in this section.


Task How to use operating system concept of Remote File Systems?

6.7.3 The Client-Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the server, and the machine wanting access to the files is the client. The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource and exactly which clients. Files are usually specified on a partition or subdirectory level. A server can serve multiple clients, and a client can use multiple servers,

Notes

depending on the implementation details of a given client-server facility. Client identification is more difficult. Clients can be specified by their network name or other identifier, such as IP address, but these can be spoofed (or imitated). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access. More secure solutions include secure authentication of the client to the server via encrypted keys. Unfortunately, with security comes many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and secure key exchanges (intercepted keys could again allow unauthorized client access). These problems are difficult enough that, most commonly, unsecure authentication methods are used. In the case of UNIX and its network file system (NFS), authentication is via the client networking information, by default. In this scheme, the user IDS must match on the client and server. If not, the server will be unable to determine access rights to files. Consider the example of a user who has the ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file, rather than the real user ID of 2000. Access would be granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. The NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to other NFS clients and a client of other NFS servers. Once the remote file system is mounted, file operation requests are sent on the behalf of the user, across the network, to the server, via the DFS protocol. Typically, a file open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then may perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file system mount, or may have different semantics.

6.7.4 Distributed Information Systems

To ease the management of client-server services, **distributed information systems**, also known as **distributed naming services**, have been devised to provide a unified access to the information needed for remote computing. **Domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNS was invented and became widespread, files containing the same information were sent via email or f t p between all networked hosts. This methodology was not scalable. Other distributed information systems provide user narne/password/user ID/group ID space for a distributed facility. UNIX systems have had a wide variety of distributed information methods. Sun Microsystems introduced yellozv pages (since renamed to **network information service (NIS)**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in clear text) and identifying hosts by IP address. Sun's NIS+ is a much more secure replacement for NIS, but is also much more complicated and has not been widely adopted. In the case of Microsoft networks **(CIFS)**, network information is used in conjunction with user authentication (user name and password) to create a **network login** that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match between the machines (as with NFS). Microsoft uses two distributed naming structures to provide a single namespace for users. The older naming technology is **domains**. The newer technology, available in Windows 2000 and beyond, is active **directory**. Once established, the distributed-naming facility is used by all clients and servers to authenticate users. The industry is moving toward **lightweight directory-access protocol (LDAP)** as a secure, distributed naming mechanism. In fact, active directory is based on LDAP. Sun Microsystems' Solaris 8 allows LDAP to be used for user authentication as well as system-wide retrieval of information such as available printers. If the convergence of the use of LDAP succeeds, then one distributed LDAP directory will be used by an organization to store all user and resource information for all computers within that organization. The result would

Notes

be **secure single sign-on** for users, who would enter their authentication information once for access to all computers within the organization. It would also ease systems-administration efforts by combining, into one location, information that is currently scattered in various files on each system or in differing distributed information services.

6.7.5 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk management information (collectively called **metadata**), disk-controller failure, cable failure, or host adapter failure. User or systems-administrator failure can also cause files to be lost, or entire directories or partitions to be deleted. Many of these failures would cause a host to crash and an error condition to be displayed, and require human intervention to repair. Some failures do not cause loss of data or loss of availability of data.

Redundant arrays of inexpensive disks (RAID) can prevent the loss of a disk from resulting in the loss of data. Remote file systems have more failure modes. By nature of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between the two hosts. This could be due to hardware failure or misconfiguration, or networking implementation issues at any of the involved hosts. Although some networks have built-in resiliency, including multiple paths between each host, many do not. Any single failure could interrupt the flow of DFS commands. Consider a client in the midst of using a remote file system. It has remote file systems mounted and may have files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of that server, such that the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client to act as it would in the case of a loss of a local file system. Rather, the system could either terminate all operations to the lost server, or delay operations until the server is again reachable. This failure semantics is defined and implemented as part of the remote file system protocol. Termination of all operations can result in users losing data, and patience. Most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again. For this kind of recovery from failure, some kind of state information may be maintained on both the client and server. If the server has crashed, but must recognize that it had exported file systems, remotely mounted them, and opened certain files, NFS takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation on a file. Likewise, it does not track which clients have its exported partitions mounted, again assuming that if a request comes it, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it makes it unsecure. For example, forged read or write requests could be allowed by an NFS server event.

6.8 Consistency Semantics

Consistency semantics is an important criterion for evaluating any file system that supports file sharing. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users. The semantics are typically implemented as code with the file system. Consistency semantics are directly related to the process synchronization algorithms. However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could

Notes

involve several network communications or several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew File System. For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the **open** and **close** operations. The series of accesses between the **open** and **close** operations is a **file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

6.8.1 UNIX Semantics

The UNIX file system uses the following consistency semantics—Writes to an open file by a user are visible immediately to other users that have this file open at the same time. One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin. In the UNIx semantics a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image results in user processes being delayed.

6.8.2 Session Semantics

The Andrew file system (AFS) uses the following consistency semantics: Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes. According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their image of the file, without delay. Almost no constraints are enforced on scheduling accesses.

6.8.3 Immutable-Shared-Files Semantics

A unique approach is that of immutable shared files. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: Its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed, rather than the file being a container for variable information. The implementation of these semantics in a distributed system is simple, because the sharing is disciplined (read-only).

6.9 Protection Mechanisms

The concept of multiprogramming introduces the sharing resources among users. This sharing involves Memory, I/O devices, Programs and Data. The ability to share these resources introduces the need for protection. An OS may offer protection along the following Spectrum: No Protection: This is appropriate when sensitive procedures are being run at separate times Isolation. This approach implies that each process operates separately from other processes, with no sharing. Each process has its own address space, files, and other objects Share all or Share nothing: In this method, the owner of an object declares it to be public or private, in the other words, only the owner's processes may access the object. Share via access limitation—The OS checks the permissibility of each access by a specific user to specific object; the OS therefore acts as a guard between users and objects, ensuring that only authorized accesses occur. Share via dynamic capabilities: This extends the concept of access control to allow dynamic creation of sharing rights for objects. Limit use of an object—This form of protection limits not just access to an object but the use to which that object may be put. A given OS may provide different degree of protection for different objects, users and applications The OS needs to balance the need to allow sharing, with the need to protect the resources of individual users.

Notes

6.9.1 Protection of Memory

In a multiprogramming environment, protection of main memory is essential. The concern here is not just security but the correct functioning of the various processes that are active. The separation of the memory space of various processes is easily accomplished with a virtual-memory scheme. Segmentation or Paging, or two in combination, provides an effective tools of managing main memory. If complete isolation is sought, then the OS must simply ensure that each segment or page accessible only by the process to which it is assigned. This is accomplished by requiring that there be no duplicate entries in page and/or segment tables. If sharing is to be allowed then the same segment or page may appear in more than one table. Segmentation specially lends itself to the implementation of protection and sharing policies. Because each segment table entry includes a length as well as a base address. A program can not access a main memory location beyond the limit of a segment. To achieve sharing, it is possible for a segment to be referenced in the segment tables of more than on process. In the paging system, the page structure of the programs and data is not visible to the programmer. The measures taken to control access in a data processing systems fall into two categories:

1. User-oriented
2. Data-oriented


Did u know?

A program can not access a main memory location beyond the limit of a segment. To achieve sharing, it is possible for a segment to be referenced in the segment tables of more than on process.

6.9.2 User-oriented Access Control

User control of access is sometimes referred to as Authentication. The most common technique for user access control on a shared system or server is the user log, which requires ID and Password. User access control in distributed environment can be either centralized or decentralized in a centralized approach network provides a log on service, determining who is allowed to use the network and to whom the user is allowed to connect. Decentralized user access control treats the network as a transport communication link, and the destination host carries out the usual log on procedure. In many networks, two levels of access control may be used. Data-Oriented Access Control Following successful log on , the user has been granted access to one or set of hosts and applications. At this time we need Data access control. In this regard real world operating system protection models fall basically into one of two types:

1. Mandatory Access Controls (MAC)
2. Discretionary Access Controls (DAC)

In computer security passive resources are called objects and active entities that utilize the resources are called subjects. Typical objects include—files, directories, memory, printers and typical subjects include: users, processes. The roles depend on situation—for example, a process can request access to some resource (act as a subject) and later be a target of access request (act as an object).

In Mandatory access controls, also called multilevel access control, Objects (information) are classified on hierarchical levels of security sensitivity (typically, top secrets, secret, confidential). Subjects (Users) are assigned their security clearance. Access of a subject to an object is granted or denied depending on the relation between the clearance of the subject and the security classification of the object. Lattice model and Bell-LaPadula model are based on MAC Discretionary access controls (DAC). Each object has its unique owner. The owner exercises its discretion over the assignment of access permissions. Lampson introduced the access matrix model for DAC. The core of this model is a matrix whose rows are indexed by subjects and columns by objects.

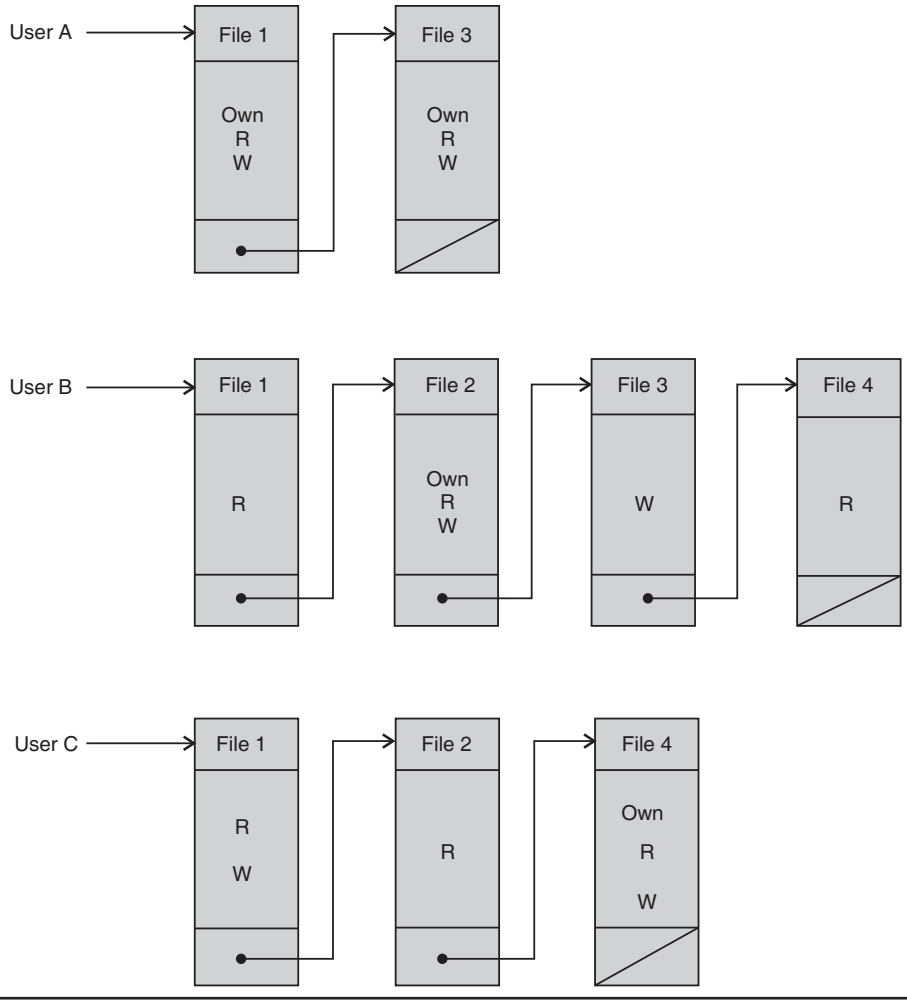
Notes

Figure 6.5: Access Matrix Model

	doc_1	password	progr_1
Alice	rw	r	X
Bob	r	r	–
Ronald	rw	rw	rwx

In real systems, however, access control matrices are not very practical, because the matrix is usually sparse and there is a lot of redundancy and new subjects and objects can be added or removed easily, but the centralized matrix could become a bottleneck. The matrix may be decomposed by columns, yielding Access Control List (ACL). Thus for each object, an ACL details users and their permitted access rights. ACL may contain a default or public entry. Decomposition by rows yields capability tickets. A capability ticket specifies authorized objects and operations for a user. Each user has a number of tickets and may be authorized to lend or give them to others because tickets may be dispersed around the system, they present a greater security problem than ACL. To accomplish this problem, OS hold all tickets on behalf of the users. These tickets would have to be held in a region of memory inaccessible to users.

Figure 6.6: Access Control List



Notes

6.9.3 Protection Based on an OS Mode

Most processor supports at least two modes of operations—1. User mode and 2. Kernel mode. The reason for using two modes should be clear. It is necessary to protect the OS and the key OS table such as process control blocks, from interference by user programs in kernel mode. This level of control is not necessary for user mode.

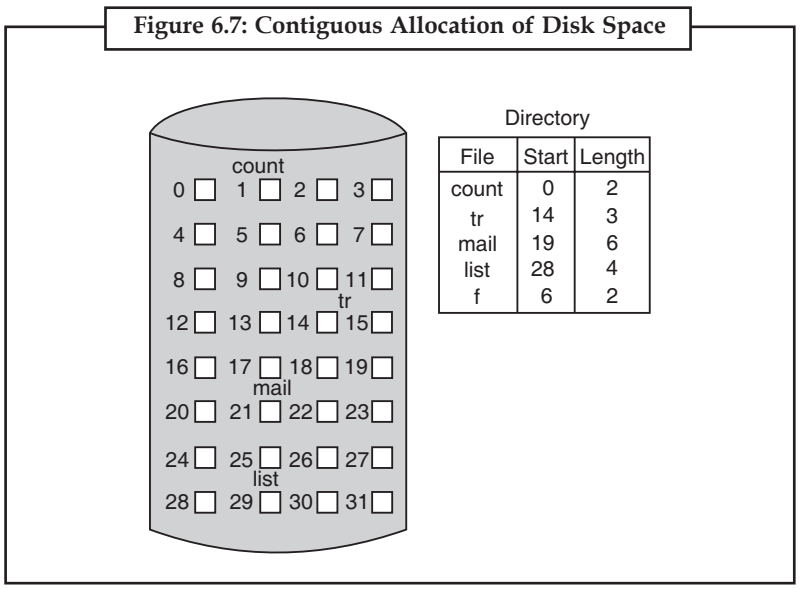
6.10 Allocation Methods

The direct-access nature of disks allows us exibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use—contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system uses one method for all files within a file system type.

6.10.1 Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blockson the disk. Disk addresses dene a linear ordering on the disk. With thisordering, assuming that only one job is accessing the disk, accessing block b + 1 after block b normally requires no head movement. When head movementis needed (from the last sector of one cylinder to the last sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance. Contiguous allocation of a file is defined by the disk address and length (inblock units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n – 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Figure 6.7: Contiguous Allocation of Disk Space



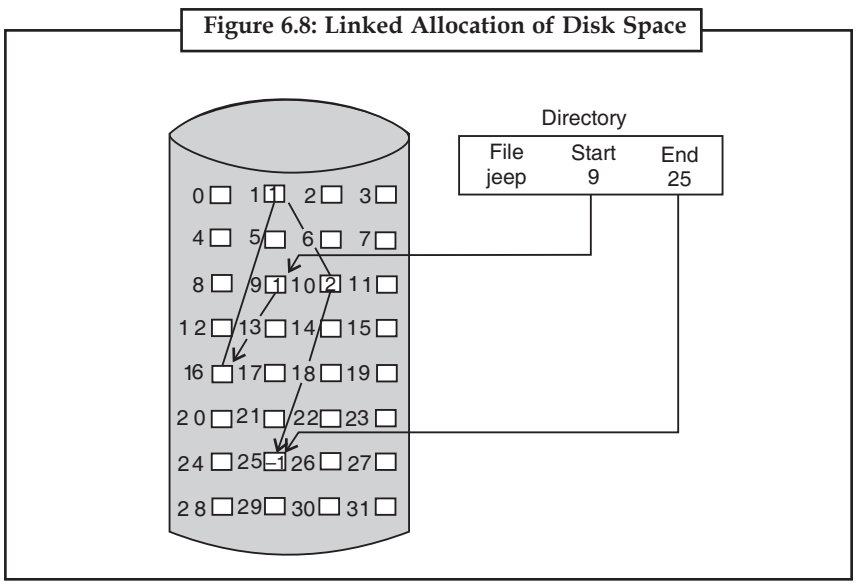
Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous

Notes

allocation. Contiguous allocation has some problems, however. One diffculty is finding space for a new file. Any management system can be used, but some are slower than others. First and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster. All these algorithms suffer from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem. Some older PC systems used contiguous allocation on floppy disks.

Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally over estimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. However, the user need never be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly. Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation. To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; and then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in efficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated. The commercial Veritas file system uses extents to optimize performance. It is a high-performance replacement for the standard UNIX UFS.

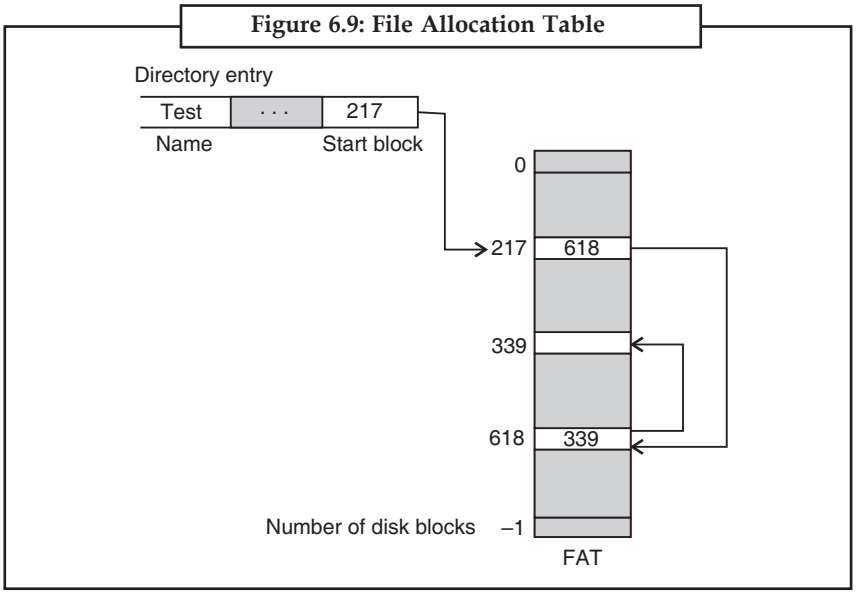
Figure 6.8: Linked Allocation of Disk Space



Notes

Linked allocation does have disadvantages. However, the major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block. Each access to a pointer requires a disk.

Figure 6.9: File Allocation Table



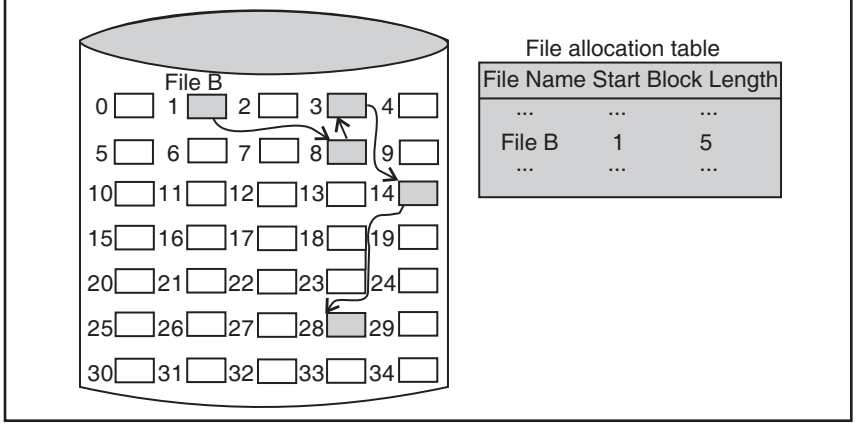
An important variation on linked allocation is the use of a file-allocation table (FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. For a file consisting of disk blocks 217, 618, and 339. The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

6.10.2 File Allocation Methods—Chained

The opposite extreme to contiguous allocation is chained allocation. Here, the blocks allocated to a file form a linked list (or chain), and as a file's length is extended (by appending to the file), a new block is allocated and linked to the last block in the file.

Notes

Figure 6.10: Chained Allocation



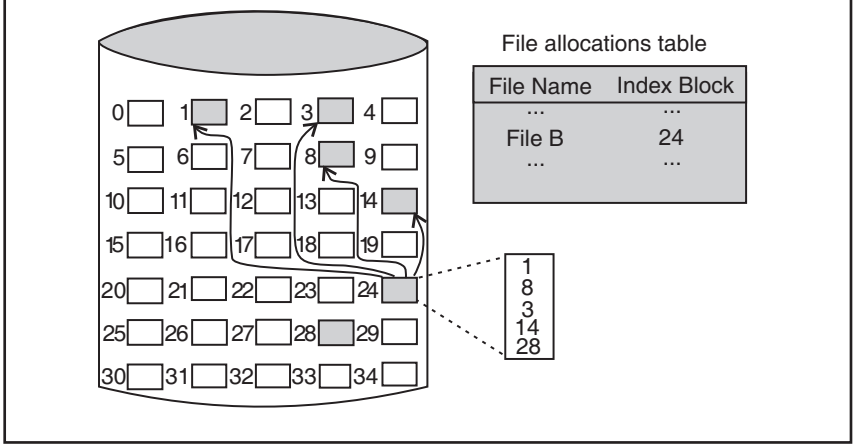
A small "pointer" of typically 32 or 64 bits is allocated within each file block to indicate the next block in the chain. Thus seeking within a file requires a read of each block to follow the pointers.

New blocks may be allocated from any free block on the disk. In particular, a file's blocks need no longer be contiguous.

6.10.3 File Allocation Methods—Indexed

The file allocation method of choice in both Unix and Windows is the indexed allocation method. This method was championed by the Multics operating system in 1966.

Figure 6.11: Indexed Allocation with Block Portions



The file-allocation table contains a multi-level index for each file. Indirection blocks are introduced each time the total number of blocks "overflows" the previous index allocation. Typically, the indices are neither stored with the file-allocation table nor with the file, and are retained in memory when the file is opened.



Caution

Before using memory allocation must be know index concept of memory.

Notes

6.11 Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we shall discuss.

6.11.1 Bit Vector

Frequently, the free-space list is implemented as a bit **map** or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be

0011110011111100011000000011100000 . . .

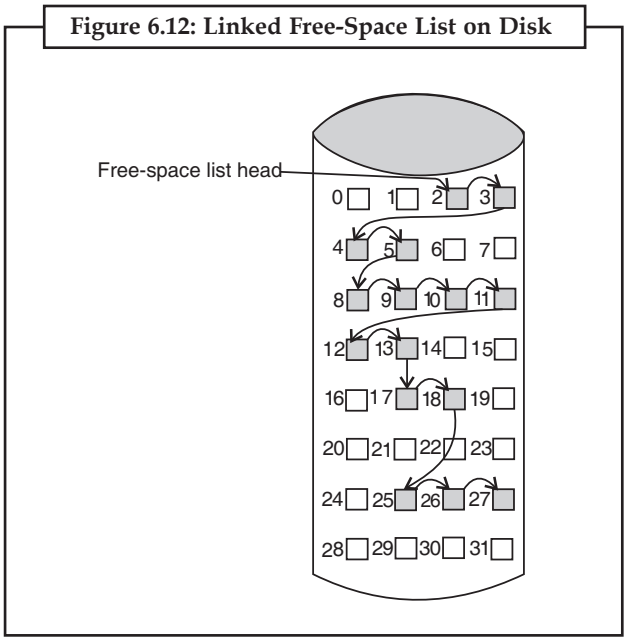
The main advantage of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. For example, the Intel family starting with the 80386 and the Motorola family starting with the 68020 (processors that have powered PCs and Macintosh systems, respectively) have instructions that return the offset in a word of the first bit with the value 1. In fact, the Apple Macintosh operating system uses the bit-vector method to allocate disk space. To find the first free block, the Macintosh operating system checks sequentially each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is (number of bits per word) x (number of 0-value words) + offset of first 1 bit. Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks, such as on microcomputers, but not for larger ones. A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks. Clustering the blocks in groups of four reduces this number to over 83 KB per disk.

6.11.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. we would keep a pointerto block 2, as the first free block. Block 2 would contain a pointer to block. **3**, which would point to block 4, which would point to block 5, which would point to block 8, However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial 1/0 time. Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

Notes

Figure 6.12: Linked Free-Space List on Disk



6.11.3 Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

6.11.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than

6.12 Directory Implementation

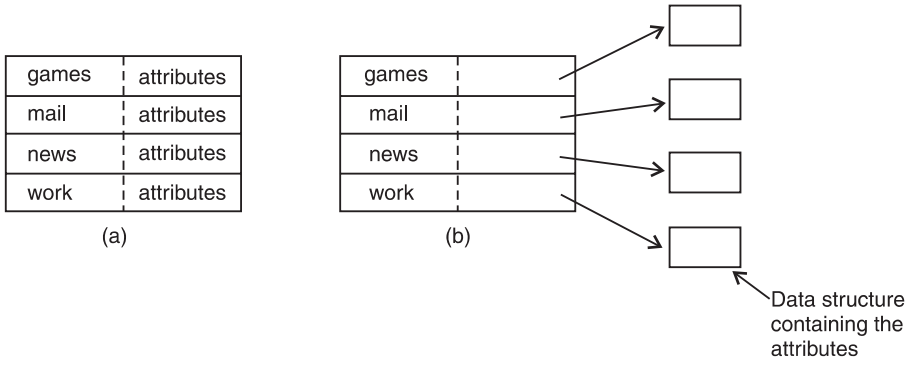
To keep track of files, file systems normally have directories or folders, which, in many systems, are themselves files. In this section, we will discuss directories, their organization, their properties, and the operations that can be performed on them.

6.12.1 Simple Directories

A directory typically contains a number of entries, one per file. One possibility is shown in Fig. 6.13(a), in which each entry contains the file name, the file attributes, and the disk addresses where the data are stored. Another possibility is shown in Fig. 6.13(b). Here a directory entry holds the file name and a pointer to another data structure where the attributes and disk addresses are found. Both of these systems are commonly used.

Notes

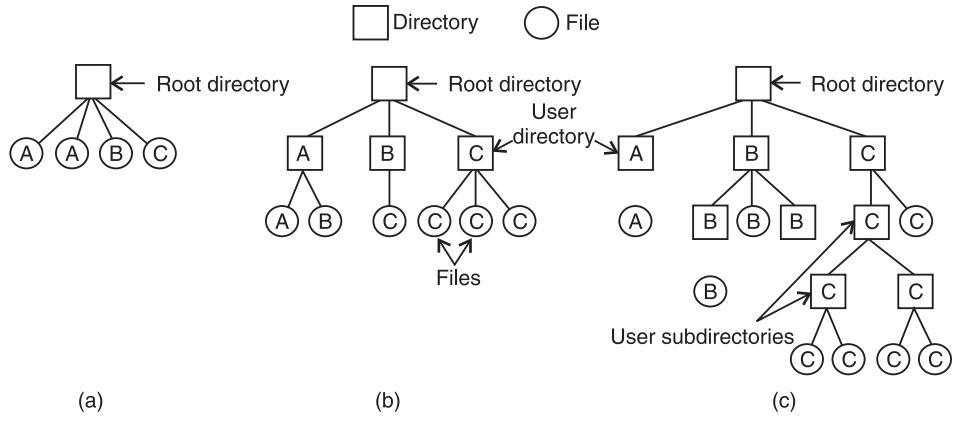
Figure 6.13: (*a*) Attributes in the Directory Entry and (*b*) Attributes Elsewhere



When a file is opened, the operating system searches its directory until it finds the name of the file to be opened. It then extracts the attributes and disk addresses, either directly from the directory entry or from the data structure pointed to, and puts them in a table in main memory. All subsequent references to the file use the information in main memory.

The number of directories varies from system to system. The simplest form of directory system is a single directory containing all files for all users, as illustrated in Fig. 6.14(a). On early personal computers, this single-directory system was common, in part because there was only one user.

Figure 6.14: Three File System Designs: (*a*) Single Directory Shared by All Users, (*b*) One Directory Per User, (*c*) Arbitrary Tree Per User. The Letters Indicate the Directory or File's Owner



The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user A creates a file called mailbox, and then later user B also creates a file called mailbox, B's file will overwrite A's file. Consequently, this scheme is not used on multi-user systems any more, but could be used on a small embedded system, for example, a handheld personal digital assistant or a cellular telephone.

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system of Fig. 6.14 (b).

Notes

This design could be used, for example, on a multi-user computer or on a simple network of personal computers that shared a common file server over a local area network. Implicit in this design is that when a user tries to open a file, the operating system knows which user it is in order to know which directory to search. As a consequence, some kind of login procedure is needed, in which the user specifies a login name or identification, something not required with a single-level directory system. When this system is implemented in its most basic form, users can only access files in their own directories.

6.12.2 Hierarchical Directory Systems

The two-level hierarchy eliminates file name conflicts between users. But another problem is that users with many files may want to group them in smaller subgroups, for instance a professor might want to separate handouts for a class from drafts of chapters of a new textbook. What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 6.14 (c). Here, the directories A, B, and C contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

The ability to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason nearly all modern PC and server file systems are organized this way.

However, as we have pointed out before, history often repeats itself with new technologies. Digital cameras have to record their images somewhere, usually on a flash memory card. The very first digital cameras had a single directory and named the files DSC0001.JPG, DSC0002. JPG, etc. However, it did not take very long for camera manufacturers to build file systems with multiple directories, as in Fig. 6.14 b). What difference does it make that none of the camera owners understand how to use multiple directories, and probably could not conceive of any use for this feature even if they did understand it? It is only (embedded) software, after all, and thus costs the camera manufacturer next to nothing to provide. Can digital cameras with full-blown hierarchical file systems, multiple login names, and 255-character file names be far behind?

6.12.3 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an absolute path name consisting of the path from the root directory to the file. As an example, the path /usr/ast/mailbox means that the root directory contains a subdirectory usr/, which in turn contains a subdirectory ast/, which contains the file mailbox. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \ . Thus the same path name would be written as follows in these two systems:

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the relative path name. This is used in conjunction with the concept of the working directory (also called the current directory). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox. In other words, the UNIX command cp /usr/ast/mailbox /usr/ast/mailbox.bak nand

Notes

the command cp mailbox mailbox.bak do exactly the same thing if the working directory is /usr/ast/. The relative form is often more convenient, but it does the same thing as the absolute form.

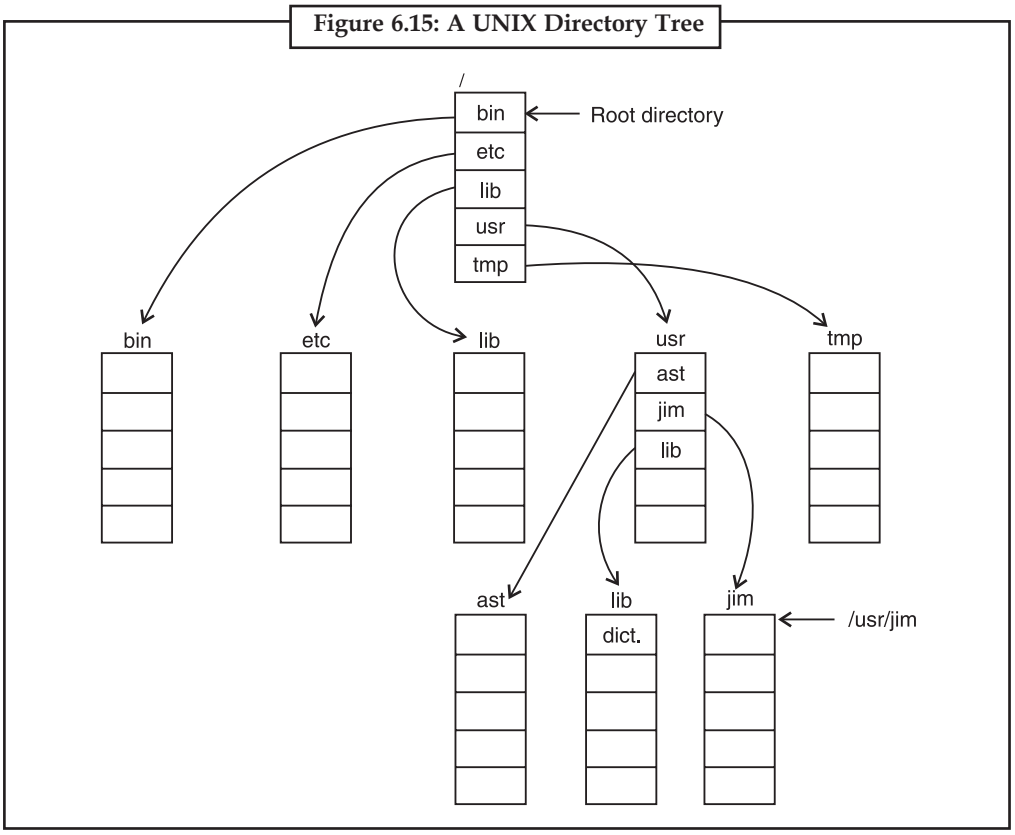
Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read /user/lib/dictionary to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is of course, if the spelling checker needs a large number of files from /user/lib/, an alternative approach is for it to issue a system call to change its working directory to /user/lib/, and then use just dictionary as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Each process has its own working directory, so when a process changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way it is always perfectly safe for a process to change its working directory whenever that is convenient on the other hand, if a library procedure changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 6.15 A certain process has /usr/ast/ as its working directory. It can use .. to go up the tree. For example, it can copy the file /usr/lib/dictionary to its own directory using the command:

cp ../lib/dictionary.

Figure 6.15: A UNIX Directory Tree



Notes

The first path instructs the system to go upward (to the usr directory), then to go down to the directory lib/ to find the file dictionary.

The second argument (dot) names the current directory. When the cp command gets a directory name (including dot) as its last argument, it copies all the files there. Of course, a more normal way to do the copy would be to type cp /usr/lib/dictionary. Here the use of dot saves the user the trouble of typing dictionary a second time.

Nevertheless, typing cp /usr/lib/dictionary dictionary also works fine, as does cp /usr/lib/dictionary /usr/ast/dictionary. All of these do exactly the same thing.



Did u know?

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read /usr/lib/dictionary to do its work.

6.12.4 Directory Operations

The system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the mkdir program).
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir. When a directory has been read, it should be closed to free up internal table space.
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a hard link.
8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

Notes



Case Study

A Study of Scalability and Performance of Solaris Zones

This thesis presents a quantitative evaluation of an operating system virtualization technology known as Solaris Containers or Solaris Zones, with a special emphasis on measuring the influence of a security technology known as Solaris Trusted Extensions, Solaris Zones is an operating system-level (OS-level) virtualization technology embedded in the Solaris OS that primarily provides containment of processes within the abstraction of a complete operating system environment. Solaris Trusted Extensions present a specific configuration of the Solaris operating system that is designed to offer multi-level security functionality.

Firstly, we examine the scalability of the OS with respect to an increasing number of zones. Secondly, we evaluate the performance of zones in three scenarios. In the first scenario we measure as a baseline—the performance of Solaris Zones on a 2-CPU core machine in the standard configuration that is distributed as part of the Solaris OS. In the second scenario we investigate the influence of the number of CPU cores. In the third scenario we evaluate the performance in the presence of a security configuration known as Solaris Trusted Extensions. To evaluate performance, we calculate a number of metrics using the AIM benchmark. We calculate these benchmarks for the global zone, a non-global zone, and increasing numbers of concurrently running non-global zones. We aggregate the results of the latter to compare aggregate system performance against single zone performance.

The results of this study demonstrate the scalability and performance impact of Solaris Zones in the Solaris OS. On our chosen hardware platform, Solaris Zones scales to about 110 zones within a short creation time (i.e., less than 13 minutes per zone for installation, configuration, and boot.) As the number of zones increases, the measured overhead of virtualization shows less than 2% of performance decrease for most measured benchmarks, with one exception: the benchmarks for memory and process management show that performance decreases of 5-12% (depending on the sub-benchmark) are typical. When evaluating the Trusted Extensions-based security configuration, additional small performance penalties were measured in the areas of Disk/Filesystem I/O and Inter Process Communication. Most benchmarks show that aggregate system performance is higher when distributing system load across multiple zones compared to running the same load in a single zone.

Questions:

1. What is the use of Solaris Zones in operating systems?
2. Explain the impact of Solaris Zones in the Solaris OS.

Self Assessment

Multiple choice questions:

6. Computers process data into information by working exclusively with:
   (*a*) multimedia (*b*) words
   (*c*) characters (*d*) numbers
7. The components that process data are located in the:
   (*a*) input devices (*b*) output devices
   (*c*) system unit (*d*) storage component

Notes

True or False:

8. Direct access does not allow a user to partitioned the read/write mark before reading or writing.
9. A disk is typically position also known as slices and minidisks.

6.13 Summary

- A file is a collection of related information defined by its creator.
- Computer can store files on the disk (secondary storage), which provide long term storage. A file can be managed in operating systems in three ways; from within a program, by using MY computer, and by using Windows Explorer. A file is an abstract data type, a "thing", which is defined and implemented by the operating system.
- The main task of the operating system is to map the logical file concept onto physical storage devices, such as disks or tapes. A file can be accessed directly or sequentially.
- A disk is typically partitioned, also known as slices and minidisks.
- The device directory or volume table of contents records and maintains the file properties such as name, size, type and location.
- WebSphere Portal and the directories used for storing locale-specific resources is also been explained.
- File sharing is also a very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.
- Consistency semantics is an important criterion for evaluating any file system that supports file sharing. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. Allocation methods are also a required feature for file management.

6.14 Keywords

Domain Name System (DNS): A system for converting host names and domain names into IP addresses on the Internet or on local networks that use the TCP/IP protocol. For example, when a Web site address is given to the DNS either by typing a URL in a browser or behind the scenes from one application to another, DNS servers return the IP address of the server associated with that name.

Network Information Service (NIS): A naming service from Sun that allows resources to be easily added, deleted or relocated. Formerly known as Yellow Pages, NIS is a de facto Unix standard. NIS+ is a redesigned NIS for Solaris 2.0 products. The combination of TCP/IP, NFS and NIS comprises the primary networking components of Unix.

Distributed File System (DFS): Is a set of client and server services that allow an organization using Microsoft Windows servers to organize many distributed SMB file shares into a distributed file system. DFS provides location transparency and redundancy to improve data availability in the face of failure or heavy load by allowing shares in multiple different locations to be logically grouped under one folder, or DFS root.

Notes

Anonymous Access: The most common Web site access control method, allows anyone to visit the public areas of your Web sites.

Free-Space List: A list of unoccupied areas of memory in main or backing store. It is a special case of an available list.



Lab Exercise

1. Assume 4 jobs arrive to a batch system at 10:00 am in the morning precisely at the following moments:

   Job 1: 10:00am (requires 4 minutes of CPU)

   Job 2: 10:10am (requires 3 minutes of CPU)

   Job 3: 10:15am (requires 2 minutes of CPU)

   Job 4: 10:20am (requires 1 minute of CPU)

   When does each job completes, If the probability of I/O is 0.8?
2. C Program For File Operations.
3. C Program For File Copy and Move.

6.15 Review Questions

1. What is a file?
2. What are the typical operations performed on files?
3. What are File Control Blocks?
4. What are file types?
5. How is free space managed?
6. Consider a file system where a file can be deleted and its disk space Reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?
7. What are the advantages and disadvantages of a system providing mandatory locks instead of providing advisory locks whose usage is left to the users' discretion?
8. What are the advantages and disadvantages of recording the name of the creating program with the file's attributes (as is done in the Macintosh Operating System)?
9. If the operating system were to know that a certain application is going to access the file data in a sequential manner, how could it exploit this information to improve performance?
10. Give an example of an application that could benefit from operating system support for random access to indexed files.
11. Discuss the merits and demerits of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).
12. Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a different set of failure semantics from that associated with local file systems.
13. Give an example of a Directory implementation that could benefit from operating system.

Notes

14. Discuss the advantages and disadvantages of access matrix.
15. What is Hierarchical Directory Systems.
16. What does Contiguous Allocation requires?

Answers to Self Assessment

1. (*b*) 2. (*a*) 3. logic 4. ctrl 5. rename
6. (*d*) 7. (*c*) 8. False 9. True

6.16 Further Readings



Books

Introduction to Operating Design and Implementation, by Michael Kifer, Scoott A. Smolka.

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes.

Operating Systems, by Stuart E. Madnick, John J. Donovan.



Online link

wiley.com/coolege.silberschatz

Unit 7: Secondary Storage Structure

Notes

CONTENTS

Objectives

Introduction

7.1 The Benefits of Secondary Storage

7.2 Disk Structure

7.3 Disk Scheduling

7.3.1 Shortest Seek Time First (SSTF)

7.3.2 Scan

7.3.3 Cscan

7.4 Selecting a Disk Scheduling Algorithm

7.4.1 Scheduling Fixed-head Devices

7.5 Disk Management

7.5.1 Disk Formatting

7.5.2 Boot Block

7.5.3 Bad Blocks

7.6 Swap Space Management

7.6.1 Pseudo-Swap Space

7.6.2 Physical Swap Space

7.6.3 Swap Space Parameters

7.6.4 Swap Space Global Variables

7.6.5 Swap Space Values

7.6.6 How Swap Space is Prioritized

7.6.7 Three Rules of Swap Space Allocation

7.7 Overview of RAID Structure

7.7.1 Improvement of Reliability via Redundancy

7.7.2 Improvement in Performance via Parallelism

7.8 RAID Levels

7.8.1 Selecting a RAID Level

7.8.2 Extensions

7.9 Summary

7.10 Keywords

7.11 Review Questions

7.12 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Explain meaning of secondary storage structure
- Understand the benefits of secondary storage
- Discuss disk structure
- Explain disk scheduling
- Understand disk management
- Explain swap-space management
- Understand overview of RAID structure
- Discuss improvement of reliability via redundancy
- Explain improvement in performance via parallelism

Introduction

Any non-volatile storage medium that is not directly accessible to the processor. Memory directly accessible to the processor includes main memory, cache and the CPU registers. Secondary storage includes hard drives, magnetic tape, CD-ROM, DVD drives, floppy disks, punch cards and paper tape.

Secondary storage devices are usually accessed via some kind of controller. This contains registers that can be directly accessed by the CPU like main memory ("memory mapped"). Reading and writing these registers can cause the device to perform actions like reading a block of data off a disk or rewinding a tape.

7.1 The Benefits of Secondary Storage

Picture, if you can, how many filing-cabinet drawers would be required to hold the millions of files of, say, tax records kept by the Internal Revenue Service or historical employee records kept by General Motors. The record storage rooms would have to be enormous. Computers, in contrast, permit storage on tape or disk in extremely compressed form. Storage capacity is unquestionably one of the most valuable assets of the computer.

Secondary storage, sometimes called auxiliary storage, is storage separate from the computer itself, where you can store software and data on a semi permanent basis. Secondary storage is necessary because memory, or primary storage, can be used only temporarily. If you are sharing your computer, you must yield memory to someone else after your program runs; if you are not sharing your computer, your programs and data will disappear from memory when you turn off the computer. However, you probably want to store the data you have used or the information you have derived from processing; that is why secondary storage is needed. Furthermore, memory is limited in size, whereas secondary storage media can store as much data as necessary. Keep in mind the characteristics of the memory hierarchy that were described in the section on the CPU and memory.

Notes

Storage	Speed	Capacity	Relative Cost ($)	Permanent
Registers	Fastest	Lowest	Highest	No
RAM	Very Fast	Low/Moderate	High	No
Floppy Disk	Very Slow	Low	Low	Yes
Hard Disk	Moderate	Very High	Very Low	Yes

The benefits of secondary storage can be summarized as follows:

Capacity: Organizations may store the equivalent of a roomful of data on sets of disks that take up less space than a breadbox. A simple diskette for a personal computer holds the equivalent of 500 printed pages, or one book. An optical disk can hold the equivalent of approximately 400 books.

Reliability: Data in secondary storage is basically safe, since secondary storage is physically reliable. Also, it is more difficult for unscrupulous people to tamper with data on disk than data stored on paper in a file cabinet.

Convenience: With the help of a computer, authorized people can locate and access data quickly.

Cost: Together the three previous benefits indicate significant savings in storage costs. It is less expensive to store data on tape or disk (the principal means of secondary storage) than to buy and house filing cabinets. Data that is reliable and safe is less expensive to maintain than data subject to errors. But the greatest savings can be found in the speed and convenience of filing and retrieving data.

These benefits apply to all the various secondary storage devices but, as you will see, some devices are better than others. We begin with a look at the various storage media, including those used for personal computers, and then consider what it takes to get data organized and processed.

7.2 Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks. Thus, tapes are currently used mainly for backup, for storage of infrequently used information, as a medium for transferring information from one system to another, and for storing quantities of data so large that they are impractical as disk systems. Modern disk drives are addressed as large one-dimensional arrays of **logical blocks,** where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to choose a different logical block size, such as 1,024 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost. By using this mapping, we can-at least in theory-convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives. On media that use **constant linear velocity (CLV),**

Notes

the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**. The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.


Did u know?

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

7.3 Disk Scheduling

In multiprogramming systems, many processes may be generating requests for reading and writing disk records. Because these processes often make requests faster than they can be serviced by the moving head disks, waiting queues are build up for each devices. In order to stop unbounded increase in the queue length these pending requests must be examined and serviced in an efficient manner. Disk scheduling involves a careful examination of pending requests to determine the most efficient ways to service the waiting requests terms.

Latency Time: The time it takes for the data block to rotate from its current to just under the read-write head is called latency time.

Seek Time: The time it takes to position the read-write head on the top of the track where data block is stored.

Transfer Time: The time it takes to transfer a block of data from the disk to memory.

These times >> CPU processing time.


Example: *Transfer rate of RL81 (VAX Disk) = 2.2 mb/sec*
Data block size = 512 bytes.
Total block transfer time (Latency+Seek+Transfer) = about 0.1 sec.
CPU will take about 9600 ns (0.0000096 sec) to read this block.


Task Give the detail how to schedule the disk in operating system.

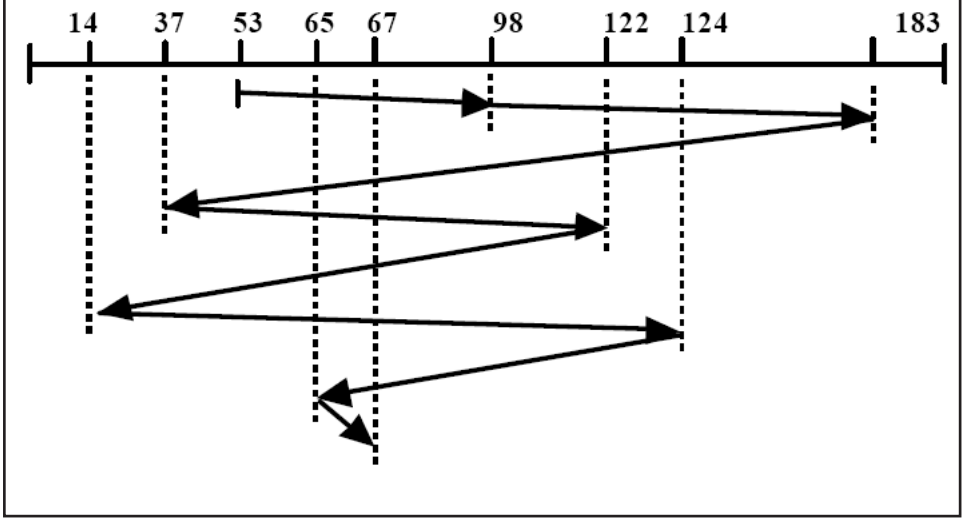
Disk Scheduling Algorithms

First Come First Served: Process the first request then the next and so on.

Notes

Example: Required tracks: 98, 183, 37, 122, 14, 124, 14, 124, 65 and 67. Head starts at: 53

Figure 7.1: Total Tracks Covered: 53 to 98 = 45, 98 to 183 = 85, and so on = 640



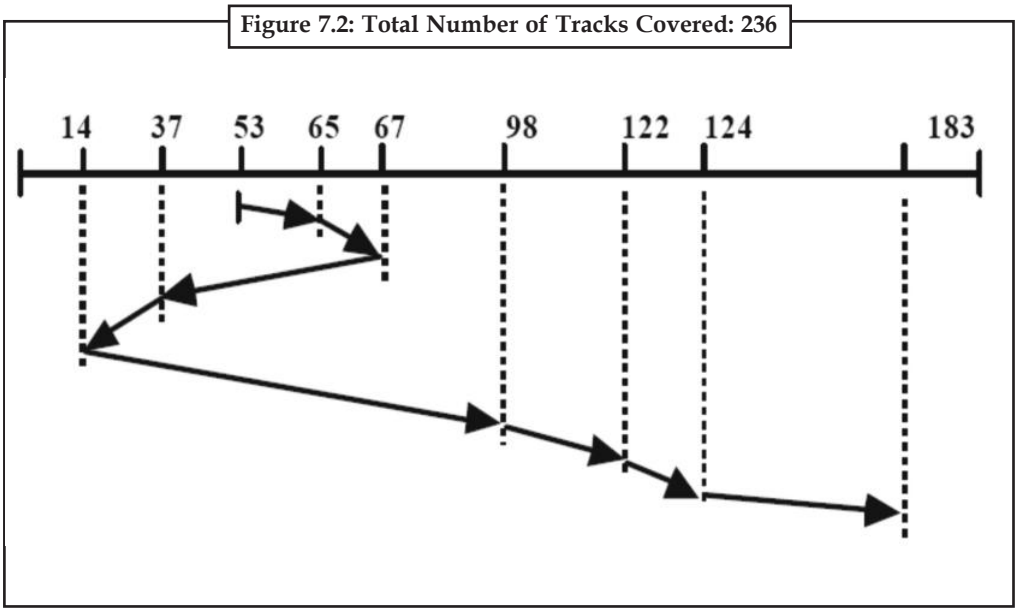
Problems: Wild swing. Several close requests can be serviced together such as 37 and 14, 122 and 124 etc.

7.3.1 Shortest Seek Time First (SSTF)

Service all requests close to the current head position together, before moving the head far away to service another request. This policy is similar to shortest job first.

Example:

Figure 7.2: Total Number of Tracks Covered: 236



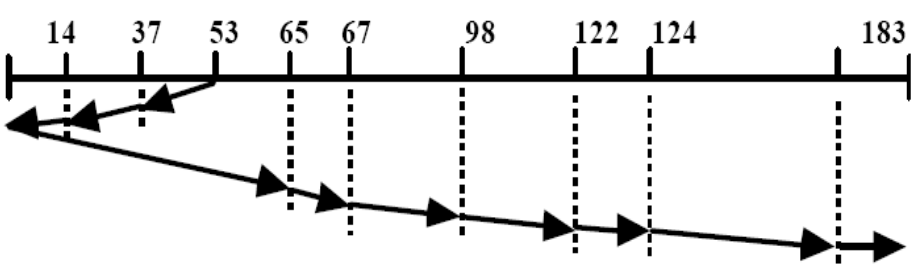
Notes

Problems: May cause starvation to some requests, since requests may arrive at anytime.

7.3.2 Scan

Head starts at one end and moves towards the other end, servicing the requests on its way. At the end the head movement direction is reversed and servicing continues. Example, head position at 53 movement towards zero, servicing 37, 14 and goes up to zero and then changes direction.

Figure 7.3: Scan

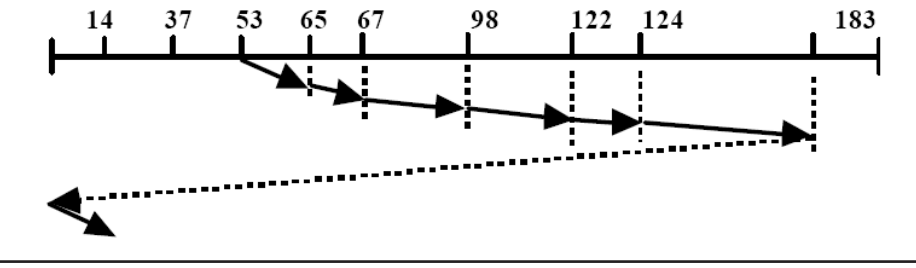


This algorithm is sometimes called "elevator" algorithm, since it resembles to the behaviour of elevator.

7.3.3 Cscan

A variant of scan designed to provide a more uniform wait. Starts from one end and moves towards the other end servicing all requests on its way. When the head reaches to the other end, it immediately returns to the beginning of the disk, without servicing any request on its return journey.

Figure 7.4: Cscan



Self Assessment

Multiple choice questions:

1. An online backing storage system capable of storing larger quantities of data is
   (*a*) CPU (*b*) Memory
   (*c*) Mass storage (*d*) Secondary storage
2. Which is an item of storage medium in the form of circular plate?
   (*a*) Disk (*b*) CPU
   (*c*) Printer (*d*) ALU

Notes

3. Which of the following disk scheduling techniques has a drawback of starvation?

   (*a*) SCAN (*b*) SST

   (*c*) FCFS (*d*) LIFO

4. The total time to prepare a disk drive mechanism for a block of data to be read from is its

   (*a*) latency

   (*b*) latency plus transmission time

   (*c*) latency plus seek time

   (*d*) latency plus seek time plus transmission time

7.4 Selecting a Disk Scheduling Algorithm

The performance of these algorithms depends heavily on the workload (number of requests). Under light load all algorithms perform the same. If the queue seldom has more than one outstanding request, then all algorithms are effectively the same. Their performance also depends upon the file organization and the type of generated requests. In a sequential processing and sequential file, the head movement will be minimum and therefore the seek time and latency time will be minimum so FCFS may perform better. A indexed sequential file, on the other hand, may include blocks that may be scattered all over the disk and a sequential processing with FCFS will be very slow. SSTF is quite common and scan and cscan are good for heavy load.

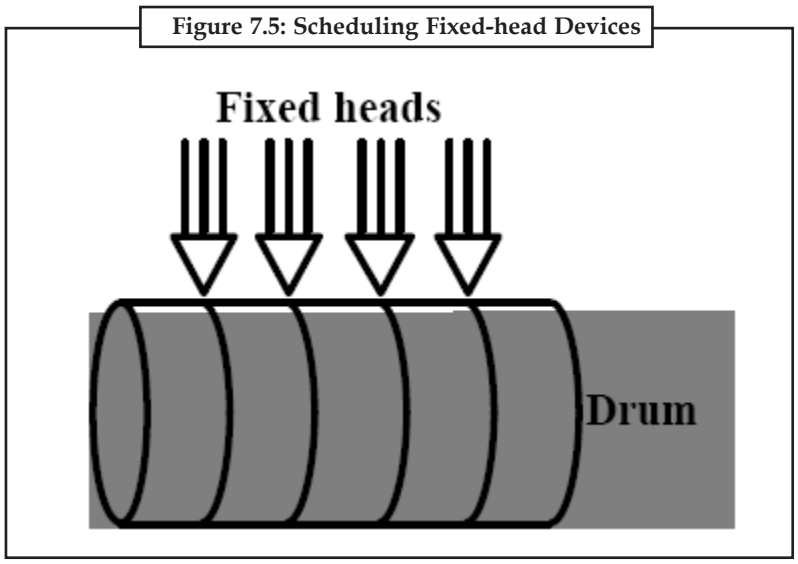
7.4.1 Scheduling Fixed-head Devices

Fixed head disk = DRUM. One head per track on the drum.

Seek time = 0.

Latency time < moving head disk.

Figure 7.5: Scheduling Fixed-head Devices



Different algorithm is required for this device.

Notes

7.5 Disk Management

The operating system is responsible for several other aspects of disk management, too. Here we discuss disk initialization, booting from disk, and bad-block recovery.

7.5.1 Disk Formatting

A new magnetic disk is a blank slate: It is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting (or **physical formatting**). **Low-level formatting** fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**. When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad. The ECC is an error-correcting code because it contains enough information that, if only a few bits of data have been corrupted, the controller can identify which bits have changed and can calculate what their correct values should be. The controller automatically does the ECC processing whenever a sector is read or written. Most hard disks are low-level formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the Idisk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but that also means fewer headers and trailers are written on each track, and thus increases the space available for user data. Some operating systems can handle only a sector size of 512 bytes.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting** (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.

Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed raw I/O. For example, some database systems prefer raw I/O because it enables them to control the exact disk location where each database record is stored. Raw 1/0 bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by implementing their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.



Task Give the step of disk management.

Notes



The controller automatically does the ECC processing whenever a sector is read or written. Most hard disks are low-level formatted at the factory as a part of the manufacturing process.

7.5.2 Boot Block

For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial bootstrap program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution. For most computers, the bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM, whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in a partition called the boot blocks, at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point), and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a non fixed location on disk, and to start the operating system running.

Even so, the full bootstrap code may be small. For example, MS-DOS uses one 512-byte block for its boot program.



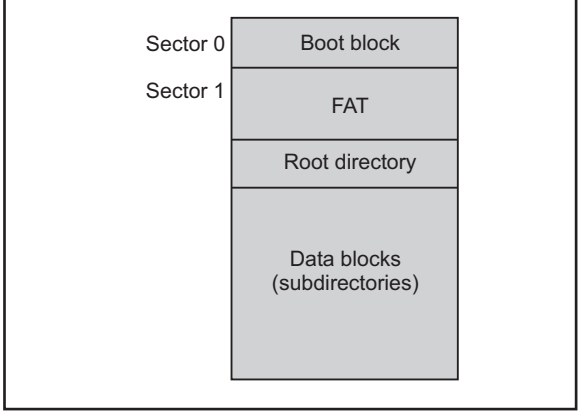
Be careful during the disk management because the disk is very important term in computer system.

7.5.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways. On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command does a logical format and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

Notes

Figure 7.6: MS-DOS Disk Layout



More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk. Low level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding. A typical bad-sector transaction might be as follows: The operating system tries to read logical block 87. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system. The next time that the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller. Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder, and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible. As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. Here is an example: Suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it. The replacement of a bad block generally is not a totally automatic process because the data in the bad block are usually lost. Thus, whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.



Did u know?

The operating system tries to read logical block 87. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

7.6 Swap Space Management

Swap space is an area on a high-speed storage device (almost always a disk drive), reserved for use by the virtual memory system for deactivation and paging processes. At least one swap

Notes

device (primary swap) must be present on the system. During system startup, the location (disk block number) and size of each swap device is displayed in 512 KB blocks. The swapper reserves swap space at process creation time, but does not allocate swap space from the disk until pages need to go out to disk. Reserving swap at process creation protects the swapper from running out of swap space. You can add or remove swap as needed (that is, dynamically) while the system is running, without having to regenerate the kernel. HP-UX uses both physical and pseudo swap to enable efficient execution of programs.

7.6.1 Pseudo-Swap Space

System memory used for swap space is called pseudo-swap space. It allows users to execute processes in memory without allocating physical swap. Pseudo-swap is controlled by an operating-system parameter; by default, swapmem on is set to 1, enabling pseudo-swap. Typically, when the system executes a process, swap space is reserved for the entire process, in case it must be paged out. According to this model, to run one gigabyte of processes, the system would have to have one gigabyte of configured swap space. Although this protects the system from running out of swap space, disk space reserved for swap is under-utilized if minimal or no swapping occurs.

To avoid such waste of resources, HP-UX is configured to access up to three-quarters of system memory capacity as pseudo-swap. This means that system memory serves two functions: as process-execution space and as swap space. By using pseudo-swap space, a one-gigabyte memory system with one-gigabyte of swap can run up to 1.75 GB of processes. As before, if a process attempts to grow or be created beyond this extended threshold, it will fail. When using pseudo swap for swap, the pages are locked; as the amount of pseudo-swap increases, the amount of lockable memory decreases.

For factory-floor systems (such as controllers), which perform best when the entire application is resident in memory, pseudo-swap space can be used to enhance performance: you can either lock the application in memory or make sure the total number of processes created does not exceed three-quarters of system memory. Pseudo-swap space is set to a maximum of three-quarters of system memory because the system can begin paging once three-quarters of system available memory has been used. The unused quarter of memory allows a buffer between the system and the swapper to give the system computational flexibility. When the number of processes created approaches capacity, the system might exhibit thrashing and a decrease in system response time. If necessary, you can disable pseudo-swap space by setting the tunable parameter swapmem on in /usr/conf/master.d/core-hpux to zero. At the head of a doubly linked list of regions that have pseudo-swap allocated is a null terminated list called pswaplist.

7.6.2 Physical Swap Space

There are two kinds of physical swap space—device swap and file-system swap.

7.6.2.1 Device Swap Space

Device swap space resides in its own reserved area (an entire disk or logical volume of an LVM disk) and is faster than file-system swap because the system can write an entire request (256 KB) to a device at once.

7.6.2.2 File-system Swap Space

File-system swap space is located on a mounted file system and can vary in size with the system's swapping activity. However, its throughput is slower than device swap, because free file-system blocks may not always be contiguous; therefore, separate read/write requests

Notes

must be made for each file-system block. To optimize system performance, file-system swap space is allocated and de-allocated in swchunk-sized chunks. Swchunk is a configurable operating system parameter; its default is 2048 KB (2 MB). Once a it is released for file system use, unless it has been preallocated with swapon. If swapping to file-system swap space, each chunk of swap space is a file in the file system swap directory, and has a name constructed from the system name and the swaptab index (such as becky.6 for swaptab[6] on a system named becky).

7.6.3 Swap Space Parameters

Several configurable parameters deal with swapping.

Table 7.1: Configurable Swap Space Parameters

Parameter	Purpose
swchunk	The number of DEV_BSIZE blocks in a unit of swap space, by deault, 2 MB on all systems.
maxswapchunks	Maximum number of swap chunks allowed on a system.
swapmen_on	Parameter allowing creation of more processes than you have physical swap space for, by using pseudo-swap.

7.6.4 Swap Space Global Variables

When the kernel is initialized, conf.c includes globals.h, which contains numerous characteristics related to swap space, shown in the next table. The most important to swap space reservation are swapspc_cnt, swapspc_max, swapmem_cnt, swapmem_max, and sys_mem.

Table 7.2: Swap Space Characteristics in Gobalsh

Element	Meaning
bswlist	head of free swap header list.
*pageoutbp	pointer to swbuf header used by pageout when swapping.
ref_hand	current reference hand used by pageout daemon.
maxmem	page count of actual max memory per process.
physmem	page count of physical memory on this CPU.
nswdev	number of swap devices.
nswap	pae count of size of swap space.
*fswdevt	pointer to file system swap table.
*swaptab	pointer to the table of swap chunks.
swapphys_buf	pages of physical swap space to keep available.
swapphys_cnt	pages of available physical swap space on disk.

Contd..

Notes

swapspc_cnt	Total amount of swap currently available on all devices and file systems enabled in units of pages. Updated each time a device or file system is enabled for swapping.
swapspc_max	Total amount of device and file-system swap currently enabled on the system in units of pages. Updated each time a device or file system is enabled for swapping.
swapspc_debit	number of of swap blocks by which to adjust swapspc_cnt.
swapspc_sparing	number of swap blocks unavailable to swap.
swapmem_max	Maximum number of pages of pseudo-swap enabled. Initialized to 3/4 available system memory.
swapmem_cnt	Total number of pages of pseudo-swap currently available. Initialized to 3/4 available system memory.
maxfs_pri	highest available device priority.
maxdev_pri	highest available swap prioirity.
sys_mem	Number of pages of memory not available for use as pseudo-swap. Initialized to 1/4 avaiable system memory.
sysmen_max	maximum pages not available for swap.
freemem	page count of remaining blocks of free memory.
freemem_cnt	Number of processes waiting for memory.

7.6.5 Swap Space Values

System swap space values are calculated as follows:

- Total swap available on the system is swapspc_max (for device swap and file system swap) + swapmem_max (for pseudo-swap).
- Allocated swap is swapspc_max – [sum(swdevt[n].sw_nfpgs)+ sum(fswdevt[n].fsw_nfpgs)] (for device swap and file systemswap) + (swapmem_max – swapmem_cnt) (for pseudo-swap). In HP-UX, only data area growth (using sbrk()) or stack growth will cause a process to die for lack of swap space. Program text does not use swap.

7.6.6 How Swap Space is Prioritized

All swap devices and file systems enabled for swap have an associated priority, ranging from 0 to 10, indicating the order that swap space from a device or file system is used. System administrators can specify swap-space priority using a parameter of the swapon(1M) command. Swapping rotates among both devices and file systems of equal priority. Given equal priority, however, devices are swapped to by the operating system before file systems, because devices make more efficient use of CPU time. We recommend that you assign the same swapping priority to most swap devices, unless a device is significantly slower than the rest. Assigning equal priorities limits disk head movement, which improves swapping performance.

Notes

7.6.7 Three Rules of Swap Space Allocation

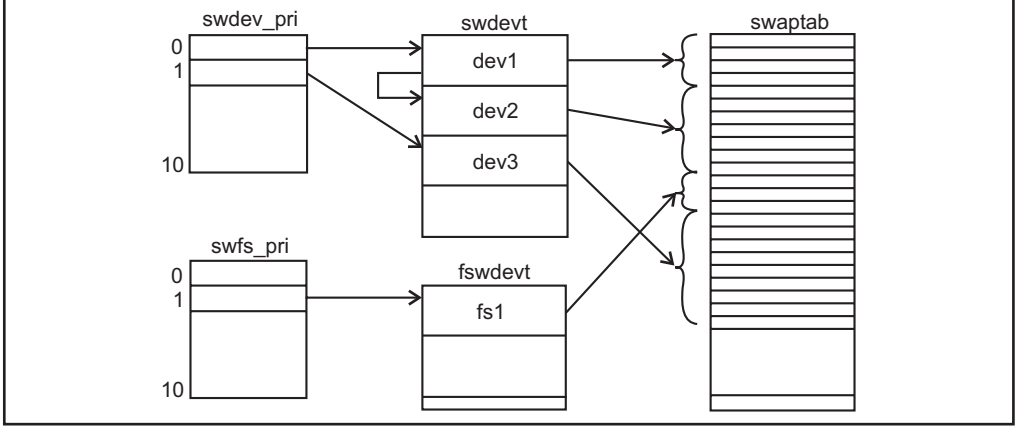
Start at the lowest priority swap device or file system. The lower the number, the higher priority; that is, space is taken from a system with a zero priority before it is taken from a system with a one priority.

If multiple devices have the same priority, swap space is allocated from the devices in a round-robin fashion. Thus, to interleave swap requests between a number of devices, the devices should be assigned the same priority. Similarly, if multiple file systems have the same priority, requests for swap are interleaved between the file systems.

In the figure below swap requests are initially interleaved between the two swap devices at priority 0.

If a device and a file system have the same swap priority, all the swap space from the device is allocated before any file-system swap space. Thus, the device at priority 1 will be filled before swap is allocated from the file system at priority 1.

Figure 7.7: Choosing a Swap Location



7.6.7.1 Swap Space Structures

Swapping is accomplished on HP-UX using the following data structures.

- Device swap priority array (swdev_pri[]), used to link together swap devices with the same priority. That is, the entry in swdev_pri[n] is the head of a list of swap devices having priority n. The first field in swdev_pri[] structure is the head of the list; the sw_next field in the swdevt[] structure links each device into the appropriate priority list.
- File system swap priority array (swfs_pri[]), which serves the same purpose as swdev_pri[], but for file system swap priority.
- Device swap table (struct swdevt), defined in conf.h to establish the fundamental swap device information.
- File system swap table (struct fswdevt), defined in swap.h for supplimentary file-system swap.
- Swap table of available chunks (struct swaptab), which keeps track of the available free pages of swap space.
- Mapping of swap pages (struct swapmap), whose entries together with swaptab combine for a swap disk block descriptor. The following table details the elements of the struct swdevt.

Notes

Table 7.3: Device Swap Table (struct swdevt)

Element	Meaning
sw_dev	Actual swap device, as defined by its major (upper 8 bits) and minor (lower 24 bits) numbers.
sw_enable	Enabled flag. Zero if device swap is disabled; one if enabled.
sw_start	Offset into the swap area on disk, in kilobytes.
sw_nblksavail	Size of swap area, in kilobytes.
sw_nblksenabled	Number of blocks enabled for swap. Must be a multiple of swchunk (2MB default).
sw_nfpgs	Number of free swap pages on the device.
	Updated whenever a page is used or freed.
sw_priority	Priority of swap device (1-10).
sw_head, sw_tail	First and last swaptab[] entry associated with swap device.
sw_next	Pointer to the next device swap entry (swdevt) at this priority; implemented as a circular list used to update the pointer in swdev_pri for round-robin use of all devices at a particular priority.

The following table details the principle elements of the struct fswdevt.

Table 7.4: File System Swap Table (struct fswdevt)

Element	Meaning
fsw_next	Pointer to next file system swap (fswdevt entry) at this priority; implemented as a circular list.
fsw_enable	Enabled flag. Zero if file-system swap is disabled; one if enabled.
fsw_nfpgs	Number of free swap pages in this file system swap; updated whenever a page is used or freed.
fsw_allocated	Number of swchunks (2MB default) allocated on this file-system swap.
fsw_min	Minimum swchunks to be preallocated when the file-system swap is enabled.
fsw_limit	Maximum swchunks allowed on file system; unlimited if set to zero.
fsw_reserve	Minimum blocks (of size fsw_bsize) reserved for non-swap use on this file system.
fsw_priority	Priority of device (0-10). Priority can also be determined by identifying swfs_pri [] linked list.

Contd...

Notes

few_vnode	vnode of the file system swap directory (/paging) under which the swap files are created.
fsw_bsize	Block size used on this file system; used to determine how much space fsw_reserve is reserving.
fsw_head	Index into swaptab [] of first, last entry
fsw_tail	associated with this file system swap.
fsw_mntpoint	File system mount point; character representation of fsw_vnode, used for utilities (such as swapinfo (1M) and error messages.

7.6.7.2 Swaptab and Swapmap Structures

Two structures track swap space. The swaptab[ ] array tracks a chunk of swap space. swapmap entries hold swap information on a per-page level. swaptab defaults to track a 2MB chunk of space and swapmap tracks each page within that 2MB chunk. Each entry in the swaptab[ ] array has a pointer (called st_swpmp) to a unique swapmap. swapmap entries have backwards pointers to the swaptab index. There is one entry in the swapmap for each page represented by the swaptab entry (default 2 MB, or 512 pages); that is, swapmap conforms in size to swchunk. A linked list of free swap pages begin at the swaptab entry's st_free and use each free swapmap entry's sm_next. When a page of swap is needed, the kernel walks the structures (using the getswap( ) routine in vm_swalloc.c), which calls other routines that actually locate the chunk, and so forth.

- Beginning with the lowest priority, we begin by examining swdev_pri[].curr, which points to a swdevt entry.
- If sw_nfpgs is zero (no free pages), we follow the pointer sw_next to get the next swdevt entry at this priority.
- If none of these have free pages, we move on to swfs pri[].curr, the file system swap at this priority, checking fsw_nfpgs for free pages.

If we are still unsuccessful, we move to the next priority and try again.

- Once we find a swdevt or fswdevt with free pages, we walk that device's swaptab list, starting with sw head or fsw head, and using st next in each swaptab entry, until we find a swaptab entry with non-zero st nfpgs.
- st free points to the first free swapmap entry (and thus first free page) in this swaptab chunk.
- The swalloc( ) routine creates a disk block descriptor (dbd) using 14 bits of dbd_data for the swaptab index and 14 bits for the swapmap index. The r_bstore in the region is set to the disk device vnode or the file system directory vnode, and the dbd is marked DBD_BSTORE.

When faulting in from swap, the same process is followed as for faulting in from the file system: r_bstore and dbd_data are hashed together and checked for a soft fault, then devswap_pagein( ) is called. The devswap_pagein( ) routine uses the dbd_data as a 14-bit swaptab index and a 14-bit swapmap index to determine the location of the page on disk. Now all information needed to retrieve the page from swap has been stored.

Notes

Figure 7.8: The Swaptab and Swapmap Structures

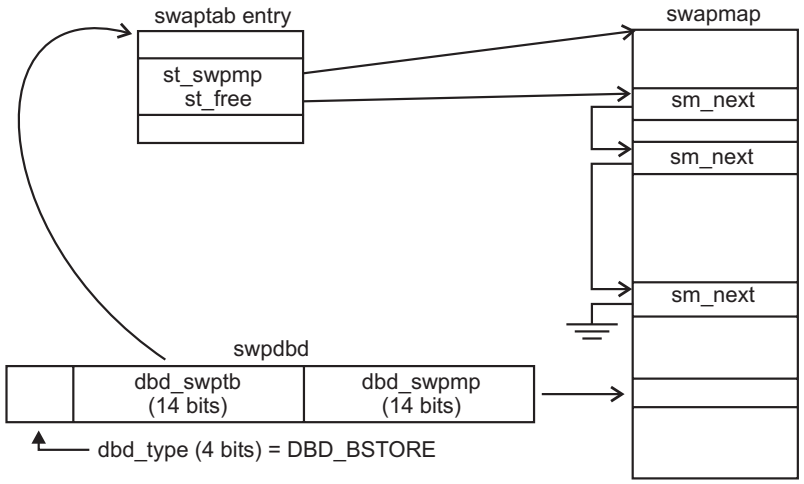


Table 7.5: Swap Table Entry (struct swaptab)

Element	Meaning
st_free	Index to the first free page in the chunk. Each entry maps to a 4KB-age of swap.
st_next	Index to next swaptab entry for same device or file-system swap; at end of list, st_next is – 1.
st_flags	ST_INDEL: File-system swap flag, indicating chunk is being deleted; do not allocate pages from it. Set only by the realswapoff () routine.
	ST_FREE: File-system swap flag, indicating chunk may be deleted, because none of its pages are in use. In the case of remote swap, the chunk should not be deleted immediately; set st_free_time to current time plus 30 minutes has elapsed, the chunk can be freed. If the chunk is needed during the interim, the flag can be cleared using.
	chunk_release(). called from lsync(). ST_INUSE: swaptab entry is being changed.
st_dev,	Pointers to swdevt entry that references the
st_fsp	swaptab entry.
st_nfpgs	Number of free pages in this (swchunk) swaptab entry.
st_swpmp	Pointer to swapmap [] array that defines this swchunk of swap pages.
st_free_time	Indicates when remote fs chunk can be freed (see explanation of ST_FREE flag).

Notes

Table 7.6: Swap Map Entry (struct swapmap)

Element	Meaning
sm_unct	Number of threads using the page. When decremented to zero, the swap page is free and the free pages linked list can be updated.
sm_next	Index of the next free page in the swapmap []. This is valid only if sm_ucnt is zero; that means that this swapmap entry is included in the linked list beginning with swaptab's st_free

.


Did u know?

A linked list of free swap pages begin at the swaptab entry's st_free and use each free swapmap entry's sm_next. When a page of swap is needed, the kernel walks the structures (using the getswap() routine in vm_swalloc.c), which calls other routines that actually locate the chunk, and so forth.

7.7 Overview of RAID Structure

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach a large number of disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAID), are commonly used to address the performance and reliability issues. In the past, RAIDS composed of small cheap disks were viewed as a cost effective alternative to large, expensive disks; today, RAIDS are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the *I* in RAID stands for "independent", instead of "inexpensive."

7.7.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the mean time to failure of a single disk is 100,000 hours. Then, the mean time to failure of some disk in an array of 100 disks will be 100,000/100 = 1,000 hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data-such a high rate of data loss is unacceptable. The solution to the problem of reliability is to introduce redundancy; we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost. The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called mirroring (or shadowing). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is replaced.

The mean time to failure-where **failure** is the loss of data-of a mirrored disk depends on two factors: the mean time to failure of the individual disks, as well as on the mean time to repair, which is the time it takes (on average) to replace a failed disk and to restore the data on it. Suppose that the failures of the two disks are independent; that is, the failure of one disk is not connected to the failure of the other. Then, if the mean time to failure of a single disk is 100,000 hours and the mean time to repair is 10 hours, then the mean time to data loss of a mirrored disk system is 100, 0002/(2 * 10) = 500 * l06 hours, or 57,000 years!

Notes

You should be aware that the assumption of independence of disk failures is not valid. Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both disks at the same time. Also, manufacturing defects in a batch of disks can cause correlated failures. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single disk systems. Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. However, even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. The solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Some extra actions are required when we restart after a power failure, to recover from incomplete writes.


Did u know?

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called mirroring (or shadowing). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is replaced.

7.7.2 Improvement in Performance via Parallelism

Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled. With multiple disks, we can improve the transfer rate as well (or instead) by striping data across multiple disks. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping.** For example, if we have an array of eight disks, we write bit i of each byte to disk *i.* The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that have eight times the access rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as many data in the same time as on a single disk. Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or divides 8. For example, if we use an array of four disks, bits ***I*** and 4+i of each byte go to disk i. Further, striping does not need to be at the level of bits of a byte: For example, in **block-level striping,** blocks of a file are striped across multiple disks; with *n* disks, block ***i*** of a file goes to disk (i mod n) + 1. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

In summary, there are two main goals of parallelism in a disk system:

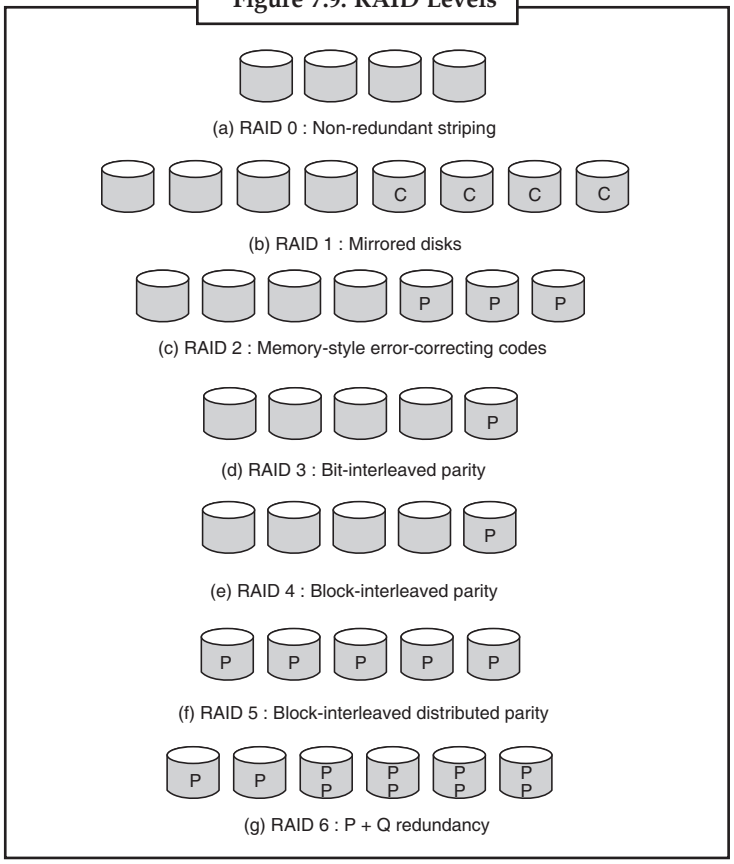
1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

7.8 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with "parity" bits (which we describe next) have been proposed. These schemes have different cost-performance tradeoffs and are classified into levels called RAID **levels.** We describe the various levels here; Figure, shows them pictorially (in the figure, *P* indicates error-correcting bits and C indicates a second copy of the data). In all cases depicted in the figure, four disks' worth of data is stored, and the extra disks are used to store redundant information for failure recovery.

Notes

Figure 7.9: RAID Levels



RAID Level 0: RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure 7.9(*a*) shows an array of size 4.

RAID Level 1: RAID level 1 refers to disk mirroring. Figure 7.9(*b*) shows a mirrored organization that holds four disks' worth of data.

RAID Level 2: RAID level 2 is also known as **memory-style error-correcting code (ECC) organization.** Memory systems have long implemented error detection using parity bits. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte set to 1 is even (parity=O) or odd (parity=l). If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all single-bit errors are detected by the memory system. Error-correcting schemes store two or more extra bits, and can reconstruct the data if a single bit gets damaged. The idea of ECC can be used directly in disk arrays via striping of bytes across disks. For example, the first bit of each byte could be stored in disk 1, the second bit in disk 2, and so on until the eighth bitis stored in disk 8, and the error-correction bits are stored in further disks. This scheme is shown pictorially in Figure 7.9(*c*), where the disks labeled *P* store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error correction bits can be read from other disks and be used to reconstruct the damaged data. Figure 7.9(*c*) shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.

RAID Level 3: RAID level **3,** or **bit-interleaved parity organization,** improves on level 2 by noting that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows. If one of the sectors gets damaged, we know exactly which sector it is, and, for each bit in the sector, we can figure out whether it is a 1 or a 0 by computing the parity

Notes

of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1. RAID level **3** is as good as level 2 but is less expensive in the number of extra disks (it has only a one-disk overhead), so, level 2 is not used in practice. This scheme is shown pictorially in Figure 7.9(*d*). RAID level **3** has two benefits over level 1. Only one parity disk is needed for several regular disks, unlike one mirror disk for every disk in level 1, thus reducing the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with N-way striping of data, the transfer rate for reading or writing a single block is N times as fast as with a RAID-level-1 organization using N-way striping. On the other hand, RAID level **3** supports a lower number of I/Os per second, since every disk has to participate in every I/O request. A further performance problem with RAID **3** (as with all parity-based RAID levels) is the expense of computing and writing the parity. This overhead results in significantly slower writes, as compared to non-parity RAID arrays. To moderate this performance penalty, many RAID storage arrays include a hardware controller with dedicated parity hardware. This offloads the parity computation from the CPU to the array. The array has a non-volatile **RAM (NVRAM)** cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the spindles. This combination can make parity RAID almost as fast as non-parity. In fact, a caching array doing parity RAID can outperform a non-caching non-parity RAID.

RAID Level 4: RAID level 4, or block-interleaved parity organization, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks. This scheme is shown pictorially in Figure 7.9(*e*). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated.

Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. This is known as the read-modify-write. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two new blocks.

RAID Level 5: RAID level 5, or block-interleaved distributed parity, differs from level 4 by spreading data and parity among all N + 1 disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. For example, with an array of five disks, the parity for the nth block is stored in disk (n mod 5) + 1; the nth blocks of the other four disks store actual data for that block. This setup is denoted pictorially in Figure 7.9(*f*), where the Ps are distributed across all level 1, thus reducing the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with N-way striping of data, the transfer rate for reading or writing a single block is N times as fast as with a RAID-level-1 organization using N-way striping. On the other hand, RAID level **3** supports a lower number of I/Os per second, since every disk has to participate in every I/O request. A further performance problem with RAID **3** (as with all parity-based RAID levels) is the expense of computing and writing the parity. This overhead results in significantly slower writes, as compared to non-parity RAID arrays. To moderate this performance penalty, many RAID storage arrays include a hardware controller with dedicated parity hardware. This offloads the parity computation from the CPU to the array. The array has a non-volatile **RAM (NVRAM)** cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the spindles. This combination can make parity RAID almost as fast as non-parity. In fact, a caching array doing parity RAID can out perform a non-caching non-parity RAID.

Notes

RAID Level 0 + 1: RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, it provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, it doubles the number of disks needed for storage, as does RAID 1, so it is also more expensive. In RAID 0 + 1, a set of disks are striped, and then the stripe is mirrored to another, equivalent stripe. Another RAID option that is becoming available commercially is RAID 1 + 0, in which disks are mirrored in pairs, and then the resulting mirror pairs are striped. This RAID has some theoretical advantages over RAID 0 + 1. For example, if a single disk fails in RAID 0 + 1, the entire stripe is inaccessible, leaving only the other stripe available. With a failure in RAID 1 + 0, the single disk is unavailable, but its mirrored pair is still available as are all the rest of the disks.

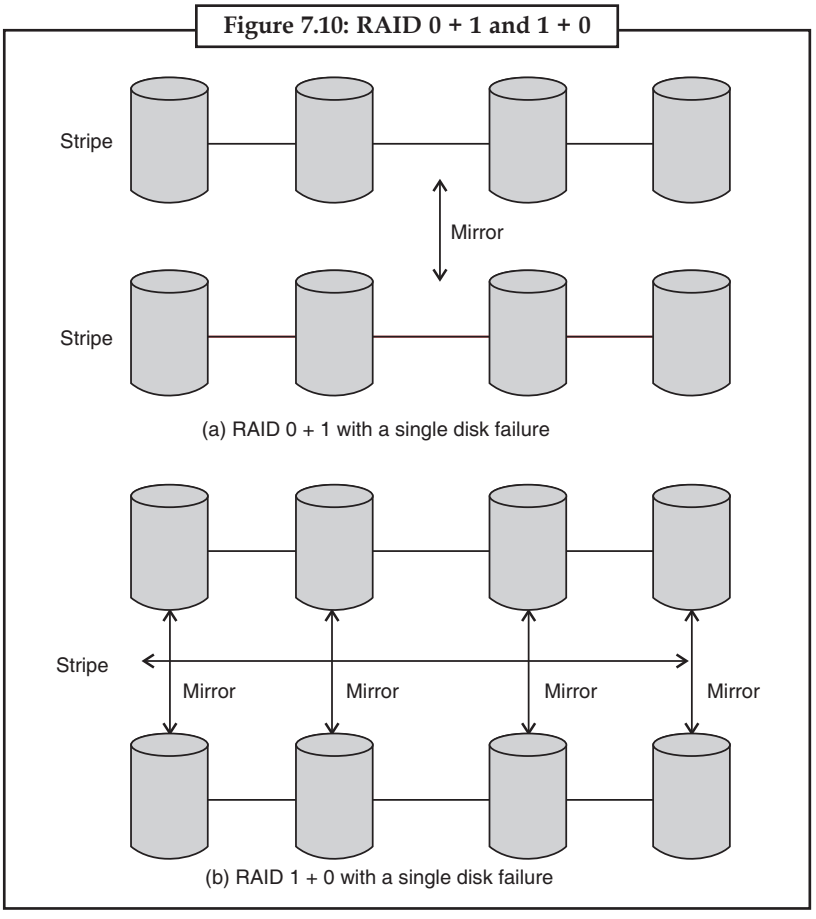
Finally, we note that numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

7.8.1 Selecting a RAID Level

If a disk fails, the time to rebuild its data can be significant and will vary with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data in a failed disk. The rebuild performance of a RAID system may be an important factor if continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time to failure.

RAID level 0 is used in high-performance applications where data loss is not critical. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID 0 + 1 and 1 + 0 are used where performance and reliability are important, for example for small databases.

Figure 7.10: RAID 0 + 1 and 1 + 0



Notes

Due to RAID 1's high space overhead, RAID level 5 is often preferred for storing large volumes of data. Level 6 is not supported currently by many RAID implementations, but it should offer better reliability than level 5. RAID system designers have to make several other decisions as well. For example, how many disks should be in an array? How many bits should be protected by each parity bit? If more disks are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second disk will fail before the first failed disk is repaired is greater, and that will result in data loss. One other aspect of most RAID implementations is a hot spare disk or disks. A **hot spare** is not used for data, but is configured to be used as a replacement should any other disk fail. For instance, a hot spare can be used to rebuild a mirror pair should one of the disks in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed disk to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

7.8.2 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.


Did u know?

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.


Case Study

Stable-storage Implementation

Stable storage is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures. To be considered atomic, upon reading back a just written-to portion of the disk, the storage subsystem must return either the write data or the data that was on that portion of the disk before the write operation. Most computer disk drives are not considered stable storage because they do not guarantee atomic write: an error could be returned upon subsequent read of the disk where it was just written to in lieu of either the new or prior data.

Multiple techniques have been developed to achieve the atomic property from weakly-atomic devices such as disks. Writing data to a disk in two places in a specific way is one technique and can be done by application software. Most often though, stable storage functionality is achieved by mirroring_data on separate disks via RAID technology (level 1 or greater). The RAID controller implements the disk writing algorithms that enable separate disks to act as stable storage. The RAID technique is robust against some single disk failure in an array of disks whereas the software technique of writing to separate areas of the same disk only protects against some kinds of internal disk media failures such as bad sectors in single disk arrangements.

Contd...

Notes

To implement stable storage:

1. Replicate information on more than one nonvolatile storage media with independent failure modes.
2. Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Questions:

1. What is the use of stable-storage implementation?
2. How stable-storage in an operating system can be achieved?



Lab Exercise

1. What is the difference between primary and secondary storage?
2. What is logical device?
3. Disk Operations Read/Write operations:
4. Calculating the disk access time on different cases:
   (*a*) The file is stored contiguous.
   (*b*) The file is stored on same cylinder.
   (*c*) The file is stored randomly or scattered on different places.

Self Assessment

Multiple choice questions:

5. Which among the following are the best tools for fixing errors on disks?
   (*a*) Fdisk (*b*) Scandisk
   (*c*) Chkdsk (*d*) Fixdsk
6. Which command can be used to create the disk's tracks and sectors?
   (*a*) Fdisk (*b*) Format
   (*c*) Chkdsk (*d*) Attrib
7. Which command is used to create root directory and FAT on disk?
   (*a*) Chkdsk (*b*) Command.com
   (*c*) Format (*d*) Fat
8. ..... is a technique of temporarily removing inactive programs from the memory of computer system.
   (*a*) Swapping (*b*) Spooling
   (*c*) Semaphore (*d*) Scheduler

Fill in the blanks:

9. Low-level formatting fill the ..... with a special data structure for each sector.
10. System memory used for swap-space is called ..... space.

Notes

11. If multiple devices have the same priority, swap-space is allocated from the devices in a ..... fashion.

7.9 Summary

- Secondary storage devices which are usually accessed via some kind of controller. This contains registers that can be directly accessed by the CPU like main memory ("memory mapped").
- Secondary storage, sometimes called auxiliary storage, is storage separate from the computer itself, where you can store software and data on a semi-permanent basis. To provide the bulk of secondary storage for modern computer systems Disc used and Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks.
- In multiprograming systems, many processes may be generating requests for reading and writing disk records.

7.10 Keywords

Bit-level Stripping: Data striping consists of splitting the bits of each byte across multiple disks; such striping is called bit-level striping.

Constant Linear Velocity (CLV): Constant linear velocity (CLV) is a qualifier for the rated speed of an optical disc drive and may also be applied to the writing speed of recordable discs.

Data Stripping: The distribution of a unit of data over two or more hard disks, enabling the data to be read more quickly, known as data striping.

Error Correcting Code (ECC): Error correction code is a coding system that incorporates extra parity bits in order to detect errors.

Logical Blocks: The logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes.

Logical Formatting: Logical formatting is the process of placing a file system upon a hard disk drive partition of a hard disk so that an operating system can use available hard disk platter space to store and retrieve files.

Low-level Formatted: The sector identification on a disk that the drive uses to locate sector for reading and writing is called low level formatted.

7.11 Review Questions

1. Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

   (*a*) How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

   (*b*) If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

Notes

2. In what situations would using memory as a RAM disk be more useful than using it as a disk cache?
3. None of the disk-scheduling disciplines, except FCFS, are truly fair (starvation may occur).
   (*a*) Explain why this assertion is true.
   (*b*) Describe a way to modify algorithms such as SCAN to ensure fairness.
   (*c*) Explain why fairness is an important goal in a time-sharing system.
   (*d*) Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.
4. Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

   86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.

   Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?
   (*a*) FCFS
   (*b*) SSTF
   (*c*) SCAN
   (*d*) LOOK
   (*e*) C-SCAN
   (*f*) C-LOOK
5. Write a Java program for disk scheduling using the SCAN and C-SCAN disk-scheduling algorithms.
6. Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.
7. Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?
8. Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?
9. Why is it important to balance file system I/O among the disks and controllers on a system in a multitasking environment?
10. Is there any way to implement truly stable storage? Explain your answer.
11. Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

Notes

12. What would be the effects on cost and performance if tape storage had the same areal density as disk storage?
13. If magnetic hard disks eventually have the same cost per gigabyte as do tapes, will tapes become obsolete, or will they still be needed? Explain your answer.
14. Imagine that a holographic storage drive has been invented. Suppose that a holographic drive costs $10,000 and has an average access time of 40 milliseconds. Suppose that it uses a $100 cartridge the size of a CD. This cartridge holds 40,000 images, and each image is a square black and white picture with resolution 6,000 x 6,000 pixels (each pixel stores 1 bit). Suppose that the drive can read or write one picture in 1 millisecond.
15. Answer the following questions:
    (*a*) What would be some good uses for this device?
    (*b*) How would this device affect the 1/0 performance of a computing system?
    (*c*) Which other kinds of storage devices, if any, would become obsolete as a result of this device being invented?
16. Suppose that we agree that 1 KB is 1,024 bytes, 1 MB is $1,024^2$ bytes, and 1 GB is $1,024^3$ bytes. This progression continues through terabytes, petabytes, and exabytes ($1,024^6$). Several newly proposed scientific projects plan to be able to record and store a few exabytes of data during the next decade. To answer the following questions, you will need to make a few reasonable assumptions; state the assumptions that you make.
    (*a*) How many disk drives would be required to hold 4 exabytes of data?
    (*b*) How many magnetic tapes would be required to hold 4 exabytes of data?
    (*c*) How many optical tapes would be required to hold 4 exabytes of data?
    (*d*) How many holographic storage cartridges would be required to hold 4 exabytes of data?
    (*e*) How many cubic feet of storage space would each option require?
17. Consider a RAID Level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?
    (*a*) A write of one block of data.
    (*b*) A write of seven continuous blocks of data.

Answers to Self Assessment

1. (*c*) 2. (*a*) 3. (*b*) 4. (*c*) 5. (*b*)
6. (*b*) 7. (*c*) 8. (*a*) 9. disk
10. pseudo-swap 11. round-robin

Notes

7.12 Further Readings


Books

Operating Systems, by Stuart E. Madnick, John J. Donovan.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.


Online link

wiley.com/coolege.silberschatz

Unit 8: System Protection

Notes

CONTENTS

Objectives

Introduction

8.1 Goals of Protection

8.1.1 Computer Architecture to Support OS Protection

8.2 Access Matrix

8.2.1 Implementation of Access Matrix

8.2.2 Access Proxies

8.2.3 Stack Check/Modified Name Space

8.3 Access Control

8.3.1 Concepts

8.3.2 Policies, Models and Mechanisms

8.3.3 Discretionary Access Control (DAC)

8.3.4 Non-Discretionary Access Control

8.3.5 Mandatory Access Control (MAC)

8.3.6 Role-based Access Control

8.3.7 Temporal Constraints

8.3.8 Workflow

8.3.9 Chinese Wall

8.4 Revocation of Access Rights

8.5 Capability Based System

8.5.1 Basic Notions

8.6 Summary

8.7 Keywords

8.8 Review Questions

8.9 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Discuss goals of protection of system protection
- Explain access matrix
- Understand implementation of access matrix
- Discuss access control
- Explain revocation of access rights
- Understand capability based system

Introduction

The processes in an operating system must be protected from one another's activities. To provide such protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcement. We distinguish between protection and security, which is a measure of confidence that the integrity of a system and its data will be preserved.

8.1 Goals of Protection

Implementation of protection in an OS generally involves three factors. The interface to the user,

the interface to the hardware, and the decision making process with regard to filling requests.

If we expand our thinking, we can make the same statement about protection in any

environment where there are external users, underlying capabilities, and a protection function to be performed.

Because the external interface is so heterogeneous and so little has been done to model its effect on protection systems, it is difficult to cover it at more than a cursory level. Hardware protection and the decision making mechanisms are fairly well developed arts, however, so we will cover them here at length.

8.1.1 Computer Architecture to Support OS Protection

The implementation of protection in OSs almost always depends heavily on a hardware separation mechanisms. A separation mechanism is a way to partition information into areas that only communicate through well-defined and controlled channels. In order to enforce separation against a serious attacker, it is insufficient to make information flow inconvenient or available only to the knowledgeable as is the case in most personal computer systems.

One way to provide protection is to simulate a hypothetical machine on a physical machine so that all operations of the hypothetical machine are controlled by the simulation. This could be as secure a system as any purely physical system, but the performance of the physical machine is severely reduced because most of the time is spent in controlling the simulation rather than

Notes

performing user level processing. A purely physical, and very reliable method must be used for most real-world applications.

A physical means for implementing protection is typically provided by architectural features of computer hardware. In order to provide physical partitioning, we must assure that there is no path through which information can flow between users. In the hardware domain, information is typically processed by finite state machines (FSMs) which maintain a state, and transform inputs combined with state information into outputs and changes in state [ ]. The only information that persists in such a system is state information, and thus we can protect information from flowing between domains by keeping their state information separate. This is just like OSs based on a sign-up sheet, wherein each user has full access to the machine over a different period of time.

A typical model for a computer system is the Von Neumann model in which a machine consists of a control unit (C), an input and output unit (I/O), an arithmetic logic unit (ALU), and a memory unit (M) VonNeumann63. The machine fetches instructions telling it what to do next from the memory. The control unit which consists of FSMs then controls M, I/O, ALU, and C to implement the instructions. In most such machines, state information is maintained in a set of special memory elements called "registers", in M, and in peripheral devices such as disks and tapes.

There are generally two classes of these registers. One class is used primarily by C for remembering the instruction being performed, the portion of the instruction currently under way, the conditions of the ALU, the memory location of the next instruction to be performed, and other control related information. The other class is used to store the current values of data registers associated with user programs including the result of the last arithmetic operation, the user's general purpose registers, and other user data.

Registers are generally quite expensive to implement compared to the state information stored in M and I/O devices, they operate at extremely high speeds compared to state information stored M and I/O, and they are central to the interpretation of instructions, so they must be connected to many other devices using specially designed complex switching devices. As a result, it is generally not cost or performance effective to provide enough sets of registers to store state information for all of the processes that could possibly coexist on a typical machine. As an alternative, designers provide two sets of registers that can be stored in and reloaded from M, the set of registers in use at any given time being associated with the protection state of the machine.

When the OS is executing its 'kernel' program, it uses a completely independent set of state information from that used by any other process, and it generally has access to the full range of hardware instructions available to the physical machine. When any other program is operating, it generally uses limited state information, and has access to only a limited number of instructions that effect the physical machine. When the machine is using the kernel registers and has full access, we will say the machine is running in the kernel state, or kernel mode, and when the other state registers are used and limited instructions are allowed, we say the machine is in the user state, or user mode. Several citations are given in [ ] for protected hardware states, but as yet, the originator of this concept has not been identified.

In user mode, machine instructions that are only allowed to kernel mode cannot be executed, so a typical method for providing these services to users is through 'system calls'. A system call is typically implemented by executing an instruction which is not allowed in user mode. This causes an error which the hardware interprets as a request to change to the kernel mode.

Notes

Since the register that determines where the next instruction comes from is different in kernel mode than in user mode, the next instruction comes from the kernel's memory location after the location of the instruction which the kernel used to return to user mode after the last kernel call. This instruction is typically the beginning of a routine which determines the nature of the request from the state of the user's registers and memory, and either processes the request or denies it.

A similar process occurs whenever the system clock signals to the kernel that it's time to switch to another user's turn to execute programs in a timesharing environment. In this case, the kernel typically saves the values of the old user's registers in the kernel's memory space, restores the register values associated with the new user from kernel space, and returns to user mode. Thus, the next user continues wherever they left off, and the last user waits to regain access to the system.

The astute reader will see several problems that remain to be resolved. In particular, the memory of the machine must be protected in some manner or the state information of one domain could be altered or examined by writing or reading its memory from another domain. Similarly, state information in peripheral devices may be exploited in this manner unless some additional protection is provided. Finally, information may be exchanged between domains by differing usage of shared facilities.

Because of the prominent position of memory in the fetching of instructions and data and the storage of control state information for the kernel and user domains, it presents a problem that must be addressed by hardware in order to afford efficient processing under most current architectures. Memory is generally protected through the use of a technique called 'memory mapping'. Memory mapping involves the use of a special set of control registers that translate memory locations specified by a user into physical memory locations in M. Thus a given user might access a memory location numbered 125, which the 'translation buffer' maps into physical location 2978. If the kernel assures that the memory maps of different users don't have overlaps in physical memory locations, we are guaranteed that they don't share state information, and thus we maintain the separation mechanism. Similarly, shared memory can be implemented with this mechanism for controlled communication. In most practical systems, memory mapping is done on a page by page basis (i.e. in 1000 word groups) so that the size of the memory map needn't be enormous or expensive.

All I/O is generally performed by kernel mode instructions so the kernel can enforce protection of I/O resources. In the case of terminal I/O, the user may have an interface that is nearly identical to the kernel's instructions, while for shared devices such as disks and printers, the kernel may abstract the physical characteristics of the device completely, and leave the user with a purely logical abstraction such as a file system.

The logical extension of the two state machine (i.e. kernel and user mode) is the use of machines with numerous states, each with an increasing level of control. In general, a two state machine is much simpler to implement. Because complexity increases the likelihood of errors, and all protection relevant instructions are likely to be equally critical to system wide protection, this is usually the way implementations are performed. If multiple logical protection 'rings' are desired, they can be implemented with a two state machine in which certain domains are treated differently than others by the kernel, rather than by providing a multitude of control registers and classes of instructions in hardware.

Notes

8.2 Access Matrix

Access matrices are widely used to hold a symbolic representation of the set of rights available to subjects for access to objects. In the case of the PO set policy, 'flow control matrices' hold only a single right which determines whether or not flow is permitted from domain to domain, while more complex policies require more rights, and thus more complex software. Matrices are well understood data structures which have been used for a long time, and implementations are very straight forward. The programmer merely implements a table lookup for every protection related OS request to determine whether or not the requested right is to be granted.

Figure 8.1: Access Matrix

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry credit	
User B	R	Own R W	W	R	Inquiry debit	Inquiry credit
User C	R W	R		Own R W		Inquiry debit

Access Matrix Example

An access matrix has several standard operations associated with it:

- Entry of a right into a specified cell
- Removal of a right from a specified cell
- Creation of a subject
- Creation of an object
- Removal of an subject
- Removal of an object

The two most used implementations are access control lists and capabilities. Access control lists are achieved by placing on each object a list of users and their associated rights to that object. For example, if we have file 1, file 2 and file 3, and users *(subjects) Pradip and Sally, an access control list might look like:

Objects (Files)

Users	File 1	File 2	File 3
Pradip	RWX	R-X	RW-
Sally	---	RWX	R--

The rights are R (Read), W (Write) and X (execute). A dash indicates that the user does not have that particular right. Thus, Pradip does not have permission to execute File 3, and Sally has no rights at all on File 1.

Notes

Capabilities are accomplished by storing on each subject a list of rights the subject has for every object. This effectively gives each user a keyring. To remove access to a particular object, every user (subject) that has access to it must be "touched". A touch is an examination of a user's rights to that object and potentially removal of rights. This brings back the problem of sweeping changes in access rights. Here is what an implementation of capabilities might look like, using the above example:

Users			
Pradip	File 1: RWX	File 2: R-X	File 3: RW-
Sally	File 1: ---	File 2: RWX	File 3: R--

Access restrictions such as access controls lists and capabilities sometimes are not enough. In some cases, information needs to be tightened further, sometimes by an authority higher than the owner of the information. For example, the owner of a top secret document in a government office might deem the information available to many users, but his manager might know the information should be restricted further than that. In this case, the flow of information needs to be controlled — secure information cannot flow to a less secure user.

8.2.1 Implementation of Access Matrix

Recall, that an access matrix may be implemented by access control lists or capabilities. These traditional implementation approaches must be extended in many ways to implement the access-control properties of distributed applications mentioned above:

Network-Wide Capabilities: A capability no longer references a local object; thus a scheme for addressing a remote object must be implemented. To address this problem, Amoeba stores in a capability an encryption of the access rights to and a network-wide id of a protected object.

Replicated Access Lists: A way must be found to replicate access control lists of replicas. Both Suite and Lotus Notes use the mechanisms provided by the replication system for replicating objects to also replicate access control lists of these objects.

Application-Defined Objects: Traditional operating systems do not support user-defined objects, thereby restricting themselves to protecting predefined rights such as file rights. As we saw above, distributed systems must protect application-defined operations such as connect. Two approaches have been used to protect application-defined objects. One approach, used in Hydra, is to develop a kernel that manages application-defined objects, intercepting, and thereby guarding, all operations on these objects. An alternative approach is to provide access control in user-space. X servers, Suite dialogue managers and Web browsers are examples of user-level code implementing access control. The advantage of the second approach is that it can be used with existing, non object-oriented, operating systems and access checks do not require context switches to the operating system. However, some form of authentication facility is required to verify a subject's identity.

8.2.2 Access Proxies

A general technique for implementing access-control in user-space is to implement for each protected class a proxy class that has the same interface as the protected class, performs access checks, and forwards operations to the protected class if these checks succeed.

8.2.3 Stack Check/Modified Name Space

Access control for the process that allows code to be dynamically downloaded into it needs to be distinguished between local and downloaded code and provide restricted rights to downloaded code to ensure, for instance, that it does not destroy or leak the contents of local data. Java-enabled Web browsers illustrate how such a mechanism can be supported. Two approaches

Notes

have been used by them to restrict the access of downloaded Java applets. One approach relies on the fact that separate class loader objects are used to load local and remote code and that the stack frame of each method points to the object that loaded it. As a result, when a protected method is called, the browser can provide restricted access if the stack contains a method that was called (directly or indirectly) by downloaded code.

The other approach relies on the fact that the loader can determine the name space of downloaded code. It creates restrictive proxy classes for the protected classes, and makes sure that downloaded code sees the proxy classes instead of the protected classes.


Did u know?

In computer science, an Access Control Matrix or Access Matrix is an abstract, formal security model of protection state in computer systems, that characterizes the rights of each subject with respect to every object in the system.

Self Assessment

Multiple choice questions:

1. What is system protection?

   (*a*) System protection is a feature that regularly creates and saves information about your computer's system files and settings.

   (*b*) System protection is a program that regularly creates and saves information about your computer's system files and settings.

   (*c*) System protection is a application that regularly creates and saves information about your computer's system files and settings.

   (*d*) None of these.

2. Access matrix is defined as ..... .

   (*a*) a model of protection mechanisms in computing systems.

   (*b*) is a matrix that shows the protection level across several domains.

   (*c*) work flow management.

   (*d*) All of the above.

3. Where we can implement the Access Matrix?

   (*a*) Network-wide capabilities.

   (*b*) Replicated access lists.

   (*c*) Application-Defined Objects.

   (*d*) All of the above.

8.3 Access Control

This section introduces concepts, common terms, and basic (popular) policies and models of access control. The contents of this section are referenced throughout the document.

Notes

Access control is concerned with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in the system. A given information technology (IT) infrastructure can implement access control systems in many places and at different levels. Operating systems use access control to protect files and directories. Database management systems DBMS apply access control to regulate access to tables and views. Most commercially available application systems implement access control, often independent of the operating systems and/or DBMSs on which they are installed.

The objectives of an access control system are often described in terms of protecting system resources against inappropriate or undesired user access. From a business perspective, this objective could just as well be described in terms of the optimal sharing of information. After all, the main objective of IT is to make information available to users and applications. A greater degree of sharing may get in the way of resource protection; in reality, a well-managed and effective access control system actually facilitates sharing. A sufficiently fine-grained access control mechanism can enable selective sharing of information where, in its absence, sharing may be considered too risky altogether.

8.3.1 Concepts

This section introduces some of the concepts that are commonly used in the access control research community and are also used throughout this document.

- **Object:** An entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are records, fields (in a database record), blocks, pages, segments, files, directories, directory trees, process, and programs, as well as processors, video displays, keyboards, clocks, printers, and network nodes. Devices such as electrical switches, disc drives, relays, and mechanical components connected to a computer system may also be included in the category of objects.
- **Subject:** An active entity, generally in the form of a person, process, or device that causes information to flow through objects (see below) or changes the system state.
- **Operation:** An active process invoked by a subject; for example, when an automatic teller machine (ATM) user enters a card and correct personal identification number (PIN), the control program operation on the user's behalf is a process, but the subject can initiate more than one operation-deposit, withdrawal, balance inquiry, etc.
- **Permission (privilege):** An authorization to perform some action on the system. In most computer security literature, the term permission refers to some combination of objects and operations. A particular operation used on two different objects represents two distinct permissions, and similarly, two different operations applied to a single object represent two distinct permissions. For example, a bank teller may have permissions to execute debit and credit operations on customer records through transactions, while an accountant may have of accounting data.
- **Access Control List (ACL):** A list associated with an object that specifies all the subjects that can access the object, along with their rights to the object. Each entry in the list is a pair (subject, set of rights). An ACL corresponds to a column of the access control matrix (described next). ACLs are frequently implemented directly or as an approximation in modern operating systems.
- **Access Control Matrix:** A table in which each row represents a subject, each column represents an object, and each entry is the set of access rights for that subject to that object. In general, the access control matrix is sparse—most subjects do not have access rights to most objects. Therefore, different representations have been proposed. The access control matrix can be represented as a list of triples, having the form <subject, rights, object>. Searching a large number of these triples is inefficient enough that this implementation is seldom used. Rather, the matrix is typically subdivided into columns (ACLs) or rows (capabilities).

Notes

- **Separation of Duty (SOD):** The principle that no user should be given enough privileges to misuse the system. For example, the person authorizing a paycheck should not also be the one who can prepare it. Separation of duties can be enforced either statically by defining conflicting roles (i.e., roles which cannot be executed by the same user) or dynamically by enforcing the control at access time. An example of dynamic separation of duty is the two-person rule. The first user to execute a two-person operation can be any authorized user, whereas the second user can be any authorized user different from the first. There are various types of SOD; an important one is a history-based SOD that regulates, for example, the same subject (role) cannot access the same object for a certain number of times.
- **Safety:** Measures that the access control configuration (e.g., access control mechanism or model) will not result in the leakage of permissions to an unauthorized principal. Thus, a configuration is said to be safe if no permission can be leaked to an unauthorized or unintended principal.
- **Domain and Type Enforcement:** The grouping of processes into domains, and objects into types, such that access operations (such as read, write, execute, and create) are restricted from domains to types and between domains. A process belongs to one domain at any given time and transits to other domains by sending signals or executing a file in a new domain.



Task Give the way of protection used in operating system.

8.3.2 Policies, Models and Mechanisms

When planning an access control system, three abstractions of controls should be considered—access control policies, models, and mechanisms. Access control policies are high-level requirements that specify how access is managed and who, under what circumstances, may access what information. While access control policies can be application-specific and thus taken into consideration by the application vendor, policies are just likely to pertain to user actions within the context of an organizational unit or across organizational boundaries. For instance, policies may pertain to resource usage within or across organizational units or may be based on need-to-know, competence, authority, obligation, or conflict-of-interest factors. Such policies may span multiple computing platforms and applications.

At a high level, access control policies are enforced through a mechanism that translates a user's access request, often in terms of a structure that a system provides. There are a wide variety of structures; for example, a simple table lookup can be performed to grant or deny access. Although no well-accepted standard yet exists for determining their policy support, some access control mechanisms are direct implementations of formal access control policy concepts.

Rather than attempting to evaluate and analyze access control systems exclusively at the mechanism level, security models are usually written to describe the security properties of an access control system. A model is a formal presentation of the security policy enforced by the system and is useful for proving theoretical limitations of a system. Access control models are of general interest to both users and vendors. They bridge the rather wide gap in abstraction between policy and mechanism. Access control mechanisms can be designed to adhere to the properties of the model. Users see an access control model as an unambiguous and precise expression of requirements. Vendors and system developers see access control models as design and implementation requirements. On one extreme, an access control model may be rigid in its implementation of a single policy. On the other extreme, a security model will allow for the expression and enforcement of a wide variety of policies and policy classes. As stated previously, the focus of this document is on the practical side of the access control system; detailed descriptions of access control models are not included in this publication.

Notes

Generating a list of access control policies is of limited value, since business objectives, tolerance for risk, corporate culture, and the regulatory responsibilities that influence policy differ from enterprise to enterprise, and even from organizational unit to organizational unit. The access control policies within a hospital may pertain to privacy and competency (e.g., only doctors and nurse practitioners may prescribe medication), and hospital policies will differ greatly from those of a military system or a financial institution. Even within a specific business domain, policy will differ from institution to institution. Furthermore, access control policies are dynamic in nature, in that they are likely to change over time in reflection of ever-evolving business factors, government regulations, and environmental conditions. There are several well-known access control policies, which can be categorized as discretionary or non-discretionary. Typically, discretionary access control policies are associated with identity-based access control, and non-discretionary access controls are associated with rule-based controls (for example, mandatory security policy).

8.3.3 Discretionary Access Control (DAC)

DAC leaves a certain amount of access control to the discretion of the object's owner or anyone else who is authorized to control the object's access. For example, it is generally used to limit a user's access to a file; it is the owner of the file who controls other users' accesses to the file. Only those users specified by the owner may have some combination of read, write, execute, and other permissions to the file. DAC policy tends to be very flexible and is widely used in the commercial and government sectors. However, DAC is known to be inherently weak for two reasons. First, granting read access is transitive; for example, when Ann grants Bob read access to a file, nothing stops Bob from copying the contents of Ann's file to an object that Bob controls. Bob may now grant any other user access to the copy of Ann's file without Ann's knowledge. Second, DAC policy is vulnerable to Trojan horse attacks. Because programs inherit the identity of the invoking user, Bob may, for example, write a program for Ann that, on the surface, performs some useful function, while at the same time destroys the contents of Ann's files. When investigating the problem, the audit files would indicate that Ann destroyed her own files. Thus, formally, the drawbacks of DAC are as follows:

- Information can be copied from one object to another; therefore, there is no real assurance on the flow of information in a system.
- No restrictions apply to the usage of information when the user has received it.
- The privileges for accessing objects are decided by the owner of the object, rather than through a system-wide policy that reflects the organization's security requirements.

ACLs and owner/group/other access control mechanisms are by far the most common mechanism for implementing DAC policies. Other mechanisms, even though not designed with DAC in mind, may have the capabilities to implement a DAC policy.

8.3.4 Non-Discretionary Access Control

In general, all access control policies other than DAC are grouped in the category of non-discretionary access control (NDAC). As the name implies, policies in this category have rules that are not established at the discretion of the user. Non-discretionary policies establish controls that cannot be changed by users, but only through administrative action.

Separation of duty (SOD) policy can be used to enforce constraints on the assignment of users to roles or tasks. An example of such a static constraint is the requirement that two roles to be mutually exclusive; if one role requests expenditures and another approves them, the organization may prohibit the same user from being assigned to both roles. So, membership in one role may prevent the user from being a member of one or more other roles, depending on the SOD rules, such as Work Flow and Role-Based Access Control (see the following sections). Another example is a history-based SOD policy that regulates, for example, whether the same subject (role) can

Notes

access the same object a certain number of times. Three popular non-discretionary access control policies are discussed in this section.

8.3.5 Mandatory Access Control (MAC)

Mandatory access control (MAC) policy means that access control policy decisions are made by a central authority, not by the individual owner of an object, and the owner cannot change access rights. An example of MAC occurs in military security, where an individual data owner can neither decide who has a Top Secret Clearance, nor the owner can change the classification of an object from Top Secret to Secret. MAC is the frequently mentioned NDAC policy.

The need for a MAC mechanism arises when the security policy of a system dictates that:

1. Protection decisions must not be decided by the object owner.
2. The system must enforce the protection decisions (i.e., the system enforces the security policy over the wishes or intentions of the object owner).

Usually a labeling mechanism and a set of interfaces are used to determine access based on the MAC policy; for example, a user who is running a process at the Secret classification should not be allowed to read a file with a label of Top Secret. This is known as the "simple security rule," or "no read up." Conversely, a user who is running a process with a label of Secret should not be allowed to write to a file with a label of Confidential. This rule is called the "*-property" (pronounced "star property") or "no write down." The *-property is required to maintain system security in an automated environment. A variation on this rule called the "strict *-property" requires that information can be written at, but not above, the subject's clearance level. Multilevel security models such as the Bell-La Padula Confidentiality and Biba Integrity models are used to formally specify this kind of MAC policy. However, information can pass through a covert channel in MAC, where information of a higher security class is deduced by inference such as assembling and intelligently combining information of a lower security class. Popular mechanisms used in implementing MAC policies are demonstrated.

8.3.6 Role-based Access Control

Although RBAC is technically a form of non-discretionary access control, recent computer security texts often list RBAC as one of the three primary access control policies (the others are DAC and MAC). In RBAC, access decisions are based on the roles that individual users have as part of an organization. Users take on assigned roles (such as doctor, nurse, teller, or manager). Access rights are grouped by role name, and the use of resources is restricted to individuals authorized to assume the associated role. For example, within a hospital system, the role of doctor can include operations to perform a diagnosis, prescribe medication, and order laboratory tests; the role of researcher can be limited to gathering anonymous clinical information for studies. The use of roles to control access can be an effective means for developing and enforcing enterprise-specific security policies and for streamlining the security management process. Under RBAC, users are granted membership into roles based on their competencies and responsibilities in the organization. The operations that a user is permitted to perform are based on the user's role. User membership into roles can be revoked easily and new memberships established as job assignments dictate. Role associations can be established when new operations are instituted, and old operations can be deleted as organizational functions change and evolve. This simplifies the administration and management of privileges; roles can be updated without updating the privileges for every user on an individual basis.

When a user is associated with a role, the user can be given no more privileges than is necessary to perform the job; since many of the responsibilities overlap between job categories, maximum privilege for each job category could cause unauthorized access. This concept of least privilege

Notes

requires identifying the user's job functions, determining the minimum set of privileges required to perform those functions, and restricting the user to a domain with those privileges and nothing more. In less precisely controlled systems, least privilege is often difficult or costly to achieve because it is difficult to tailor access based on various attributes or constraints. Role hierarchies can be established to provide for the natural structure of an enterprise. A role hierarchy defines the roles that have unique attributes and that may contain other roles; that is, one role may implicitly include the operations that are associated with another role.

8.3.7 Temporal Constraints

Temporal constraints are formal statements of access policies that involve time-based restrictions on access to resources; they are required in several application scenarios. In some applications, temporal constraints may be required to limit resource use. In other types of applications, they may be required for controlling time-sensitive activities. It is the time-based constraints (in addition to other constraints like workflow precedence relationships) that must be evaluated for generating dynamic authorizations during workflow execution time. Temporal constraints may also be required in non-workflow environments as well. For example, in a commercial banking enterprise, an employee should be able to assume the role of a teller (to perform transactions on customer accounts) only during designated banking hours (such as 9 a.m. to 2 p.m., Monday through Friday, and 9 a.m. to 12 p.m. on Saturday). To meet this requirement, it is necessary to specify temporal constraints that limit role availability and activation capability only to those designated banking hours.

Popular access control policies related to temporal constraints are the history-based access control policies, which are not supported by any standard access control mechanism but have practical application in many business operations such as task transactions and separation of conflicts-of-interests. History-based access control is defined in terms of subjects and events where the events of the system are specified as the object access operations associated with activity at a particular security level. This assures that the security policy is defined in terms of the sequence of events over time, and that the security policy decides which events of the system are permitted to ensure that information does not "flow" in an unauthorized manner. Popular history-based access control policies are Workflow and Chinese Wall, which are described below.

8.3.8 Workflow

Based on the definition provided by the Workflow Management Coalition (WFMC), an international organization of workflow vendors, users, and research groups, a workflow is a representation of an organizational or business process in which "documents, information, or tasks are passed from one participant to another in such a way that is governed by rules or procedures." A workflow separates the various activities of a given organizational process into a set of well-defined tasks. Hence, typically, a workflow (often synonymous with a process) is specified as a set of tasks and a set of dependencies among the tasks, and the sequencing of these tasks is important. The various tasks in a workflow are usually carried out by several users in accordance with organizational rules relevant to the process represented by the workflow.

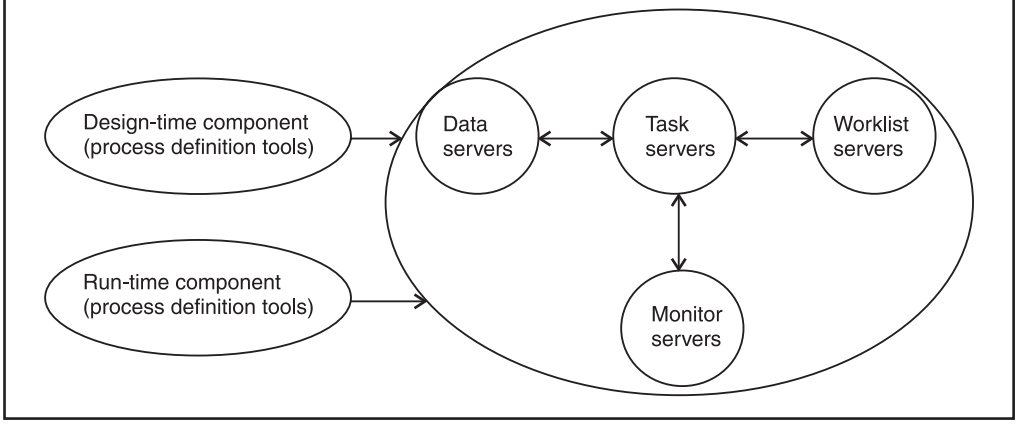
The representation of a business process using a workflow involves a number of organizational rules or policies. An important class of organization policies is the organization's security policies. Within the realm of security policies, access control policies play a key role, and hence defining and enforcing access control requirements becomes a key function of a Workflow Management System (WFMS).

Figure 8.2 presents a schematic diagram of the overall architecture of a WFMS, which consists of two main components—design-time and run-time. The design-time component consists of a set of tools (called the process definition tools) that are used for defining and modeling the

Notes

business processes and their constituent tasks. A process definition consists of a process name (e.g., purchase order process), the definition of various tasks within the process (e.g., purchase order approval task), and a set of business rules associated with the process (e.g., task sequence or data flow among tasks). The run-time component of a WFMS (also called a workflow engine) consists of a set of servers that interpret the process definition and create and maintain process instances. Task instances associated with each process instance are also created (based on process definition). The list of instantiated tasks pending to be executed is presented to the user (for his or her action) through a work list server. The tasks themselves are executed in task servers. Data servers act as repositories of data that are needed by tasks. In addition, there are monitor servers that maintain the execution history for various process or task instances to facilitate run-time access control decisions.

Figure 8.2: Components of the Workflow Management System



The goal of the Workflow policy is to maintain consistency between the internal data and external (users') expectations of that data. Note that many individual process instances may be operational during process enactment; each needs to be associated with a specific set of data relevant to that individual process instance.

8.3.9 Chinese Wall

Brewer and Nash identified the Chinese Wall policy to address conflict-of-interest issues related to consulting activities within banking and other financial disciplines. Like WFMS, the Chinese Wall policy is application-specific as it applies to a narrow set of activities that are tied to specific business transactions. For example, consultants naturally are given access to proprietary information to provide a service for their clients. When a consultant gains knowledge amounting to insider information, that knowledge can be used outside the company, thus undermining the competitive advantage of one or both institutions, or used for personal profit. The stated objective of the Chinese Wall policy is to prevent illicit flows of information that can result in conflicts of interest.

The Chinese Wall policy is a commercially inspired confidentiality policy, whereas most other commercial policies focus on integrity. The access permissions change dynamically—as a subject accesses some objects, other objects that would previously have been accessible are now denied. For example, the Chinese Wall policy is used where company-sensitive information is categorized into mutually disjoint conflict-of-interest categories (COI). Each company belongs to only one COI, and each COI has two or more member companies. The membership within a COI includes like companies, whereby a consultant obtaining sensitive information regarding one company would risk a conflict of interest if he or she were to obtain sensitive information concerning

Notes

another company. Several COIs may coexist. For example, COI1 may pertain to banks, while COI2 may pertain to energy companies. The Chinese Wall policy aims to prevent a consultant from reading information for more than one company in any given COI.

There are several observations that we can make regarding this policy with respect to read operations. First, as long as a consultant has not read information belonging to any institution, the consultant is not yet bound by the policy and is free to read any sensitive information of any institution. Note that although a consultant may be free to read sensitive information under the Chinese Wall policy, he or she may be restricted from reading sensitive information with respect to another policy, such as a MAC policy. Second, once a consultant has read sensitive information of bank A, the consultant is prohibited from reading sensitive information belonging to any other bank included in the COI of which bank A is a member. Third, all consultants are free to read all the public information of all institutions.

In the history-based access control policies, previous access events are used as one of the decision factors for the next access authorization; the policies require sophisticated historical system state control for tracking and maintaining of historical events. For example, the Chinese Wall policy is simple and easy to describe; however, its implementation and deployment are less straightforward.

8.4 Revocation of Access Rights

These are to be performed:

- Immediately or after a delay
- For all users or a selective group
- All rights or partial rights
- Temporary or permanent

Remove access rights for an object — given to a user/domain. Easy with global table or access list — search list for object and remove entry. Capabilities are distributed throughout the system — must be found and destroyed — difficult:

- **Expiry Time:** Capabilities expire after a time and new must be requested — this is refused if rights have been revoked.
- **Back Pointers:** Objects maintain pointers to all capabilities issued — costly to implement, particularly if capabilities are passed around as parameters.
- **Indirect Capabilities:** Capability points to table entry which points to object — Invalidate entry to revoke capability — No selective revocation.
- **Keys:** Capability contains encrypted key checked by object — change key in object to revoke capability — No selective revocation.

8.5 Capability Based System

Capability based systems were first described in the literature in the mid-1960's. Their informal descriptions are typically based upon the notion that a capability is equivalent to a ''ticket,'' in the sense that possession of the ticket allows the possessing process access to the object described in the capability, provided that the access mode is compatible with the ''access rights'' stored within the capability. Whether a computer system based upon capabilities can provably enforce the DoD security policy has been a matter of discussion for some time. Boebert has argued that an ''unmodified'' capability machine must be incapable of enforcing the property defined by

Notes

Bell and LaPadula. Since the property is an essential ingredient of the most widely used model of DoD security policy, this result implied that an ''unmodified'' capability machine cannot meet the DoD requirements. Boebert's discussion introduces the undefined term ''unmodified capability machine.'' In this paper we describe several classes of capability machine designs for managing access control information and show some that some classes cannot meet the DoD requirements but others can. We thereby circumvent a debate about the meaning of the term ''unmodified capability machine.''

This paper begins with the brief definitions of the basic notions concerning capabilities and capability machines. We next consider the sequence of events a capability may undergo between the creation of a segment and an access to that segment, and we discuss strategies for controlling access rights in this context. A design taxonomy is developed to describe these options. Finally, we show some classes in the design taxonomy that are not compatible with the DoD security policy.

8.5.1 Basic Notions

The basic notions center on the properties of data and processes, the descriptions of these entities, the mechanisms that control access, and the policies that define ''correct'' access limitations. We start with data, processes, and capabilities.

Definition: A segment is a group of data possessing identical security attributes. Additionally, the segment may contain a set of capabilities possessing identical security attributes; these security attributes need not necessarily be the same as the attributes of the data contained within the segment. A segment that can hold only capabilities is called a capability list or c-list.

Definition: A capability is an object describing a segment, and possibly containing access rights or other access control information, as described below. Note that if the capability contains access rights, it must be distinguished from data to prevent unauthorized changes to the access rights, which, if permitted, would defeat all attempts to limit access based upon the access rights present in accessible capabilities. Capabilities can be distinguished from data either by tagging or by limiting their locations to distinguished segments or portions of segments that may only contain capabilities.

Definition: A segment possesses certain data security attributes, including, but not limited to, a security level and an access control list. In addition, the segment may possess a separate set of capability security attributes describing any capabilities stored within the segment. All data within the segment possess the data security attributes associated with the segment. All capabilities within the segment possess the capability security attributes associated with the segment.

Definition: A process is the execution of a program on behalf of a user logged in at a certain security level.

Definition: The security attributes of a process include a security level and the identity of the user on whose behalf the program is executed. A process may have other attributes, such as its domain of execution, in certain designs.

Definition: A reference monitor is a mechanism for checking each attempted access by a process to an item within a segment for conformance with the access modes permitted for the process to that segment. A process can attempt access to a segment only via a capability that has been prepared for access (e.g. by placing it in a capability register). Capabilities prepared for access are not shared among processes.

Definition: A security policy is a set of rules for determining the maximum permissible access rights for a particular process to a particular segment, given the attributes of both the process and the segment.

Definition: The DoD mandatory security policy limits the access rights to a segment based upon a comparison between the security level of the segment and the security level of the accessing process. Write is allowed if the level of the segment dominates the level of the process, read if

Notes

the level of the process dominates the level of the segment. Domination is a reflexive relation, so that both read and write are permitted if the two levels are identical. (The DoD policy).



A capability is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights.



Describe the DoD policy in term of Protection.



OS Rings in System Protection

In practice, this means that formal mechanisms must be in place to segregate the trusted operating system from untrusted user programs. The most reliable way to accomplish this is in hardware. If the segregation occurs in software, a software failure (such as a buffer overflow) can be used to compromise the system. The first system to support rings in hardware was the MULTICS time-sharing system in the 1960s, which included eight rings. This approach of hardware-enforced rings has been almost universally adopted by later architectures.

The most common CPU architecture in use today is the x86 compatible architecture. Beginning with the 80286 chipset, the x86 family has provided two main methods of addressing memory—real mode and protected mode. Real mode, limited to a single megabyte of memory, quickly became obsolete. Protected mode provided numerous new features to support multitasking. These included segmenting processes, so that they could no longer write outside their address space, along with hardware support for virtual memory and task switching.

In the x86 family, protected mode uses four priority levels, numbered 0 to 3. System memory is divided into segments, and each segment is assigned a priority level. The processor uses the priority level to determine what can and cannot be done with code or data within a segment. The term rings comes from the MULTICS system, where privilege levels were visualized as a set of concentric rings. Ring 0 is considered to be the innermost ring, with total control of the processor. Ring 3, the outermost ring, is provided only with restricted access.

Windows, Linux, and most Unix variants all use rings, although they have generally dropped the four-ring structure and instead adopted a two-layer approach that uses only rings 0 and 3. Security mechanisms in the hardware enforce restrictions on ring 3 by limiting code access to segments, paging, and input/output. If a user program running in ring 3 tries to address memory outside of its segments, hardware interrupt stops code execution. Some assembly language instructions are not even available for execution outside of Ring 0.

Questions:

1. Which was the first system to use rings as hardware for system protection?
2. How rings can be used in system protection?

Self Assessment

Notes

Multiple choice questions:

4. What is the use of Chinese Wall?
   (*a*) To address conflict-of-interest issues related to consulting activities within banking and other financial disciplines.
   (*b*) To address conflict-of-interest issues related to consulting activities within college and other Institutional disciplines.
   (*c*) To address conflict of-interest issues related to consulting activities within government offices.
   (*d*) All of the above.
5. What is Access Control List?
   (*a*) A list associated with an object that specifies all the objects that can access the subject, along with their rights to the object.
   (*b*) A list associated with an object that specifies all the subjects that can access the object, along with their rights to the object.
   (*c*) A list associated with a subject that specifies all the subjects that can access the object, along with their rights to the object.
   (*d*) None of the above.
6. What is Discretionary Access Control (DAC)?
   (*a*) DAC leaves a certain amount of access control to the discretion of the object's owner or anyone else who is not authorized to control the object's access.
   (*b*) DAC leaves a certain amount of access matrix to the discretion of the object's owner or anyone else who is authorized to control the object's access.
   (*c*) DAC leaves a certain amount of access control to the discretion of the object's owner or anyone else who is authorized to control the object's access.
   (*d*) All of the above.

Fill in the blanks:

7. ..... control is concerned with determining the allowed activities of legitimate uses.
8. ..... is an entity that contains or receives information in the access control research community.

8.6 Summary

- The processes in an operating system must be protected from one another's activities.
- Implementation of protection in an OS generally involves three factors.
- Access matrices are widely used to hold a symbolic representation of the set of rights available to the subjects for access to objects.
- Access control is concerned with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in the system.
- Capability systems were first described in the literature in the mid-1960's.

Notes

8.7 Keywords

Access Control Mechanisms: The manner by which the operating system enforces the access control policy.

Access Control Policies: Access control policy defined "whose data is to be protected from whom".

DoD Policy: This publication, DoD 5200.28-STD, "Department of Defense Trusted Computer System Evaluation Criteria," is issued under the authority of an in accordance with DoD Directive 5200.28, "Security Requirements for Automatic Data Processing (ADP) Systems," and in furtherance of responsibilities assigned by DoD Directive 52l5.

Workflow Management Coalition (WFMC): Work Flow Management is a fast evolving technology which is increasingly being exploited by businesses in a variety of industries.

System Protection: A model of protection mechanisms in computing systems is presented and its appropriateness is argued. The "safety" problem for protection systems under this model is to determine in a given situation whether a subject can acquire a particular right to an object. In restricted cases, it can be shown that this problem is decidable, i.e. there is an algorithm to determine whether a system in a particular configuration is safe. In general, and under surprisingly weak assumptions, it cannot be decided if a situation is safe. Various implications of this fact are discussed.



Lab Exercise

1. C program for file permissions.
2. Give the steps for protection of the operating system.

8.8 Review Questions

1. The access-control matrix could be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?
2. Consider a system in which "computer games" can be played by students only between 10 p.m. and 6 a.m., by faculty members between 5 p.m. and 8 a.m., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.
3. What hardware features are needed for efficient capability manipulation? Can these be used for memory protection?
4. Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with the objects.
5. Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with the domains.
6. Explain why a capability-based system such as Hydra provides greater flexibility than the ring protection scheme in enforcing protection policies.
7. What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?
8. How are the access-matrix facility and the role-based access-control facility similar? How do they differ?

Notes

9. How does the principle of least privilege aid in the creation of protection systems?
10. How can systems that implement the principle of least privilege still have protection failures that lead to security violations?
11. Discuss the strengths and weaknesses of capability based system.
12. What is the goal of protection in Operating System?
13. Discuss about the DoD policy in system protection.
14. What is Workflow Management Coalition (WFMC)?
15. Explain the Mandatory Access Control (MAC).

Answers to Self Assessment

1. (*a*) 2. (*d*) 3. (*d*) 4. (*a*) 5. (*b*)
6. (*c*) 7. Access 8. Object

8.9 Further Readings



Books

Operating Systems, by Andrew Tanenbaum, Albert S. Woodhull.
Operating Systems, by Stuart E. Madnick, John J. Donovan.



Online link

wiley.com/coolege.silberschatz

Notes

Unit 9: System Security

CONTENTS

Objectives

Introduction

9.1 Overview

9.1.1 Protection

9.1.2 Access Control Lists

9.1.3 Capabilities

9.1.4 Mandatory Access Control

9.1.5 Cryptographic Access Control

9.2 Security Problem

9.3 Program Threats

9.3.1 Types of Threats

9.3.2 System Threats

9.3.3 Network Threats

9.3.4 Types of Network Threats

9.4 Cryptography

9.4.1 The Purpose of Cryptography

9.4.2 Types of Cryptographic Algorithms

9.5 User Authentication

9.5.1 Implementation of Defenses

9.6 Firewall

9.6.1 First Generation: Packet Filters

9.6.2 Second Generation: Application Layer

9.6.3 Third Generation: "Stateful" Filters

9.6.4 Subsequent Developments

9.6.5 Types

9.7 Summary

9.8 Keywords

9.9 Review Questions

9.10 Further Readings

Notes

Objectives

After studying this unit, you will be able:

- Explain security problem
- Understand program threats
- Explain system threats
- Understand network threats
- Discuss cryptography
- Explain user authentication
- Understand implementation of defenses
- Discuss firewall

Introduction

It is unnecessary to say that all operating systems are not equal in every sphere. None of the most accepted operating systems now-a-day are developed keeping safe and sound electronic business in mind. From the moment the computer is booted (after loading the BIOS), the interaction with the operating system starts. This essential portion of software describes what can be done with the system of the computer and how it is to be done. Whether the interaction is with the file system or while chatting with somebody with the help of an instant messenger line up, the operating system works at the back of the pictures to provide with a perfect experience as it understands the procedures and changes them into those things that the computer can process. Even as operating systems differ on a lot of levels, the most ordinary operating systems offer much more than an easy crossing point between the user and the machine.

These include many programs with the intention to provide the user with many extras, starting from the straightforward screen savers to multifaceted file-encryption plans. However, it's vital to know that these programs are add-ons which are added on to the OS and are not essential for the computer to function. A lot of users become closely familiar with the operating system's frills (such as the Solitaire), but do not remember about the security parameters that are incorporated to help the user preserve a secure and dependable operating setting. And therefore, a lot of information systems continue living in an insecure condition that keeps the system at the danger of a virus infection or at the total compromise by a hacker. From creation a protected home network to creating well-built passwords, it is very vital to know the particulars of using an operating system in a secure and safe and sound mode. In today's world, where the computer is an essential gadget it is foolish to set up a computer devoid of the security system. It is only one virus or Trojan horse that is enough to form a concurrent consequence of contaminated computers and compromise systems.

UNIX which is the eldest and the mainly used networking operating system of today's world has got the benefit of having been patched and hacked by crackers and hackers for many years. Amongst all other UNIX derivatives, the most popular derivatives is Linux, which was developed by Linus Torvalds and now being maintained by a lot many of volunteers and a lot of software companies. But this Linux still has got errors which are being discovered in a day-to-day basis. It is tremendously significant to keep an eye on these incidences and use the essential patches

Notes

of the updates as soon as they are accessible. Microsoft's Windows family platform has got an extraordinary expansion as a server and client platform. Whether it is installed in the thousands of home PCs, in the Internet server or on corporate LANs, its fame has made it more prone for breakage by many hackers. Bodily security is becoming more and more common through out the world as novel and graver terrorizations are emerging. Physical security of public and possessions comes in course of time to most human; still the lost in the mix up is the prime necessitate for securing the data. Over and over again, this data contains responsive information that is sought after by people against the law.

Encryption and Privacy Resource

- It acts as a source for learning about a range of encryption schemes.
- This includes privacy fortification tips and security fundamentals.

NIST Vulnerability and Threat Portal

- It is an absolute portal for recent attacks or incidents, bugs, advisories, etc. kept up by the National Institute of Standards and Technology.

PC Talk - Security

- This is the latest news from the field of IT security.

CERT/CCC Current Activity

- Central depository for high force security vulnerabilities and incidents are being informed to the CERT or CCC.

The two best security oriented operating systems are:

Back Track

- This is an Innovative Penetration Testing which comes live from Linux distribution.
- This brilliant bootable live-CD has generated as a result of from the merger of Whax and Auditor.
- It shows off an enormous diversity of Security and Forensics apparatus and provides an affluent maturity of the environment.
- The user modularity is highlighted so the allocation can be customized with no trouble by the user to incorporate private scripts, extra tools, customized kernels, etc.

Knoppix

- This is designed as the general purpose live bootable system to be taken on CD or DVD. It consists of a diplomatic set of involuntary hardware revealing, GNU/Linux software, and hold up for a lot of sound cards, graphics cards, USB and SCSI appliances and other peripherals.
- KNOPPIX can also be used as a creative Linux system as the educational CD, desktop, rescue system.

Notes

9.1 Overview

Security systems generally implement a useful subset of a 3-dimentional matrix with dimensions of actor, object, and action. Implementing the full matrix is never done for both performance and usability reasons.

Mainstream operating system security is primarily based on access control lists. For each object, we can maintain a list of actors which may manipulate it.

Capability-based systems, not to be confused with the systems that refer to special administrative abilities as capabilities, flip things around the other way. For each actor, they maintain a list of objects upon which the actor may act.

Mainstream operating system security is discretionary. That is, an object owner may decide who else has access to the object.

Mandatory access control is non-discretionary. A set of rules acts to enforce security, setting up permissions that users are unable to override. This stops insiders from being effective spies, selling designs to the competition, sharing medical records with the press, running spyware, and making many types of user errors.

None of the above methods excludes any other of the above methods. DG/UX has capabilities, discretionary ACLs, and mandatory access control. Linux provides discretionary ACLs and several choices for mandatory access control. File descriptors, available on all UNIX-like systems, can serve as capabilities.

9.1.1 Protection

The first security attempts were protection schemes, which controlled the access of programs to sensitive areas like the 0 page, where the software interrupts for the operating system calls were usually stored for efficient calling, or the operating system areas, where the operating system code was kept while it was running. This capability was required for time-sharing because neophyte programmers sometimes overwrote the operating system shutting down the whole computer because they didn't understand the addressing system. CPU's like the Z80 began to be designed to set aside "System" areas so that they couldn't be overwritten by mistake.

One way of doing this was to create a separate name space for the system and control access to it. Often the documentation of how to access the protected mode was scant, or missing from popular books, as an attempt to secure it by obscurity. This turned out to be relatively useless since it meant that an underground market for information was created, and only the crackers knew for sure how they were breaking into the operating systems.

Some CPU's set up complex software interrupts as gateways between the protected mode memory and the user areas. Each user area had its own software interrupt area, that did a system call to the protected system area because the interrupt area was within the user space, the interrupt vectors could be overwritten with impunity, without affecting any other programs use of the operating system. This was a useful mechanism because peripheral drivers tended to supply their own interrupt service routines for specific peripherals.

An application like Windows could create a virtualized version of itself, and modify the interrupts within that one virtual version, without affecting the rest of the operating system. This is usually what the difference is, between 286 and 386 protective mode, the Virtual copy of windows with the ability to modify its own interrupt vectors and not affect the rest of the machine. Up to now until the 386 protected mode, all programs shared the same interrupt vectors and one

Notes

programs mishandling or malware sabotage of the interrupts could trigger the collapse of the whole system.



Task Show or explain how to use firewall insecurity of operating system.

9.1.2 Access Control Lists

Traditional UNIX access control is a type of discretionary ACL. The number of possible actions are grouped into four categories—read, write, execute, and special operations normally reserved for an object owner. The list of actors associated with each object is rather restricted, simplifying both the implementation and the user experience. There are three actors listed—owner, group, and other. An actor gets the permissions of the first of those that it qualifies as. The owner always gets special operations that the others do not get. The group is an indirect reference to a list of actors specified elsewhere, and thus is a form of compression. The "other" is just that, all other actors. Through the creation of groups, traditional UNIX access control can provide a great deal of power. The creation of groups is normally limited however, often being an administrative action that requires human approval.

The action categories may be more or less fine-grained. NetWare uses read, write, create, erase, modify, file scan, access control, and supervisor. A system may split the normal "write" category into overwrite and append. A system may lack an "execute" category, instead simply requiring read access to execute a file.

Many systems allow for somewhat arbitrary lists of actors to be associated with each object. This includes Windows, modern UNIX-like systems including Linux, and Netware. The list might support a dozen entries or a few hundred entries, as determined to be a good compromise between performance and control.

An interesting innovation in ACLs has been hierarchical actors. For example, the VST a OS used decimal, dot-delimited actor identifiers. In this scheme, a userid is a series of decimal numbers seperated by dots (eg, 1.85.23.323.888) and if a user possesses a userid which corresponds to the object's userid up to its end, then the user owns the object. So for example, a user possesing userid 1.85.23 would own any object assigned to 1.85.23.323.888 whereas the object itself (whether another user or program) wouldn't be able to access other objects owned by 1.85.23. This scheme allows for the dynamic creation of a hierarchy of users and subusers. A weaker form of this, compatible with the vast body of POSIX software, can be had by providing a mechanism for users of a UNIX-like system to create and control groups.



Did u know?

An access control list (ACL) is a table that tells a computer operating system which access rights each user has to a particular system object, such as a file directory or individual file.

9.1.3 Capabilities

Capabilities are unforgeable references to objects that let their holder access a well-defined subset of operations defined on that object.

Notes

There are many different things which may serve as capabilities. There are human-readable capabilities called "passwords" which people make use of. There are flat, low-level, centralized capabilities implemented by experimental OS kernels. There are tickets granted by Kerberos servers. There are UNIX-style file descriptors, which may be passed from process to process to provide access.

Operating systems such as KeyKOS and EROS have used flat, low-level and highly centralized capabilities, leaving the implementation of human-useful capabilities to higher level layers.

Capabilities in such schemes are flat because they can not be interrogated. You can not ask such a capability about, say, when it was created or how many times it has been accessed. None of such capabilities possess internal state which can be mutated. Typically, one can create a new downgraded capability from a capability one possesses, but one can not downgrade or change in any way the original capability itself. These capabilities are not objects but primitive data types.

Capabilities in such schemes are low level because they do not offer high-level security abstractions. In KeyKOS, capabilities were exactly of two types, read and read+write, though KeyKOS capabilities could contain additional information to implement service levels and other such levels. Capabilities in such schemes do not naturally aggregate other capabilities (a very simple and highly useful security abstraction), one must artificially create an object in between whose sole job is to aggregate them. Note that certain security abstractions (like dynamic permission inheritance) are practically unimplementable on the top of a low-level system.

Capabilities in such schemes are centralized because the kernel, and only the kernel, manages all capabilities. Rather than every module exporting its own capabilities, the only capabilities there are in the kernel. This is actually a consequence of using low-level capabilities. If capabilities were sufficiently high-level so as to provide message-passing services, then it would be feasible to force every module to use secondary capabilities that all route through a single protected kernel capability. Instead, each module (called a Domain) in KeyKOS had access to a maximum of exactly 16 capabilities in special slots protected by the kernel and accessible only through special calls.

9.1.4 Mandatory Access Control

Traditional mandatory access control or MAC is modeled on the scheme used by the U.S. government for handling classified information. More specifically, this is mandatory sensitivity control or MSEN. There are levels of security such as TOP SECRET, SECRET, CLASSIFIED, and UNCLASSIFIED. There are categories based on need-to-know or special program access. An actor with SECRET access is prohibited from reading TOP SECRET data, but might be permitted to write to it. (there may be an additional discretionary ACL which prevents this though) One may be able to read from lower security levels and write to higher security levels. There may be a concept of contamination, allowing an actor with SECRET access to write UNCLASSIFIED data as long as no CLASSIFIED or SECRET data has been read.

Mandatory integrity control (MINT) works the other way, preventing pristine objects from being contaminated with junk. In this case, all actors are permitted to read the most trusted data. An actor that tries to execute something downloaded from an untrusted source will either be stopped or will lose the ability to write to more trusted objects.

More modern systems combine the functions of MSEN and MINT into a state-transition model with roles. As actors do things, their state may change. In a UNIX-like system with role-based access control, the execution of a new executable is a particularly important point at which state transitions happen. Role-based security is particularly useful for enforcing privacy regulations such as HIPPA (U.S. medical info law) and the EU privacy laws. Two popular implementations of this are SE Linux and RSBAC, both available for Linux.

Notes


Did u know?

In computer security, mandatory access control (MAC) refers to a type of access control by which the operating system constrains the ability of a subject or initiator to access or generally perform some sort of operation on an object or target.

9.1.5 Cryptographic Access Control

The advent of International Networking spread the security concerns outside the usual realm of the operating system, to the data that was moving to and from the system over the network. The ability to mimic a valid data packet meant that data could be changed enroute simply by rerouting the valid packet, and replacing it with an invalid packet. As a result, security had to be spread not only to the local system, but also to all critical correspondence between systems. At first such mechanisms were implemented by sending digests of the original data as part of the data stream, under the assumption that a changed packet would not fit the original digest. However it was found that digests were not cryptographically secure and could be fooled into thinking that data was valid when it was not. As well, sending data in clear meant that someone in an intervening system could read the packet with a packet sniffer, and learn what the information was.

The idea when this was determined became to cryptographically protect the data, which would at least, it was hoped, slow down the reading of the mail, and cryptographically secure the digest, so that it could not be fooled as easily.

9.2 Security Problem

Even casually computer savvy users these days know about beware of security threats on the Internet. They know that the online universe is a crawl with computer viruses, worms, Trojan horses and other malicious bits of code, and if they are prudent, they have equipped their computers with up-to-date anti-virus and firewall software for repelling these invaders. They are leery of unsolicited e-mail attachments, and careful about the websites they visit. They have probably heard about (or experienced) "denial of service" attacks in which malicious hackers direct thousands of computers to bombard a company's servers with requests to shut them down. They probably even know not to fall for "phishing" scams in which hyperlinks take users to phony sites posing as legitimate banks and credit card companies for the purpose of stealing passwords and account information.

What few in the public realize, however, is that the Internet is vulnerable to much deeper levels of fraud-ones that exploit fundamental security gaps in the network protocols themselves. These attacks, often called "pharming," are all but impossible for individuals to guard against or even detect. They represent a growing threat to personal, corporate and national security that the federal government needs to address urgently.

Consider, for example, the defenselessness of the domain name system (DNS), the Internet's version of "411 information". When you type a "www." style name into your browser software, the browser converts it into an IP address, a string of digits that is the equivalent to a phone number. It gets the IP address by contacting a local name server, typically operated by your Internet service provider. Unlike telephone numbers, however, which are often valid for several years, IP addresses change frequently and so the IP address comes with an expiration date, known as a "time to live" (or TTL). On the Internet, TTLs are typically measured in seconds, hours or days, even if the associated IP address does not change that often. If a local name server receives a request for an "expired" DNS name, it in turn queries a hierarchy of other servers, keying its request to two 16-bit identification codes—one for a transaction ID and other for a port number. Unfortunately, the port number is often predictable, and so it becomes possible for a cyberthief to produce a likely match to both codes by generating a relatively small number of answers (say 65,536).

Notes

The cyberthief can then ask the local name server for the IP address for XYZ Bank's home page and learn when it will expire. At the moment of expiration, he again asks for the bank's address and immediately sends out the 65,536 answers that list his own computer's IP address as that of the bank. Under the DNS protocol, the local name server simply accepts the first answer that matches its codes; it does not check from where the answer came, and it ignores any additional replies. Even though XYZ Bank's IP address has not really changed, the local name server still replaces the correct address with the hacker's address and communicates the false information to customers.

So, if our hacker gets his answers in first, the local name server will direct customers seeking XYZ Bank to his computer. Assuming that the hacker runs a convincing imitation of the bank's sign-in page, customers will not realize that they are handing their confidential information over to a fake.

Similar flaws plague other Internet protocols, such as the Border Gateway Protocol (BGP), which governs the pathways followed by data packets on the Internet. They also affect the Dynamic Host Configuration Protocol (DHCP), which roaming computers utilize to find network resources when they are connected in new locations. For example, suppose you are sitting in your favorite coffee shop and want to open a connection to the shop's wireless router. Your laptop broadcasts a query for the server to identify itself, and DHCP directs that your laptop will accept the first response it gets as legitimate. If a hacker sitting across the room can fire off a reply before the coffee shop's router does, your laptop will be connected to his. Everything will seem normal to you, but his computer can record all your communications and covertly direct you to malicious sites at will.

Such vulnerabilities imperil more than individuals and commercial institutions. Secure installations in the government offices and the military can be compromised this way, too. And indeed there have been cases in which these loopholes did allow data to be stolen and records to be altered.

How do we come to be in such a mess? The reasons are partly historical. Today's protocols descend from ones developed 35 years ago when the Internet was still a research network. There was no need to safeguard the network against malicious entities. Now the Internet has opened up and grown explosively, but we have not developed inherently stronger security—the protocols still take for granted that the billions of people and devices online are both competent and honest. Nobody ever went back to do the difficult job of developing inherently stronger security.

Fixing the Internet protocols will be a formidable challenge. Some improvements are relatively simple to imagine—for example, switching to identification codes that use more than 16 bits—but would involve considerable work to adopt on a global basis. Techniques for authenticating that messages coming from the proper parties are well-developed, but those technologies are not necessarily fast enough to be embedded in all the routers on the Internet without bringing traffic to a crawl (or forcing prohibitive investments in new equipment). Some other important kinds of protocol improvements still need to be conceived. Of course, an essential feature of any new protocol is that it can be implemented without seriously disrupting Internet operations in the process.

For these reasons and more, in its February 2005 report, the President's Information Technology Advisory Committee (PITAC), of which I was a member, strongly recommended increased federal funding for basic research into cybersecurity. The Department of Homeland Security currently devotes only one-tenth of 1 percent of its research budget to this concern. DARPA (the Defense Advanced Research Projects Agency) used to fund this kind of work more generously but its current focus is more narrowly military and its research on cybersecurity is classified, limiting the amount of research that can be conducted at universities, and inhibiting the transfer of technology to industry. The National Science Foundation studies the problem but can only do so much. And, although industry takes the problem seriously, inadequate profit incentives discourage companies from aggressively developing broad-based solutions.

Notes

Even once better protocols are in hand, convincing the world to accept them represents its own set of headaches. No central governing body rules the Internet, and standards bodies have been ineffective at getting parties to adopt adequate security specifications. The situation is further complicated by the fact that national governments differ in their views of how the Internet should be run, and many key Internet players argue against any government intervention at all.

What is clear is that cybersecurity deserves immediate, sustained attention. As noted in the PITAC report, "the IT infrastructure of the U.S. is highly vulnerable to terrorist and criminal attacks. It is imperative that we take action before the situation worsens and the cost of inaction becomes even greater."


Did u know?

Operating systems provide the fundamental mechanisms for securing computer processing. Since the 1960s, operating systems designers have explored how to build "secure" operating systems.

9.3 Program Threats

Threats originated from viruses, which are strictly speaking programs that replicate themselves without your knowledge. The earliest known viruses were simply annoying, appearing as a MS-DOS program in folders of infected computers, transferred through floppy disks. They became dangerous when these viruses started to spoof and infect valid files. With the advent of the Internet, viruses further evolved into worms that spread through networks. Worms not only propagate themselves, but also "carry" other malicious files in them, such as Trojan horses, which drop malicious files in computers.


Caution

Threats make affected the system.

9.3.1 Types of Threats

Worms: This malicious program category largely exploits operating system vulnerabilities to spread itself. The class was named for the way the worms crawl from computer to computer, using networks and e-mail. This feature gives many worms a rather high speed in spreading themselves.

Viruses: Programs that infected other programs, adding their own code to them to gain control of the infected files when they are opened. This simple definition explains the fundamental action performed by a virus-infection.

Trojans: Programs that carry out unauthorized actions on computers, such as deleting information on drives, making the system hang, stealing confidential information, etc. This class of malicious program is not a virus in the traditional sense of the word (meaning it does not infect other computers or data). Trojans cannot break into computers on their own and are spread by hackers, who disguise them as regular software. The damage that they incur can exceed that done by traditional virus attacks by several fold.

Spyware: Software that collects information about a particular user or organization without their knowledge. You might never guess that you have spyware installed on your computer.

Riskware: Potentially dangerous applications include software that has not malicious features but could form part of the development environment for malicious programs or could be used by hackers as auxiliary components for malicious programs.

Rootkits: Utilities used to conceal malicious activity. They mask malicious programs to keep

Notes

anti-virus programs from detecting them. Rootkits modify the operating system on the computer and alter its basic functions to hide its own existence and actions that the hacker undertakes on the infected computer.



Task Give the basic security task used in operating system.

9.3.2 System Threats

In all the device drivers considered so far in the book, we have not been overly concerned about the thread context in which our driver subroutines have executed. Most of the time, our subroutines run in an arbitrary thread context, which means we cannot block and cannot directly access user-mode virtual memory. Some devices are very difficult to program when faced with the first set of these constraints.

Some devices are best handled by polling. A device that can not asynchronously interrupt the CPU, for example, needs to be interrogated from time-to-time to check its state. In other cases, the natural way to program the device might be to perform an operation in steps with waits in between. A floppy disk driver, for example, goes through a series of steps to perform an operation. In general, the driver has to command the drive to spin upto speed, wait for the spin-up to occur, commence the transfer, wait for a short while, and then spin the drive back down. You could design a driver that operates as a finite state machine to allow a callback function to properly sequence operations. It would be much easier, though, if you could just insert event and timer waits at the appropriate spots of a straight-line program.

Dealing with situations that require you to periodically interrogate a device is easy with the help of a system thread belonging to the driver. A system thread is a thread that operates within the overall umbrella of a process belonging to the operating system as a whole. In the next section, we will discuss about system threads that execute solely in kernel mode and the mechanism by which you create and destroy your own system threads. Next, we will discuss about an example that how to use a system thread to manage a polled input device.

9.3.3 Network Threats

Trojan horses, worms and DoS (denial of service) attacks are often maliciously used to consume and destroy the resources of a network. Sometimes, misconfigured servers and hosts can serve as network security threats as they unnecessarily consume resources. In order to properly identify and deal with probable threats, one must be equipped with the right tools and security mechanisms. In this article, we will discuss some of the best practices for identifying and dealing with such threats.

9.3.4 Types of Network Threats

Most experts classify network security threats in two major categories—logic attacks and resource attacks. Logic attacks are known to exploit existing software bugs and vulnerabilities with the intent of crashing a system. Some use this attack to purposely degrade network performance or grant an intruder access to a system.

One such exploit is the Microsoft PnP MS05-039 overflow vulnerability. This attack involves an intruder exploiting a stack overflow in the Windows PnP (plug and play) service and can be executed on the Windows 2000 system without a valid user account. Another example of this network security threat is the infamous ping of death where an attacker sends ICMP packets

Notes

to a system that exceeds the maximum capacity. Most of these attacks can be prevented by upgrading vulnerable software or filtering specific packet sequences.

Resource attacks are the second category of network security threats. These types of attacks are intended to overwhelm critical system resources such as CPU and RAM. This is usually done by sending multiple IP packets or forged requests. An attacker can launch a more powerful attack by compromising numerous hosts and installing malicious software. The result of this kind of exploit is often referred to zombies or botnet. The attacker can then launch subsequent attacks from thousands of zombie machines to compromise a single victim. The malicious software normally contains code for sourcing numerous attacks and a standard communications infrastructure to enable remote control.

9.3.4.1 Seek and Destroy

The first step in training your staff to identify network security threats is achieving network visibility. This concept is all rather simple as you cannot defend against or eradicate what you can not see. This level of network visibility can be achieved with existing features found in devices you already have. Additionally, you can create strategic diagrams to fully illustrate packet flows and where exactly within the network you may be able to implement security mechanisms to properly identify and mitigate potential threats.

You must establish a baseline of normal network activity and patterns in order to detect abnormal activity and potential network security threats. Mechanisms like NetFlow can be integrated within your infrastructure to help effectively identify and classify problems. Prior to implementing such a system, you should perform some sort of traffic analysis to fully comprehend the rates and patterns of general traffic. In a successful detection system, learning is achieved over a huge interval which includes the peaks and valleys of network activity.

The best defense against common network security threats involves devising a system that is adhered to by everyone in the network. Furthermore, you can strengthen your level of security with reliable software that makes this process much easier.



Did u know?

No program or operating system is built perfectly secured unless your computer is disconnected from the Internet or unplugged from any connection.

Self Assessment

Multiple choice questions:

1. A stored procedure is a precompiled sequence of Transact-SQL commands in the ..... that are executed by calling the procedure within another SQL command or from the database driver.

   (*a*) database (*b*) software

   (*c*) application (*d*) None of these

2. ..... from viruses, which are strictly speaking programs that replicate themselves without your knowledge.

   (*a*) Database (*b*) Robotics

   (*c*) Simulation (*d*) Threats originated

True or False:

Notes

3. Virus programs infected other programs by adding their own code to them.
4. Rootkit is a program that carry out unauthorized action on computers.

9.4 Cryptography

Does increased security provide comfort to paranoid people? Or does security provide some very basic protections that we are naive to believe that we do not need? During this time when the Internet provides essential communication between millions of people and is being increasingly used as a tool for commerce, security becomes a tremendously important issue to deal with.

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography, which is the focus of this chapter. But it is important to note that while cryptography is necessary for secure communications, it is not by itself sufficient. The reader is advised, then, that the topics covered in this chapter only describe the first of many steps necessary for better security in any number of situations.

This paper has two major purposes. The first is to define some of the terms and concepts behind basic cryptographic methods, and to offer a way to compare the myriad cryptographic schemes in use today. The second is to provide some real examples of cryptography in use today.

No mention is made here about pre-computerized crypto schemes, the difference between a substitution and transposition cipher, cryptanalysis, or other history.

9.4.1 The Purpose of Cryptography

Cryptography is the science of writing in secret code and is an ancient art; the first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. It is no surprise, then, that new forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about any network, particularly the Internet.

Within the context of any application-to-application communication, there are some specific security requirements, including:

- **Authentication:** The process of proving one's identity. (The primary forms of host-to-host authentication on the Internet today are name-based or address-based, both of which are notoriously weak.)
- **Privacy/Confidentiality:** Ensuring that no one can read the message except the intended receiver.
- **Integrity:** Assuring the receiver that the received message has not been altered in any way from the original.
- **Non-repudiation:** A mechanism to prove that the sender really sent this message.

Notes

Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication. There are, in general, three types of cryptographic schemes typically used to accomplish these goals—secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash functions, each of which is described below. In all cases, the initial unencrypted data is referred to as plaintext. It is encrypted into ciphertext, which will in turn (usually) be decrypted into usable plaintext.

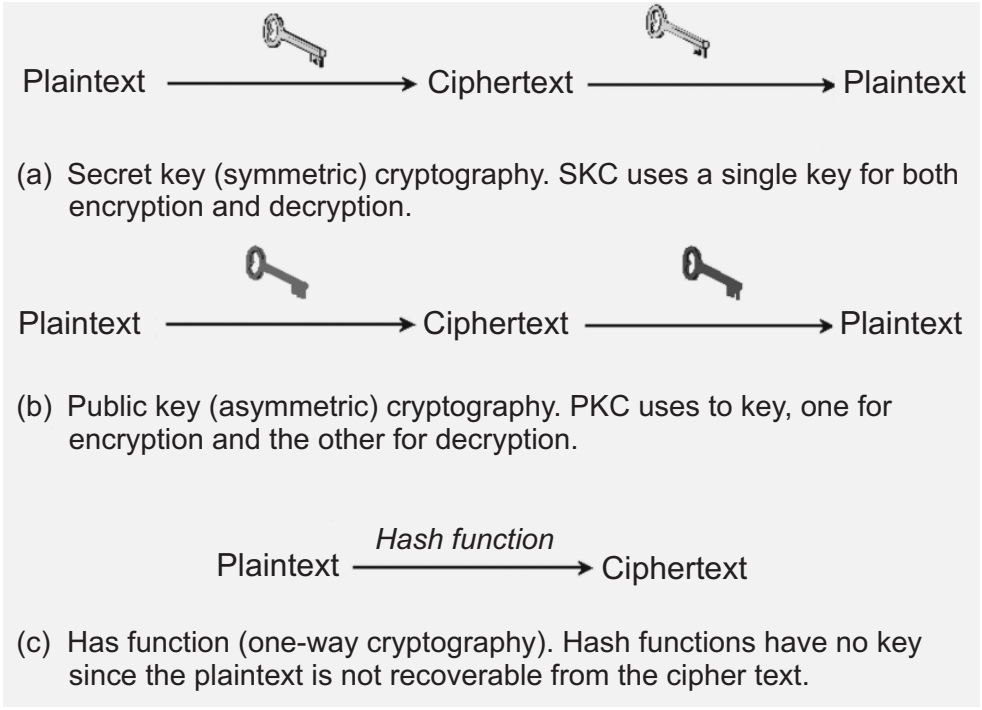
In many of the descriptions below, two communicating parties will be referred to as Alice and Bob; this is the common nomenclature in the crypto field and literature to make it easier to identify the communicating parties. If there is a third or fourth party to the communication, they will be referred to as Carol and Dave. Mallory is a malicious party, Eve is an eavesdropper, and Trent is a trusted third party.

9.4.2 Types of Cryptographic Algorithms

There are several ways of classifying cryptographic algorithms. For purposes of this paper, they will be categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use. The three types of algorithms that will be discussed are (Figure 9.1):

- **Secret Key Cryptography (SKC):** Uses a single key for both encryption and decryption.
- **Public Key Cryptography (PKC):** Uses one key for encryption and another for decryption.
- **Hash Functions:** Uses a mathematical transformation to irreversibly "encrypt" information.

Figure 9.1: Three Types of Cryptography—Secret-key, Public key and Hash Function



Notes

9.4.2.1 Secret Key Cryptography

With secret key cryptography, a single key is used for both encryption and decryption. As shown in Figure 9.1, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or rule set) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called symmetric encryption.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver; that, in fact, is the secret. The biggest difficulty with this approach, of course, is the distribution of the key.

Secret key cryptography schemes are generally categorized as being either stream ciphers or block ciphers. Stream ciphers operate on a single bit (byte or computer word) at a time and implement some form of feedback mechanism so that the key is constantly changing. A block cipher is so-called because the scheme encrypts one block of data at a time using the same key on each block. In general, the same plaintext block will always encrypt to the same ciphertext when using the same key in a block cipher whereas the same plaintext will encrypt to different ciphertext in a stream cipher.

Stream ciphers come in several flavors but two are worth mentioning here. Self-synchronizing stream ciphers calculate each bit in the keystream as a function of the previous n bits in the keystream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the n-bit keystream it is. One problem is error propagation; a garbled bit in transmission will result in n garbled bits at the receiving side. Synchronous stream ciphers generate the keystream in a fashion independent of the message stream but by using the same keystream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the keystream will eventually repeat.

Block ciphers can operate in one of several modes; the following four are the most important:

- Electronic Codebook (ECB) mode is the simplest, most obvious application—the secret key is used to encrypt the plaintext block to form a ciphertext block. Two identical plaintext blocks, then, will always generate the same ciphertext block. Although this is the most common mode of block ciphers, it is susceptible to a variety of brute-force attacks.
- Cipher Block Chaining (CBC) mode adds a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption. In this mode, two identical blocks of plaintext never encrypt to the same ciphertext.
- Cipher Feedback (CFB) mode is a block cipher implementation as a self-synchronizing stream cipher. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting interactive terminal input. If we were using 1-byte CFB mode, for example, each incoming character is placed into a shift register the same size as the block, encrypted, and the block transmitted. At the receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded.
- Output Feedback (OFB) mode is a block cipher implementation conceptually similar to a synchronous stream cipher. OFB prevents the same plaintext block from generating the same ciphertext block by using an internal feedback mechanism that is independent of both the plaintext and ciphertext bitstreams.

Notes


Did u know?

The Cryptography API contains functions that allow applications to encrypt or digitally sign data in a flexible manner, while providing protection for the user's sensitive private key data.

9.5 User Authentication

When things go wrong it is useful to be able to identify the people involved, both the possible victims and those who may have caused the problem. This is as true on computer networks as anywhere else. The aim should be to have all users of JANET identify themselves whenever they are on the network, but in a few situations the cost or inconvenience of achieving this may be unreasonable. Why identify users? The JANET Security Policy requires that connected organizations exercise 'responsibility about giving, controlling and accounting for access to JANET'. The Policy does not mandate that everyone accessing the network must log on to it, but lets each organization to decide how to control network access responsibly. Likewise, the law of the land and the expectations of society do not insist that every action be traceable to an individual. There is no legal requirement to identify or record every logon, e-mail, web request or mouse click. However activity on a network can almost always be traced to an organization that owns an Internet domain or address. Organizations are expected to behave responsibly and will be blamed if they are not seen to do so. For example:

- JISC (Joint Information Systems Committee) may, in extreme cases, suspend or withdraw the right to connect to JANET if an organization's behaviour represents a serious threat to other users of the network;
- Other users may be reluctant to accept communications from an organization that does not deal promptly and effectively with problems, for example some JANET sites have found themselves on blacklists that prevent them exchanging e-mail with others;
- In a few circumstances, the courts may fine an organization or imprison its directors if crimes were committed as a result of their negligence, in other words, if they have not taken reasonable care to avoid causing foreseeable harm;
- More often, courts may require organizations to pay damages to individuals or businesses who have suffered loss or harm because of their negligence;
- Society and the press may publicly blame an organization that fails to meet the standards expected of it. JISC's Legal Information Service (JISCLegal) publishes an article on the legal liability of universities and colleges at—Organizations should consider the risk of misuse when deciding if any groups of users and systems do not need individual identification. An individual account should only take a few minutes to set up. If the user only needs it for a few seconds then creating and deleting an account may be an unreasonable overhead. However, the convenience of not setting up and managing individual accounts cannot justify a significantly increased risk of harm to others and the organization. Harm can be caused by hacking, malicious messages, downloading illegal material and many other types of activity, the scope for which will normally be less where an individual's access is limited to a few systems, rather than the whole Internet. However, if critical internal systems may be accessed then the potential harm should not be underestimated. How to identify users the most common way for individuals to identify themselves is to log on when they sit down at a terminal; however, this is not the only option. If users have to prove their identity to get into a workstation room or borrow a laptop then a record can be kept of who used which computer when. Some organizations let anyone see a limited set of web pages but require a login to gain access to other sites or services. However they

Notes

are collected, records linking a user to his or her IP address should be kept long enough for misuse to be reported and investigated. Staff and students of the organization should have their own local accounts. Visitors may also have local accounts, or authorized staff may be enabled to set up daily accounts for their guests. Visitors from other organizations may be authenticated by their home organization if both organizations are members of JANET Roaming or another partner in the teReNA (trans-European Research and education Network Association) eduroam federation.

Even if individual identities are not checked, access to the JANET network must still be limited to those who are known to the organization. Knowingly providing network access to strangers is likely to be a breach of JANET policies and to be considered irresponsible by other users of the network. Access may be limited by physical barriers, although this does not work for wireless networks, or by providing temporary access codes to guests such as conference delegates.

Organizations may wish to arrange their networks so that these visitors do not accidentally obtain access to internal resources controlled or licensed by IP address.

Organizations that provide access to networks, and users who benefit from that access, should regard it as normal to require an individual identity. Systems for establishing electronic identity are becoming easier to use and manage. In a few situations there may be a justification for not checking and recording identity but this should only be done after a rational assessment of the risks and benefits.



Task Explain how to change the user authentication password in the computer.

9.5.1 Implementation of Defenses

Several layers of security can be built on the top of the database. This article primarily focuses on use of encryption and stored procedures at the database level. While the use of encryption techniques prevents eavesdropping and interception of the traffic at the network level, the use of stored procedures protects against attacks involving tampering of data sent to the server.

9.5.1.1 First Layer of Defense (Encryption)

When encryption is used, the traffic between the database driver and the database server is encrypted. This makes it difficult for the attackers to intercept the data in transit, thereby preventing successful execution of several attacks such as injection based attacks on two-tier thick client applications.

The two most used encryption techniques are Internet Protocol Security (IPSEC) and Secure Socket Layer (SSL). Many of the latest versions of the databases support both types of encryption techniques. While IPSEC encryption works on the network layer, SSL encryption works at the transport layer leading to an easier implementation. Currently, SSL encryption is the more popular one due to its ease of implementation. Let us see a few examples of using SSL encryption on popular databases.

9.5.1.2 The SSL Handshake

When a client (database driver) initiates a connection to the server over SSL, a SSL handshake occurs between the client and server. During this handshake, both the client and the server agree upon a specific cipher suite that specifies the encryption algorithm to be used. Then the server authenticates itself to the client by providing its certificate signed by a trusted CA. Later, both the client and server generate a session key and exchange it using a public key cryptography. Any further communication happens in an encrypted form.

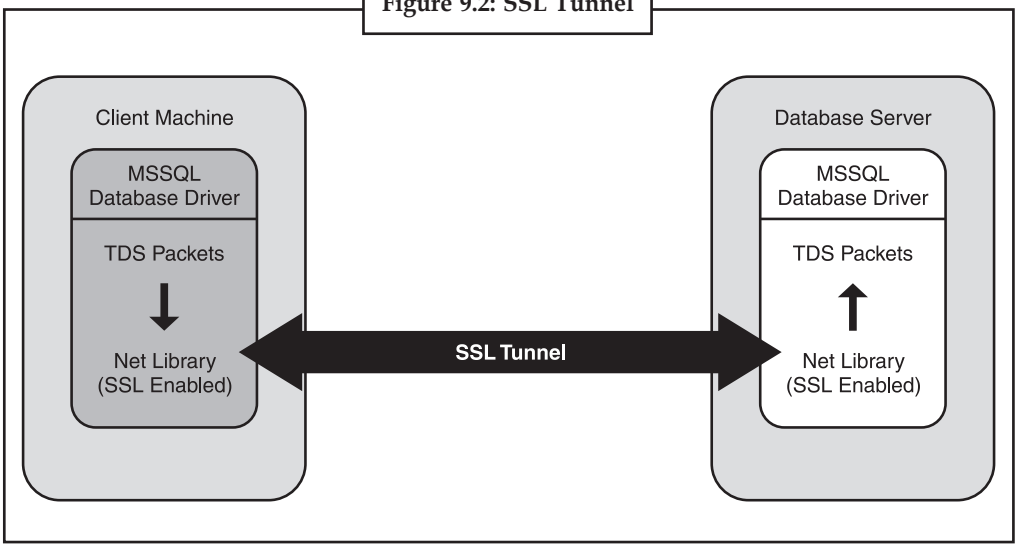
Notes

9.5.1.3 SSL Encryption in MS SQL

Here, we consider SQL Server 2000 for the discussion. SQL Server uses Tabular Data Stream (TDS) packets for exchanging commands with its client counterparts. These TDS packets are handled by Net Library protocols, which enable communication between the SQL Server and its clients over a network. In SQL Server 2000, these Net Libraries can be configured using SQL Server Network Utility for Secure Socket Layer encryption that uses a Super Socket Net Library, which aids other Net Libraries.

SSL encryption can be implemented between SQL Server 2000 and its clients by obtaining a certificate from an appropriate Certificate Authority and installing it on the server. Then all the clients need to be configured to trust the issuing CA. Finally, the protocol encryption has to be forced using the Server Network Utility. A detailed description of Net Libraries and implementing SSL over them is available here.

Figure 9.2: SSL Tunnel



9.5.1.4 SSL Encryption in Oracle

Oracle database uses various features of the Oracle Advanced Security option to provide security to the enterprise networks. The SSL feature of the Oracle Advanced Security option enables a secure communication between Oracle Database server and client by encrypting the traffic. In addition, it also provides authentication of server or client or both. This SSL functionality can also be combined with other authentication methods supported by Oracle Advanced Security, thereby using the SSL encryption feature alone.

9.5.1.5 Securing Oracle Network Traffic

Oracle provides a platform independent networking infrastructure for accessing databases, which is called Net8. This Net8 product with the Oracle Advanced Security option has a feature to use Secure Shell (SSH) protocol to secure the traffic between the client and the server. Though this mechanism protects against eavesdropping, it does not protect against the attacks discussed in the previous article as the database server and database driver are separated from the SSH tunnel.

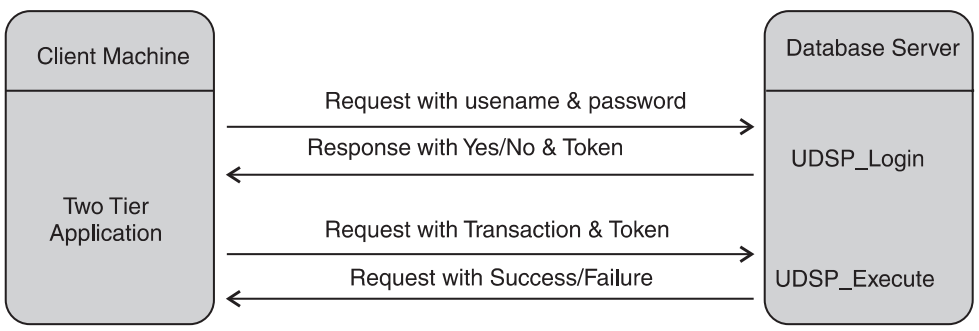
Notes

9.5.1.6 Second Layer of Defense (Stored Procedures)

A stored procedure is a precompiled sequence of Transact-SQL commands in the database that are executed by calling the procedure within another SQL command or from the database driver. It also significantly reduces the amount of data being transmitted between the database server and the web server.

User defined stored procedures can be used to perform several activities at the database such as authentication and authorization checks. A stored procedure can also be used to manage user sessions by using authentication tokens. The following diagram explains how a stored procedure can be used for authentication and authorization purposes.

Figure 9.3: Authentication and Authorization Purposes



In the above diagram, UDSP Login is the stored procedure used to perform authentication checks and UDSP Execute is the stored procedure used to perform authorization checks.

Authentication checks

When a user submits the login credentials, the application sends it to the database in an SQL query. The SQL query calls the stored procedure UDSP Login, which verifies the credentials and if correct returns a randomly created token. The database server responds with authentication success along with the token. This token is mapped with the username used for login in a database table and is used as a Session ID in a web application. Incase the authentication fails, the database returns only an authentication failure message. This protects the application from various authentication attacks.

Authorization checks

After the user has logged in to the application, all transaction/execution requests contains the token created at the time of authentication. The SQL query request calls the UDSP_Execute stored procedure, which validates the request by using the token to identify the user mapped to the token and the rights assigned to that user. This protects the application against unauthorized access and privilege escalation attacks.

Notes

Other implications

Use of stored procedures also has several other security implications. By using stored procedures, a user can be restricted to access only specific rows and columns of a database table. This enables effective management of user permissions across all the database tables. This database journal article on SQL Stored Procedures discusses the implications of using them.

9.5.1.7 Additional Layer of Defense (Database Security Patches)

Database vendors release periodic security patches to fix several software bugs in the database left open during development of the particular version. These patches should be installed on the database servers as and when they are available. This reduces the chances of the database being exploited through the known vulnerabilities. Vendors have their own periodicity and distribution mechanism for security patches. Oracle uses its Security Technology Center to announce Security Alerts and Patches.

There is an article series published in the database journal that explains the importance and installation procedures for database patches. One of the articles, describes an SQL Injection error in the Oracle database and the patch released to fix the error. Similarly, several patches address certain specific errors that may lead to compromise of the database through different applications used to access the database. Hence, it is essential to establish several layers of security on the database to ensure it is safe and secure from attacks.

9.6 Firewall

A **firewall** is a part of a computer system or network that is designed to block unauthorized access while permitting authorized communications. It is a device or set of devices that is configured to permit or deny network transmissions based upon a set of rules and other criteria.

Firewalls can be implemented in either hardware or software, or a combination of both. Firewalls are frequently used to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets. All messages entering or leaving the intranet pass through the firewall, which inspects each message and blocks those that do not meet the specified security criteria.

There are several types of firewall techniques:

1. **Packet Filter:** Packet filtering inspects each packet passing through the network and accepts or rejects it based on user-defined rules. Although difficult to configure, it is fairly effective and mostly transparent to its users. It is susceptible to IP spoofing.
2. **Application Gateway:** Applies security mechanisms to specific applications, such as FTP and Telnet servers. This is very effective, but can impose a performance degradation.
3. **Circuit-level Gateway:** Applies security mechanisms when a TCP or UDP connection is established. Once the connection has been made, packets can flow between the hosts without further checking.

Notes

4. **Proxy Server:** Intercepts all messages entering and leaving the network. The proxy server effectively hides the true network addresses.

The term firewall/fire block originally meant a wall to confine a fire or potential fire within a building; cf. firewall (construction). Later uses refer to similar structures, such as the metal sheet separating the engine compartment of a vehicle or aircraft from the passenger compartment.

- The Morris Worm spread itself through multiple vulnerabilities in the machines of the time. Although it was not malicious in intent, the Morris Worm was the first large scale attack on Internet security; the online community was neither expecting an attack nor prepared to deal with one.

Figure 9.4: Firewall Window



9.6.1 First Generation: Packet Filters

The first paper published on firewall technology was in 1988, when engineers from Digital Equipment Corporation (DEC) developed filter systems known as packet filter firewalls. This fairly basic system was the first generation of what became a highly evolved and technical internet security feature. At AT&T Bell Labs, Bill Cheswick and Steve Bellovin were continuing their research in packet filtering and developed a working model for their own company based on their original first generation architecture.

This type of packet filtering pays no attention to whether a packet is part of an existing stream of traffic (i.e. it stores no information on connection "state"). Instead, it filters each packet based only on information contained in the packet itself (most commonly using a combination of the

Notes

packet's source and destination address, its protocol, and, for TCP and UDP traffic, the port number).

TCP and UDP protocols constitute most communication over the Internet, and because TCP and UDP traffic by convention uses well known ports for particular types of traffic, a "stateless" packet filter can distinguish between, and thus control, those types of traffic (such as web browsing, remote printing, email transmission, file transfer), unless the machines on each side of the packet filter are both using the same non-standard ports.

Packet filtering firewalls work mainly on the first three layers of the OSI reference model, which means most of the work is done between the network and physical layers, with a little bit of peeking into the transport layer to figure out source and destination port numbers. When a packet originates from the sender and filters through a firewall, the device checks for matches to any of the packet filtering rules that are configured in the firewall and drops or rejects the packet accordingly. When the packet passes through the firewall, it filters the packet on a protocol/port number basis (GSS). For example, if a rule in the firewall exists to block telnet access, then the firewall will block the IP protocol for port number 23.

9.6.2 Second Generation: Application Layer

The key benefit of application layer filtering is that it can "understand" certain applications and protocols (such as File Transfer Protocol, DNS, or web browsing), and it can detect if an unwanted protocol is sneaking through on a non-standard port or if a protocol is being abused in any harmful way.

An application firewall is much more secure and reliable compared to packet filter firewalls because it works on all seven layers of the OSI model, from the application down to the physical Layer. This is similar to a packet filter firewall but here we can also filter information on the basis of content. Good examples of application firewalls are MS-ISA (Internet Security and Acceleration) server, McAfee Firewall Enterprise & Palo Alto PS Series firewalls. An application firewall can filter higher-layer protocols such as FTP, Telnet, DNS, DHCP, HTTP, TCP, UDP and TFTP (GSS). For example, if an organization wants to block all the information related to "foo" then content filtering can be enabled on the firewall to block that particular word. Software-based firewalls (MS-ISA) are much slower than hardware based stateful firewalls but dedicated appliances (McAfee & Palo Alto) provide much higher performance levels for Application Inspection.

In 2009/2010, the focus of the most comprehensive firewall security vendors turned to expanding the list of applications such firewalls are aware of now covering hundreds and in some cases thousands of applications which can be identified automatically. Many of these applications can not only be blocked or allowed but manipulated by the more advanced firewall products to allow only certain functionally enabling network security administrations to give users functionality without enabling unnecessary vulnerabilities. As a consequence these advanced version of the "Second Generation" firewalls are being referred to as "Next Generation" and surpass the "Third Generation" firewall. It is expected that due to the nature of malicious communications this trend will have to continue to enable organizations to be truly secure.

9.6.3 Third Generation: "Stateful" Filters

From 1989-1990, three colleagues from AT&T Bell Laboratories, Dave Presetto, Janardan Sharma, and Kshitij Nigam, developed the third generation of firewalls, calling them circuit level firewalls.

Notes

Third-generation firewalls, in addition to what first- and second-generation look for, regard placement of each individual packet within the packet series. This technology is generally referred to as a stateful packet inspection as it maintains records of all connections passing through the firewall and is able to determine whether a packet is the start of a new connection, a part of an existing connection, or is an invalid packet. Though there is still a set of static rules in such a firewall, the state of a connection can itself be one of the criteria which trigger specific rules.

This type of firewall can actually be exploited by certain Denial-of-service attacks which can fill the connection tables with illegitimate connections.

9.6.4 Subsequent Developments

In 1992, Bob Braden and Annette DeSchon at the University of Southern California (USC) were refining the concept of a firewall. The product known as "Visas" was the first system to have a visual integration interface with colours and icons, which could be easily implemented and accessed on a computer operating system such as Microsoft's Windows or Apple's MacOS. In 1994, an Israeli company called Check Point Software Technologies built this into readily available software known as FireWall-1.

The existing deep packet inspection functionality of modern firewalls can be shared by Intrusion-prevention systems (IPS).

Currently, the Middlebox Communication Working Group of the Internet Engineering Task Force (IETF) is working on standardizing protocols for managing firewalls and other middleboxes.

Another axis of development is about integrating identity of users into Firewall rules. Many firewalls provide such features by binding user identities to IP or MAC addresses, which is very approximate and can be easily turned around. The NuFW firewall provides real identity-based firewalling, by requesting the user's signature for each connection. authpf on BSD systems loads firewall rules dynamically per user, after authentication via SSH.

9.6.5 Types

There are several classifications of firewalls depending on where the communication is taking place, where the communication is intercepted and the state that is being traced.

9.6.5.1 Network Layer and Packet Filters

Network layer firewalls, also called packet filters, operate at a relatively low level of the TCP/IP protocol stack, not allowing packets to pass through the firewall unless they match the established rule set. The firewall administrator may define the rules; or default rules may apply. The term "packet filter" originated in the context of BSD operating systems.

Network layer firewalls generally fall into two subcategories, stateful and stateless. Stateful firewalls maintain context about active sessions, and use that "state information" to speed packet processing. Any existing network connection can be described by several properties, including source and destination IP address, UDP or TCP ports, and the current stage of the connection's lifetime (including session initiation, handshaking, data transfer, or completion connection). If a packet does not match an existing connection, it will be evaluated according to the rule set for new connections. If a packet matches an existing connection based on comparison with the firewall's state table, it will be allowed to pass without further processing.

Notes

Stateless firewalls require less memory, and can be faster for simple filters that require less time to filter than to look up a session. They may also be necessary for filtering stateless network protocols that have no concept of a session. However, they cannot make more complex decisions based on what stage communications between hosts have reached.

Modern firewalls can filter traffic based on many packet attributes like source IP address, source port, destination IP address or port, destination service like WWW or FTP. They can filter based on protocols, TTL values, netblocks of originator, of the source, and many other attributes.

Commonly used packet filters on various versions of Unix are ipf (various), ipfw (FreeBSD/Mac OS X), pf (OpenBSD, and all other BSDs), iptables/ipchains (Linux).

9.6.5.2 Application-layer

Application-layer firewalls work on the application level of the TCP/IP stack (i.e., all browser traffic, or all telnet or ftp traffic), and may intercept all packets traveling to or from an application. They block other packets (usually dropping them without acknowledgment to the sender). In principle, application firewalls can prevent all unwanted outside traffic from reaching protected machines.

On inspecting all packets for improper content, firewalls can restrict or prevent outright the spread of networked computer worms and Trojans. The additional inspection criteria can add extra latency to the forwarding of packets to their destination.

9.6.5.3 Proxies

A proxy device (running either on dedicated hardware or as software on a general-purpose machine) may act as a firewall by responding to input packets (connection requests, for example) in the manner of an application, whilst blocking other packets.

Proxies make tampering with an internal system from the external network more difficult and misuse of one internal system would not necessarily cause a security breach exploitable from outside the firewall (as long as the application proxy remains intact and properly configured). Conversely, intruders may hijack a publicly-reachable system and use it as a proxy for their own purposes; the proxy then masquerades as that system to other internal machines. While use of internal address spaces enhances security, crackers may still employ methods such as IP spoofing to attempt to pass packets to a target network.

9.6.5.4 Network Address Translation

Main article—Network address translation

Firewalls often have network address translation (NAT) functionality, and the hosts protected behind a firewall commonly have addresses in the "private address range", as defined in RFC 1918. Firewalls often have such functionality to hide the true address of protected hosts. Originally, the NAT function was developed to address the limited number of IPv4 routable addresses that could be used or assigned to companies or individuals as well as reduce both the amount and therefore cost of obtaining enough public addresses for every computer in an organization. Hiding the addresses of protected devices has become an increasingly important defense against network reconnaissance.



Did u know?

A firewall is a set of related programs, located at a network gateway server, that protects the resources of a private network from users from other networks.

Notes


Case Study

Digital Signatures and Public Key Cryptography

Signatures on documents in the real world are the specialized or unique mark or impression made by the person with the help of ink. It is assumed that no two persons would have the same signature. The signature of a person on the document implies that the document is attributed to him. Signature are made by the person to authenticate the documents. 'The authenticity of many legal, financial and other documents is determined by the presence or absence of an authorized handwritten signature. For computerized message systems to replace the physical medium of paper and ink documents, a solution must be found to the problem of authenticating the messages. The solution to this problem in digital media is called Digital Signature.

Relating Digital Signatures with Public Key Cryptography

As we have seen in the public cryptographic system, a sender encrypts the message using the receiver's public key. The encrypted message is then decrypted by the receiver's private key. Digital Signatures are based on the same concept but the difference is that the sender encrypts the message with its private key and the receiver decodes it with the sender's public key. Thus the mechanism of public key cryptography is reversed in the implementation of digital signature system. The signer or sender encodes the document with his own private key. This allows anyone with his public key to decode the document. Since the documents can be decoded with his public key, and he is the only one who has access to the corresponding private key, everyone knows that he really did encode (sign) it. This proves the authenticity and the integrity of the document.

The sender or the signer (A) of a document (D) will sign the document in the following ways:

1. Encrypt document (D) with the private key of sender (A).
2. Cipher text produced in the step 1, is thus the signed document 3.
3. The receiver (B) of the cipher text will decrypt it using the public key of sender (A).

The sender publishes his public key to all the potential recipients of a document signed by him. The private key of the sender remains with the sender thus, only he will be able to encrypt the document with this key. Only the receiver who has the public key of the sender will be able to decrypt the document. This serves the purpose of a digital signature. Therefore, we may define the digital signature as follows:

Digital Signature represents that way of document encryption or public key cryptography, in which the documents are encrypted by the private key of the signer and decrypted by the receiver using the public key of the signer.

Questions:

1. What are digital signatures?
2. What is the relationship of Digital signatures with Public Key Cryptography?

Self Assessment

Fill in the blanks:

5. ..... is the science of writing in secret code.
6. Secret key cryptography uses a single key for both ..... and decryption.

Notes

7. ..... encryption technique works on the network layer.
8. A stored procedure is a precompiled sequence of ..... command in the database.
9. ..... mode adds a feedback mechanism to the encryption scheme.
10. A ..... is a part of computer system that is designated to block unauthorized access while permitting authorized communications.

9.7 Summary

- CERT/CCC Current Activity KNOPPIX can also be used as a creative Linux system as the educational CD, desktop, rescue system.
- Security systems generally implement a useful subset of a 3-dimentional matrix with dimensions of actor, object, and action. Implementing the full matrix is never done for both performance and usability reasons.
- Mainstream operating system security is primarily based on access control lists.
- Traditional UNIX access control is a type of discretionary ACL.
- The many possible actions are grouped into four categories—read, write, execute, and special operations normally reserved for an object owner.
- Capabilities are unforgivable references to objects that let their holder access a well-defined subset of operations defined on that object.
- Threats originated from viruses, which are strictly speaking programs that replicate themselves without your knowledge.
- Secret key cryptography schemes are generally categorized as being either stream ciphers or block ciphers.
- Database vendors release periodic security patches to fix several software bugs in the database left open during development of the particular version.
- A firewall is a part of a computer system or network that is designed to block unauthorized access while permitting authorized communications.
- It is a device or set of devices that is configured to permit or deny network transmissions based upon a set of rules and other criteria.

9.8 Keywords

1. ***Domain Name System (DNS)*:** The Domain Name System (DNS) is a hierarchical naming system built on a distributed database for computers, services, or any resource connected to the Internet or a private network.
2. ***Dynamic Host Configuration Protocol (DHCP)***: Dynamic host configuration protocol is used to automatically assign TCP/IP addresses to clients along with the correct subnet mask, default gateway, and DNS server.

   Two ways for a computer to get its IP address:

   - Using DHCP from a DHCP server.
   - Manual configuration.
3. ***Encryption*:** The Encrypting File System (EFS) on Microsoft Windows is a file system filter that provides filesystem-level encryption and was introduced in version 3.0 of NTFS. The

Notes

technology enables files to be transparently encrypted to protect confidential data from attackers with physical access to the computer.

4. ***The President's Information Technology Advisory Committee (PITAC)*:** The President's Information Technology Advisory Committee (PITAC) was established in February 1997, to provide the President, OSTP, and the federal agencies involved in IT R&D with guidance and advice on all areas of high-performance computing, communications, and information technologies. Representing the research, education, and library communities and including network providers and representatives from critical industries, the committee advises the Administration's effort to accelerate development and adoption of information technologies. PITAC was appointed by the President to provide independent expert advice on maintaining America's preeminence in advanced information technology (IT).
5. ***KeyKOS*:** KeyKOS is a persistent, pure capability-based operating system for the IBM S/370 mainframe computers. It allows emulating the VM, MVS, and POSIX environments. It is a predecessor of the Extremely Reliable Operating System (EROS), and its successors, the CapROS and Coyotos operating systems. KeyKOS is a nanokernel-based operating system.
6. ***Mandatory Integrity Control (MIC) or Integrity Levels*:** (or Protected Mode in the context of applications like Internet Explorer, Google Chrome and Adobe Reader) MIC is a core security feature, introduced in Windows Vista and Windows Server 2008, that adds Integrity Levels (IL) to processes running in a login session.
7. ***Protection*:** A model of protection mechanisms in computing systems is presented and its appropriateness is argued. The "safety" problem for protection systems under this model is to determine in a given situation whether a subject can acquire a particular right to an object.
8. ***Secure Sockets Layer (SSL)*:** Secure Sockets Layer (SSL) protocol to create a uniquely encrypted channel for private communications over the public Internet. Each SSL Certificate consists of a public key and a private key.



Lab Exercise

1. How to enable the security of operating system?
2. Give the step of installing antivirus.

9.9 Review Questions

1. Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.
2. A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.
3. The list of all passwords is kept within the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (*Hint:* Use different internal and external representations.)
4. Discuss a means by which managers of systems connected to the Internet could have designed their systems to limit or eliminate the damage done by a worm. What are the drawbacks of making the change that you suggest?
5. Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

Notes

6. What are the two advantages of encrypting data stored in the computer system?
7. Why does not $D(kd , N)(E(ke , N)(m))$ provide authentication of the sender? To what uses can such an encryption be put?
8. Discuss, what you know about system security.
9. Explain in detail the program threats in system security.
10. Explain the networking threats with examples.
11. What is the role of the cryptography in system security?
12. What is the difference between program threads and networking threads?
13. Briefly describe about the authentication in system security.
14. Describe the role of firewall in system security.
15. Differentiate between implementing security defense and firewall.

Answers to Self Assessment

1. (*a*)
2. (*d*)
3. True
4. False
5. Cryptography
6. encryption
7. IPSEC
8. Transact-SQL
9. Cipher Block Chaining (CBC)
10. firewall

9.10 Further Readings



Books

Operating Systems, by Andrew Tanenbaum, Albert S. Woodhull.
Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes.



Online link

wiley.com/coolege.silberschatz

Unit 10: Introduction of Windows and its Programming

Notes

CONTENTS

Objectives

Introduction

10.1 History of Windows 2000

10.1.1 MS-DOS

10.1.2 Windows 95/98/Me

10.1.3 Windows NT

10.1.4 Windows 2000

10.2 Programming Windows 2000

10.2.1 Win32 Application Programming Interface

10.2.2 Registry

10.3 Summary

10.4 Keywords

10.5 Review Questions

10.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain history of Windows 2000
- Explain programming Windows 2000

Introduction

Windows 2000 is the latest update in the Microsoft Windows family of products. It is a combination of features designed in the Windows 98 and NT 4.0. Like previous versions of Windows, it uses a Graphical User Interface (GUI) format, Plug-and-Play compatibility and USB support. What makes Windows 2000 significantly different are the formats it is available in. There are following four products that compose the Windows 2000 family.

Windows 2000 Professional

This version of Windows 2000 is equivalent to the Windows 98/NT 4.0 workstation clients. It is designed to offer basic peer-to-peer networking services and client services in a client-server network. It is designed to integrate the ease of usability of Windows 98 with the reliability

Notes

and security of Windows NT 4.0. Basic improvements include a more reliable user interface, enhanced Plug-and-Play compatibility, increased power management options and extended hardware compatibility, including direct USB and FireWire support. It also uses a new file encryption system that increases security on the network when integrated with Active Directory Services. Finally, it has a host of new application management tools that simplify and extend administrative and user control over the network.

Windows 2000 Server

Windows 2000 Server is a network-enhanced version of Windows 2000 Professional. It contains all the same aspects as Windows 2000 Pro, but adds network serving ability, enhanced file and print sharing services, application server technology, and Web-Server utilities. It is designed to allow small-to-medium-sized businesses network their systems efficiently at a lower cost than traditional NT 4.0 methods by stripping out unused tools.

Windows 2000 integrates Active Directory Services into several existing services such as Domain Name System (DNS), Dynamic Host Control Protocol (DHCP) and WINS (Windows Internet Name Service) allowing central control over management of users, groups, security and network resources. It supports single-processor systems as well as four-way symmetric multiprocessing (SMP) systems. It supports up to 4 GB of physical memory.

Windows 2000 Advanced Server

Advanced server is essentially the same as Windows 2000 Server with enhanced scalability and advanced high availability required for larger enterprise servers and departmental solutions. It focuses more on application and departmental networking, with support for eight-way symmetric multiprocessing and two-way clustering. It also integrates Intel's Physical Address Extensions (PAEs) technology to allow for support for larger physical memory quantities. It is meant for larger businesses with database-intensive requirements.

Windows 2000 Datacenter Server

Datacenter is a highly specialized version of Windows 2000 designed for large-scale enterprise solutions. It integrates technologies optimized for large data warehouses, econometric analysis, large-scale simulations in science and engineering, online transaction processing (OLTP) and server consolidation projects. It adds elements to enhance Internet Service Provider (ISP) support and Web Hosting Services. It supports 4-way clustering and sixteen-way Symmetric multiprocessing (Upgradeable to 32-way SMP).

10.1 History of Windows 2000

Microsoft operating systems for desktop and laptop PCs can be divided into three families—MS-DOS, Consumer Windows (Windows 95/98/Me), and Windows NT. Below we will briefly sketch each of these families.



Windows 2000 is a line of operating systems produced by Microsoft for use on personal computers, business desktops, laptops, and servers. Released on 17 February 2000.

10.1.1 MS-DOS

In 1981, IBM, at that time, the biggest and most powerful computer company in the world, produced the 8088-based IBM PC. The PC came equipped with a 16-bit real-mode, single-user, command-line oriented operating system called MS-DOS 1.0. The operating system was provided

Notes

by Microsoft, a tiny startup, mostly known at that time for its BASIC interpreter used on 8080 and Z-80 systems. This operating system consisted of 8 KB of memory resident code and was closely modeled on CP/M, a tiny operating system for the 8-bit 8080 and Z80 CPUs. Two years later, a much more powerful 24-KB operating system, MS-DOS 2.0, was released. It contained a command line processor (shell), with a number of features borrowed from UNIX.

When Intel came out with the 286 chip, IBM built a new computer around it, the PC/AT, released in 1986. AT stood for "Advanced Technology", because the 286 ran at a then impressive 8 MHz and could address-with great difficulty-all of 16 MB of RAM. In practice, most systems had at most 1 MB or 2 MB, due to the great expense of so much memory. The PC/AT came equipped with Microsoft's MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line oriented system.



Task Give the command of make directory in the DOS.

10.1.2 Windows 95/98/ME

Inspired by the user interface of the Apple Lisa, the forerunner to the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface (shell) which is called **Windows**. Windows 1.0, released in 1985, was something of a dud. Windows 2.0, designed for the PC-AT and released in 1987, was not much better. Finally, Windows 3.0 for the 386 (released in 1990), and especially its successors 3.1 and 3.11, caught on and were huge commercial successes. None of these early versions of Windows were true operating systems, but more like graphical user interfaces on top of MS-DOS, which was still in control of the machine and the file system. All programs ran in the same address space and a bug in any one of them could bring the whole system to a grinding halt.

The release of Windows 95 in August 1995 still did not completely eliminate MS-DOS, although it transferred nearly all the features from the MS-DOS part to the Windows part. Together, Windows 95 and the new MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit assembly code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations. The only major change to the file system was the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS.

Even with the release of Windows 98 in June 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although yet more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not much different from Windows 95. The main difference was the user interface, which integrated the desktop and the Internet more closely. It was precisely this integration that attracted the attention of the U.S. Deptartment of Justice, which then sued Microsoft claiming that it was an illegal monopoly, an accusation Microsoft vigorously denied. In April 2000, the U.S. Federal court agreed with the government. In addition to containing a large lump of old 16-bit assembly code in the kernel, Windows 98 had two other serious problems. First, although it was a multiprogramming system, the kernel itself was not reentrant. If a process was busy in manipulating some kernel data structure and then suddenly its quantum ran out and another process started running, the new process might find the data structure in an inconsistent state. To prevent this type of problem, after entering the kernel, most processes first acquired a giant

Notes

mutex covering the whole system before doing anything. While this approach eliminated potential inconsistencies, it also eliminated the most of the values of multiprogramming since processes were frequently forced to wait for unrelated processes to leave the kernel before they could enter it.

Second, each Windows 98 process had a 4-GB virtual address space. Of this, 2 GB was completely private to the process. However, the next 1 GB was shared (writably) among all other processes in the system. The bottom 1 MB was also shared among all processes to allow all of them to access the MS-DOS interrupt vectors. This sharing facility was heavily used by most Windows 98 applications. As a consequence, a bug in one program could wipe out key data structures used by unrelated processes, leading to whole crashing. Worse yet, the last 1 GB was shared (writably) with the kernel and contained some critical kernel data structures. Any rogue program that overwrote these data structures with garbage could bring down the system. The obvious solution of not putting kernel data structures in user space was not possible because this feature was essential to making old **MS-DOS** programs work under Windows 98.

In the millennium year, 2000, Microsoft brought out a minor revision to Windows 98 called **Windows Me (Windows Millennium Edition).** Although it fixed a few bugs and added a few features, under the covers it is essentially Windows 98. The new features included better ways to catalog and share images, music, and movies, more support for home networking and multiuser games, and more Internet-related features, such as support for instant messaging and broadband connections (cable modems and ADSL). One interesting new feature was the ability to restore the computer to its previous settings after a misconfiguration. If a user reconfigures the system (e.g., changing the screen from 640 × 480 to 1024 × 768) and it no longer works, this feature makes it possible to revert back to the last known working configuration.

10.1.3 Windows NT

By the late 1980s, Microsoft realized that building a modern 32-bit operating system on the top of the leaky 16-bit MS-DOS probably was not the best way to go. It recruited David Cutler, one of the key designers of DEC's VMS operating system, to work for Microsoft and gave him the job of leading a team to produce a brand-new 32-bit Windows compatible operating system from the ground up. This new system, later called **Windows NT (Windows New Technology),** was intended for mission-critical business applications as well as for home users. At the time, mainframes still ruled the (business) world, so designing an operating system on the assumption that companies would use personal computers for anything important was a visionary goal, but one that history has shown to be a very good one. Features such as security and high reliability, clearly lacking on the MS-DOS-based versions of Windows, were high on the agenda for (Windows) NT. Cutler's background with VMS clearly shows in various places, with there being more than a passing similarity between the design of NT and that of VMS.

The project succeeded and the first version, called Windows NT 3.1, was released in 1993. This initial release number was chosen to match the number of Microsoft's then popular 16-bit Windows 3.1 system. Microsoft expected that NT would rapidly replace Windows 3.1 because it was technically a far superior system.

Much to its surprise, nearly all users preferred to stick with the old 16-bit system they knew, rather than upgraded to an unknown 32-bit system they did not know, however better it might be. Furthermore, NT required far more memory than Windows 3.1 and there were no 32-bit programs for it to run, so why bother? The failure of NT 3.1 to catch on in the marketplace was the reason Microsoft decided to build a 32-bit-ish version of Windows 3.1, namely Windows 95. The continued user resistance to NT then caused Microsoft to produce Windows 98 and finally Windows Me; each one claimed to be the very last release of the MS-DOS-based systems.

Notes

Despite the fact that nearly all consumers and most businesses ignored NT 3.1 for desktop systems, it did acquire a small following in the server market. A few new 3.x releases with small changes occurred in 1994 and 1995. These slowly began to acquire more following among desktop users as well.

The first major upgradation to NT came with NT 4.0 in 1996. This system had the power, security, and reliability of the new operating system, but also supported the same user interface as the by then very popular Windows 95. This compatibility made it much easier for users to migrate from Windows 95 to NT, and many of them did so.

From the beginning, NT was designed to be portable, so it was written almost entirely in C, with only a tiny bit of assembly code for low-level functions such as interrupt handling. The initial release consisted of 3.1 million lines of C for the operating system, libraries, and the environment subsystems (discussed below). When NT 4.0 came out, the code base had grown to 16 million lines of code, still mostly C, but with a small amount of C++ in the user interface part. By this time the system was highly portable, with versions running on the Pentium, Alpha, MIPS, and PowerPC, among other CPUs. Some of these have been dropped since then. The story of how NT was developed is given in the site Showstopper. The site also tells a lot about the key people involved.

Figure 10.1: Some Differences between Windows 98 and Windows NT

Item	Windows 95/98	Windows NT
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Private addr space for each MS-DOS prog?	No	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha, MIPS, …
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Plug and play?	Yes	No
Power management?	Yes	No
FAT-32 file system?	Yes	Optional
NTFS file system?	No	Yes
Win32 API?	Yes	Yes
Run all old MS-DOS programs?	Yes	No
Some critical OS data writable by user?	Yes	No

Notes



Microsoft announced Multi-Tool Word for Xenix and MS-DOS in 1983. Its name was soon simplified to Microsoft Word. Free demonstration copies of the application were bundled with the November 1983 issue of PC World, making it the first program to be distributed on-disk with a magazine.

10.1.4 Windows 2000

The release of NT following NT 4.0 was originally going to be called NT 5.0. However, in 1999, Microsoft changed the name to Windows 2000, mostly in an attempt to have a neutral name that both Windows 98 users and NT users could see as a logical next step for them. To the extent that this approach succeeds, Microsoft will have a single main operating system built on reliable 32-bit technology but using the popular Windows 98 user interface.

Since Windows 2000 really is NT 5.0, it inherits many properties from NT 4.0. It is a true 32-bit (soon to be 64-bit) multiprogramming system with individually protected processes. Each process has a private 32-bit (soon 64-bit) demand-paged virtual address space. The operating system runs in kernel mode, whereas user processes run in user mode, providing complete protection (with none of the protection flaws of Windows 98). Processes can have one or more threads, which are visible to, and scheduled by, the operating system. It has Department of Defense C2 security for all files, directories, processes, and other shareable objects (at least, if the floppy disk is removed and the network is unplugged). Finally, it also has full support for running on symmetric multiprocessors with up to 32 CPUs.

The fact that Windows 2000 really is NT 5.0 is visible in many places. For example, the system directory is called \winNT and the operating system binary (in \winNT\system32) is called ntoskrnl.exe. Right clicking on this file to examine its properties shows that its version number is 5.xxx.yyy.zzz, where the 5 stands for NT 5, xxx is the release number, yyy is the build (compilation) number, and zzz is the minor variant. Also, many of the files in \winNT and its subdirectories have NT in their names, such as ntvdm, NT's virtual MS-DOS emulator.

Windows 2000 is more than just a better NT 4.0 with the Windows 98 user interface. To start with, it contains a number of other features previously found only in Windows 98. These include complete support for plug-and-play devices, the USB bus, IEEE 1394 (FireWire), IrDA (the infrared link between portable computers and printers), and power management, among others. In addition, a number of new features not present in any other Microsoft operating system have been added, including active directory service, security using Kerberos, support for smart cards, system monitoring tools, better integration of laptop computers with desktop computers, a system management infrastructure, and job objects. Also, the main file system, NTFS, has been extended to support encrypted files, quotas, linked files, mounted volumes, and content indexing, for example. Another novel NTFS feature is the single instance store, which is a kind of copy-on-write link in which two users can share a linked file until one of them writes on it, at which time a copy is made automatically.

One of the other major improvement is internationalization. NT 4.0 came in separate versions for different languages with the text strings embedded in the code. Installing an English software package on a Dutch computer often caused parts of the operating system to stop using Dutch and start using English because certain files containing code and text strings were overwritten. This problem has been eliminated. Windows 2000 has a single binary that runs everywhere in the world. An installation, or even an individual user, can choose the language to use at run time because all the menu items, dialog strings, error reports, and other text strings have been removed from the operating system and put in separate directories, as per installed language. Like all previous versions of NT, Windows 2000 uses Unicode throughout the system to support languages not using the Latin alphabet, such as Russian, Greek, Hebrew, and Japanese.

One thing that Windows 2000 does not have is MS-DOS. It is simply not there in any form (nor was it there in NT). There is a command line interface, but this is a new 32-bit program that includes the old MS-DOS functionality and considerably new functionality as well.

Notes

Despite many portability features with regard to the code, hardware, language, etc., in one respect Windows 2000 is less portable than NT 4.0—it runs on only two platforms, the Pentium and the Intel IA-64. Originally NT supported additional platforms, including the PowerPC, MIPS, and Alpha, but over the years, Microsoft dropped one after another for commercial reasons.

Like previous versions of NT, Windows 2000 comes in several product levels, this time: Professional, Server, Advanced server, and Datacenter server. The differences between all these versions are minor however, with the same executable binary used for all versions. When the system is installed, the product type is recorded in an internal database (the registry). At boot time, the operating system checks the registry to see which version it is. The differences are shown in Figure 10.2.

Figure 10.2: The Different Versions of Windows 2000

Version	Max RAM	CPUs	Max clients	Cluster size	Optimized for
Professional	4 GB	2	10	0	Response time
Server	4 GB	4	Unlimited	0	Throughput
Advanced server	8 GB	8	Unlimited	2	Throughput
Datacenter server	64 GB	32	Unlimited	4	Throughput

As can be seen from the figure, the differences include the maximum memory supported, the maximum number of CPUs (for a multiprocessor configuration), and the maximum number of clients that can be served. The cluster size relates to the ability of Windows 2000 to make two or four machines look like a single server to the outside world, a useful feature for Web servers, for example. Finally, the default parameters are tuned differently on Professional, to favor interactive programs over batch work, although these can easily be changed if desired. One last difference is that some extra software is provided on the servers and some extra tools are provided on Datacenter server for managing large jobs.

The reason for having multiple versions is simply marketing: this allows Microsoft to charge big companies more than they charge individuals for what is essentially the same product. This idea is not new, however, and hardly unique to Microsoft. For years, airlines having been charging business passengers much more, not only for Business Class, but also for Cattle Class if they want the luxury of buying the ticket a day before the flight instead of a month before the flight.

Technically, the way the version differences are maintained is that in a few places in the code, two variables are read from the registry, **ProductType** and **ProductSuite**. Depending on their values, slightly different code is executed. Changing these variables is in violation of the license. In addition, the system traps any attempt to change them and records the attempt at tampering in an indelible way so it can be detected later.

In addition to the basic operating system, Microsoft has also developed several tool kits for advanced users. These include the Support Tools, the Software Development Kit, the Driver Development Kit, and the Resource Kit. These include a large number of utilities and tools for tweaking and monitoring the system. The support tools are on the Windows 2000 CD-ROM in the directory \support\tools. The standard installation procedure does not install them, but there is a file setup.exe in that directory that does. The SDK and DDK are available to developers at msdn.microsoft.com. The Resource Kit is a Microsoft product in a box. There are also various third-party tools available for snooping on the Windows 2000 internals, including a nice set available for free at the Website www.sysinternals.com. Some of these even provide more information than the corresponding Microsoft tools.

Notes

Windows 2000 is an immensely complex system, now consisting of over 29 million lines of C code. If printed 50 lines per page and 1000 pages per bound site, the full code would occupy 580 volumes. This opus would occupy 23 running meters of shelf space (for the paperback version). If arranged in sitecases 1 m wide with 6 shelves per sitecase, the set would occupy a wall 4 m wide.

A comparison of a few operating system source code sizes is given in Figure 10.3. However, this table should be taken with a grain (or better yet, a metric ton) of salt because what constitutes the operating system is different for different systems. For example, the entire window system and GUI is a part of the kernel in Windows, but not in any UNIX version. It is simply a user process there. Counting X Windows adds another 1.5 million lines of code to all the UNIX versions, and that does not even count the GUI code (Motif, GNOME, etc.), which is also not a part of the operating system in the UNIX world. Additionally, some systems include code for multiple architectures (e.g., five for 4.4 BSD and nine for Linux), with each architecture adding 10,000 to 50,000 lines of code. The reason Free BSD 1.0 has only 235,000 lines of code whereas 4BSD Lite, from which it is derived, has 743,000 lines it supports for all. The obsolete architectures (e.g., the VAX) was dropped in Free BSD.

Also, the number of file systems, devices drivers, and libraries supplies varies greatly from system to system. In addition, Windows contains large amounts of test code that UNIX does not contain as well as some utilities and support for numerous languaged besides English. Finally, the measurements were made by different people, which introduce considerable variance (e.g., did make files, headers, configuration files and documentation count and how much was there?). This is not like comparing apples with oranges; it is like comparing apples with telephones. However, all the counts within a single family came from the same source, so intra family counts are somewhat meaningful.

Despite all these disclaimers, two conclusions are fairly clear:

1. System bloat seems to be as inevitable as death and taxes.
2. Windows is much bigger than UNIX.

Whether small is beautiful or big is beautiful is a matter of heated controversy. The argument for the former is that small-size and a lean-and-mean mentality produces a manageable, reliable system that users can understand. The argument for the latter is that many users want lots of features. In any event, it should also be clear that any students planning to write a full-blown, state-of-the-art operating system from scratch have their work cut out for them.

Figure 10.3: A comparison of some operating system sizes. The first string in each box is the version; the second is the size measured in lines of source code, where K = 1000 and M = 1,000,000. Comparisons within a column have real meaning; comparisons across columns do not, as discussed in the text.

Year	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9K					
1979	V7 21K					
1980		4.1 38K				
1982	Sys III 58K					
1984		4.2 98K				
1986		4.3 179K				
1987	SVR3 92K		1.0 13K			
1989	SVR4 280K					
1991				0.01 10K		
1993		Free 1.0 235K			5.3 850K	3.1 6M
1994		4.4 Lite 743K		1.0 165K		3.5 10M
1996				2.0 470K		4.0 16M
1997			2.0 62K		5.6 1.4M	
1999				2.2 1M		
2000		Free 4.0 1.4M			5.8 2.0M	2000 29M

Notes

Although Windows 2000 is already the world heavyweight champion in terms of pure mass, it is still growing, with bugs being fixed and new features being added. The way Microsoft manages its development is worth noting. Hundreds of programmers work on various aspects of Windows 2000 all days. Whenever a piece of code is finished, the programmer submits it electronically to the build team. At 6 P.M. every day, the door is closed and the system is rebuilt (i.e., recompiled and linked). Each build gets a unique sequence number, which can he seen by examining the version number of *ntoskrnl.exe* (the first public release of Windows 2000 was build 2195).

The new operating system is electronically distributed to thousands of machines around the Microsoft campus in Redmond, WA, where it is subjected to intense stress tests all night. Early the next morning, the results of all the tests are sent to the relevant groups, so they can see if their new code works. Each team then decides which code they want to work on that day. During the day, the programmers work on their code and at 6 p.m. the build-and-test-cycle begins anew.

10.2 Programming Windows 2000

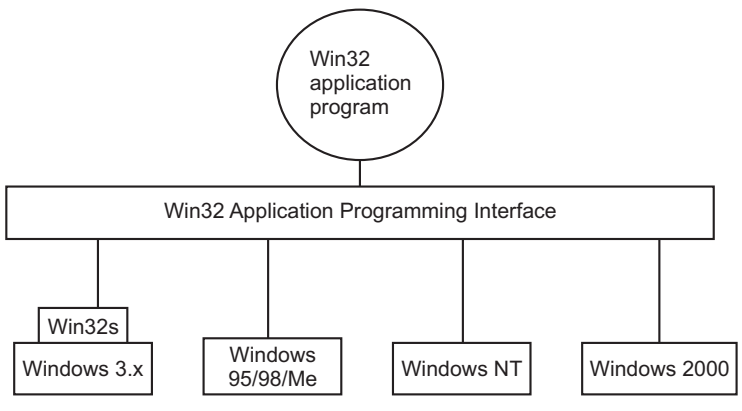
It is now time to start our technical study of Windows 2000. However, before getting into the details of the internal structure, we will first take a look at the programming interface and the registry, a small in-memory data base.

10.2.1 Win32 Application Programming Interface

Like all other operating systems, Windows 2000 has a set of system calls it can perform. However, Microsoft has never made the list of Windows system calls public, and it also changes them from release to release. Instead, what Microsoft has done is to define a set of function calls called **Win32 API (Win32 Application Programming Interface)** that are publicly known and fully documented. These are library procedures that either make system calls to get the work done, or, in some cases, do the work right in user space. The existing Win32 API calls do not change with new releases of Windows, although new API calls are added frequently.

Binary programs for the Intel x86 that adhere exactly to the Win32 API interface will run unmodified on all versions of Windows since Windows 95. As shown in Figure 10.4, an extra library is needed for Windows 3.x to match a subset of the 32-bit API calls to the 16-bit operating system, but for the other systems no adaptation is needed. It should be noted that Windows 2000 adds substantial new functionality to Win32, so it has additional API calls not included on older versions of Win32 and which will not work on older versions of Windows.

Figure 10.4: The Win32 API Allows Programs to Run on Almost all Versions of Windows



Notes

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions (i.e., procedures) that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, and so on. Every call creating an object returns a result called a handle to the caller. This handle can subsequently be used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object also has a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Not all system-created data structures are objects and not all objects are kernel objects. The only ones that are true kernel objects are those that need to be named, protected, or shared in some way. Every kernel object has a system-defined type, has well-defined operations on it, and occupies storage in kernel memory. Although users can perform the operations (by making Win32 calls), they cannot get at the data directly.

The operating system itself can also create and use objects and does so heavily. Most of these objects are created to allow one component of the system to store some information for a substantial period of time or to pass some data structure to another component. For example, when a device driver is loaded, an object is created for it holding its properties and pointers to the functions it contains. Within the operating system, the driver is then referred to by using its object.

Windows 2000 is sometimes said to be object-oriented because the only way to manipulate objects is by invoking operations on their handles by making Win32 API calls. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

The Win32 API calls cover every conceivable area an operating system could deal with, and quite a few of it arguably should not deal with. Naturally, there are calls for creating and managing processes and threads. There are also many calls that relate to interprocess (actually, interthread) communication, such as creating, destroying, and using mutexes, semaphores, events, and other IPC objects.

Although much of the memory management system is invisible to programmers (fundamentally, it is just demand paging), one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows the process the ability to read and write parts of the file as though they were memory words.

An important area for many programs is file I/O. In the Win32 view, a file is just a linear sequence of bytes. Win32 provides over 60 calls for creating and destroying files and directories, opening and closing files, reading and writing them, requesting and setting file attributes, and much more.

Another area for which Win32 provides calls is security. Every process has an ID telling who it is and every object can have an access control list telling in great detail precisely which users may access it and which operations they may perform on it. This approach provides for a fine-grained security in which specific individuals can be allowed or denied specific access to every object.

Notes

Processes, threads, synchronization, memory management, file I/O, and security system calls are nothing new. Other operating systems have them too, although generally not hundreds of them, as Win32 does. But what really distinguishes Win32 are the thousands upon thousands of calls for the graphical interface. There are calls for creating, destroying, managing and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for drawing geometric figures, filling them in, managing the colour palettes they use, dealing with fonts, and placing icons on the screen. Finally, there are calls for dealing with the keyboard, mouse and other input devices as well as audio, printing, and other output devices. In short, the Win32 API (especially the GUI part) is immense and we could not even begin to describe it in any detail in this unit, so we will not try. Interested readers should consult one of the many sites on Win32.

Although the Win32 API is available on Windows 98 (as well as on the consumer electronics operating system, Windows CE), not every version of Windows implements every call and sometimes there are minor differences as well. For example, Windows 98 does not have any security, so those API calls that relate to security just return error codes on Windows 98. Also, Windows 2000 file names use the Unicode character set, which is not available on Windows 98 and Windows 98 file names are not case sensitive, whereas Windows 2000 file names are case sensitive (although some kinds of searches on file names are not case sensitive). There are also differences in parameters to some API function calls. On Windows 2000, for example, all the screen coordinates given in the graphics functions are true 32-bit numbers; on Windows 95, only the low-order 16 bits are used because much of the graphics subsystem is still 16-bit code. The existence of the Win32 API on several different operating systems makes it easier to port programs between them, but since these minor variations exist, some care must be taken to achieve portability.



Task How to differentiate between Win32 and Win64?

10.2.2 Registry

Windows needs to keep track of a great deal of information about hardware, software, and users. In Windows 3.x, this information was stored in hundreds of .int (initialization) files spread all over the disk. Starting with Windows 95, nearly all the information needed for booting and configuring the system and tailoring it to the current user was gathered in a big central database called the **registry**. In this section we will give an overview of the Windows 2000 registry.

To start with, it is worth noting that although many parts of Windows 2000 are complicated and messy, the registry is one of the worst, and the cryptic nomenclature does not make it much better. Fortunately, entire sites have been written describing it. That said, the idea behind the registry is very simple. It consists of a collection of directories, each of which contains either subdirectories or entries. In this respect it is a kind of file system for very small files. It has directories and entries (the files).

The confusion starts with the fact that Microsoft calls a directory a **key**, which is definitely not. Furthermore, all the top-level directories start with the string HKEY, which means handle to key. Subdirectories tend to have somewhat better chosen names, although not always.

At the bottom of the hierarchy are the entries, called **values**, which contain the information. Each value has three parts: a name, a type, and the data. The name is just a Unicode string, often default if the directory contains only one value. The type is one of 11 standard types. The most common ones are Unicode string, a list of Unicode strings, a 32-bit integer an arbitrary length

Notes

binary number, and a symbolic link to a directory or entry elsewhere in the registry. Symbolic names are completely analogous to symbolic links in file systems or shortcuts on the Windows desktop: they allow one entry to point to another entry or directory. A symbolic link can also be used as a key, meaning that something that appears to be a directory is just a pointer to a different directory.

Figure 10.5: The root keys registry keys and selected subkeys. The capitalization has no meaning but follows the Microsoft practice here

Key	Description
HKEY_LOCAL_MACHINE	Properties of the hardware and software
HARDWARE	Hardware description and mapping of hardware to drivers
SAM	Security and account information for users
SECURITY	System-wide security policies
SOFTWARE	Generic information about installed application programs
SYSTEM	Information for booting the system
HKEY_USERS	Information about the users; one subkey per user
USER_AST_ID	User AST's profile
AppEvents	Which sound to make when (incoming email/fax, error, etc.)
Console	Command prompt settings (colours, fonts, history, etc.)
Control Panel	Desktop appearance, screensaver, mouse ensitivity, etc.
Environment	Environment variables
Keyboard Layout	Which keyboard: 102-key US, AZERTY, Dvorak, etc.
Printers	Information about installed printers
Software	User preferences for Microsoft and third party software
HKEY_PERFORMANCE_DATA	Hundreds of counters monitoring system performance
HKEY_CLASSES_ROOT	Link to HKEY_LOCAL_MACHINE\SOFTWARE CLASSES
HKEY_CURRENT_CONFIG	Link to the current hardware profile
HKEY_CURRENT_USER	Link to the current user profile

At the top level, the Windows 2000 registry has six keys, called **root keys**, as listed in Figure 10.5. Some interesting **sub keys** (subdirectories) are also shown here. To see this list on your system, use one of the registry editors, either regedit or regedt32, which unfortunately display different information and use different formats. They can also change registry values. Amateurs should not change the keys or values on any system they plan to boot again. Just looking is safe, though. You have been warned.

The first key (i.e., directory), HKEY_LOCAL_MACHINE, is probably the most important as it contains all the information about the local system. It has five subkeys (i.e., subdirectories). The HARDWARE subkey contains many subkeys telling all about the hardware and which driver controls which piece of hardware. This information is displayed on the fly by the plug-and-play manager as the system boots. Unlike the other subkeys, it is not stored on disk.

Notes

The SAM (Security Account Manager) subkey contains the user names, groups, passwords, and other account and security information needed for logging in. The SECURITY subkey contains general security policy information, such as minimum length for passwords, how many failed login attempts are tolerated before the police is called, etc.

The SOFTWARE subkey is where software manufacturers store preferences etc. For example, if a user has Adobe Acrobat, Photoshop and Premiere installed, there will be a subkey Adobe here, and below that further subkeys for Acrobat, Photoshop, Premiere, and any other Adobe products. The entries in these subdirectories can store anything the Adobe programmers want to put there, generally system-wide properties such as the version and build number, how to uninstall the package, drivers to use, and so forth. The registry saves them the trouble of having to invent their own method for storing this information. User-specific information also goes in the registry, but under HKEY_USERS.

The SYSTEM subkey holds most of the information about booting the system, for example, the list of drivers that must be loaded. It also holds the list of services (daemons) that must be started after booting up and the configuration information for all of them.

The next top-level key is HKEY_USERS, which contains the profiles of all the users. All the user-specific preferences in a number of areas are stored here. When a user changes a preference using the control panel, for example, the desktop colour scheme, the new settings are recorded here. In fact, many of the programs on the control panel do little more than collect user information and change the registry accordingly. Some of the subkeys under HKEY_USERS are shown in Figure 10.5 and should need little additional comment. Some of the subkeys, such as Software, contain surprisingly large numbers of subkeys, even if no software packages are installed.

The next top-level key, HKEY_PERFORMANCE_DATA contains neither data read in from the disk nor data collected by the plug-and-play manager. Instead, it offers a window into the operating system. The system itself contains hundreds of counters for monitoring system performance. These counters are accessible via this registry key. When a subkey is queried, a specified procedure is run to collect and return the information (possibly by reading one or more counters and combining them in some way). This key is not visible using regedit or regedt32. Instead one has to use the performance tools, such as pfmon, perfmon, and pview. There are many such tools, some on the Windows 2000 CD-ROM, some in the resource kits, and some from third parties.

The next three top-level keys do not actually exist. Each one is a symbolic link to some place elsewhere in the registry. The HKEY_CLASSES_ROOT key is the most interesting. It points to the directory that handles COM (Component Object Model) objects and also the associations between file extensions and programs. When a user double clicks on a file ending in, say, .doc, the program catching the mouse click looks here to see which program to run (probably Microsoft Word). The complete database of recognized extensions and which program each one is owned by is under this key.

The HKEY_CURRENT_CONFIG key links to the current hardware configuration. A user can construct multiple hardware configurations, for example by disabling various devices to see if they were the cause of strange system behaviour. This key points to the current configuration. Similarly, HKEY_CURRENT_USER points to the current user so that user's preferences can be found quickly.

None of the last three keys really adds anything, since the information was available elsewhere anyway (although less conveniently accessible). Thus despite the fact that regedit and regedt32 list five top-level keys, there are really only three top-level directories and one of them is not shown among the five displayed.

Notes

The registry is fully available to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Figure 10.6.

Figure 10.6: Some of the Win32 API calls for Using the Registry

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

When the system is turned off, most of the registry information (but not all, as discussed above) is stored on the disk in files called **hives**. Most of them are in \winnt\system32\config. Because their integrity is so critical to correct system functioning, when they are updated, backups are made automatically and writes are done using atomic transactions to prevent corruption in the event of system crash during the write. Loss of the registry requires reinstalling all software.

Self Assessment

Multiple choice questions:

1. ..... is a line of operating systems produced by Microsoft for use on personal computers, business desktops, laptops and servers.
   (*a*) Windows 95 (*b*) Windows 98
   (*c*) Windows 2000 (*d*) Windows Me
2. Window 95 was released in ..... .
   (*a*) August 1995 (*b*) June 1998
   (*c*) April 2000 (*d*) May 2000
3. In the millennium year 2000, Microsoft brought out a minor revision to Windows 98 called ..... .
   (*a*) Windows 95 (*b*) Windows NT
   (*c*) Windows 2000 (*d*) Windows Me
4. One thing that Windows 2000 does not have is ..... .
   (*a*) Server (*b*) MS-DOS
   (*c*) System32 (*d*) Windows NT
5. At the top level, the Windows 2000 registry has ..... keys, called root keys.
   (*a*) 2 (*b*) 4
   (*c*) 6 (*d*) 8

10.3 Summary

- Windows 2000 which includes Microsoft Operating Systems for desktop and laptop.
- PCs can be divided into three families, i.e. MS-DOS, Consumer Windows and Windows NT.

Notes

- Windows NT (Windows New Technology) was intended for mission critical business applications as well as for home users.
- Programming (Windows 2000), which includes the Win32 application programming interface which tells us that Microsoft has never made the list of Windows system calls public and it also changes them from release to release.
- Windows 98 and Windows NT, different versions of Windows 2000 and the description of the key.
- The SAM (Security Account Manager) subkey contains the user names, groups, passwords and other accounts and security information needed for logging in.

10.4 Keywords

Hives: When the system is turned off, most of the registry information is stored on the disk files called hives.

Registry: In Windows 95, nearly all the information needed for booting and configurating the system and tailoring it to the current user was gathered in a big central database called the registry.

Win 32API: Windows 2000 has a set of system calls it can perform. However, Microsoft has never made the list of Windows system calls public and it also changes them from release to release. Instead, what Microsoft has done is define a set of function calls called the Win32 API (Win32 Application Programming Interface).

Windows Me: In the millennium year, 2000, Microsoft brought out a minor revision to Windows 98 called Windows Me (Windows Millennium Edition).

Windows: Microsoft decided to give MS-DOS a graphical user interface (shell) that is called Windows.

10.5 Review Questions

1. When the kernel catches system call, how does it know which system call it is supposed to carry out?
2. Define Windows NT and describe why it is named so.
3. Describe in detail Windows 2000 and its versions.
4. What are the differences between Windows 98 and Windows NT? Describe.
5. Define Programming Windows 2000 and its structures.

Answers to Self Assessment

1. (*c*) 2. (*a*) 3. (*d*) 4. (*b*) 5. (*c*)

10.6 Further Readings



Books

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes. *Introduction to Operating Design and Implementation,* by Michael Kifer, Scoott A. Smolka.



Online link

wiley.com/coolege.silberschatz

Notes

Unit 11: Operating System Structure

CONTENTS

Objectives

Introduction

11.1 Operating System Structure

11.1.1 Hardware Abstraction Layer

11.1.2 Kernel Layer

11.1.3 Executive

11.1.4 Device Drivers

11.1.5 Implementation of Objects

11.1.6 Object Name Space

11.1.7 Environment Subsystems

11.2 Summary

11.3 Keywords

11.4 Review Questions

11.5 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain operating system structure
- Discuss the hardware abstraction layer
- Explain kernel layer
- Explain executive
- Understand the device drivers
- Explain implementation of objects
- Discuss object name space
- Explain environment subsystems

Notes

Introduction

In this unit, we will briefly look at the operating systems from its functional point of view; that is the services which are provided by the operating system. A detailed discussion will follow in the subsequent units.

Common System Components

- Due to the complex nature of the modern operating systems, it is partitioned into smaller component. Each component performs a well-defined function with well-defined inputs and outputs.
- Many modern operating systems have the following components.
  - Process Management
  - Main Memory Management
  - File Management
  - I/O System Management
  - Secondary Management
  - Networking
  - Protection System
  - Command-Interpreter System

11.1 Operating System Structure

Windows 2000 consists of two major parts: the operating system itself, which runs in kernel mode, and the environment subsystems, which run in user mode. The kernel is a traditional kernel in the sense that it handles process management, memory management, file systems, and so on. The environment subsystems are somewhat unusual because they are separate processes that help user programs carry out certain system functions. In the following sections we will examine each of these parts in turn.

One of NT's many improvements over Windows 3.x was its modular structure. It consisted of a moderately small kernel that ran in kernel mode, plus some server processes that ran in user mode. User processes interacted with the server processes using the client-server model: a client sent a request message to a server, and the server did the work and returned the result to the client via a second message. This modular structure made it easier to port it to several computers besides the Intel line, including the DEC Alpha, IBM PowerPC, and SGI MIPS. It also protected the kernel from bugs in the server code. However, for performance reasons, starting with NT 4.0, pretty much all of the operating system (e.g., system call handling and all of the screen graphics) was put back into kernel mode. This design was carried over to Windows 2000.

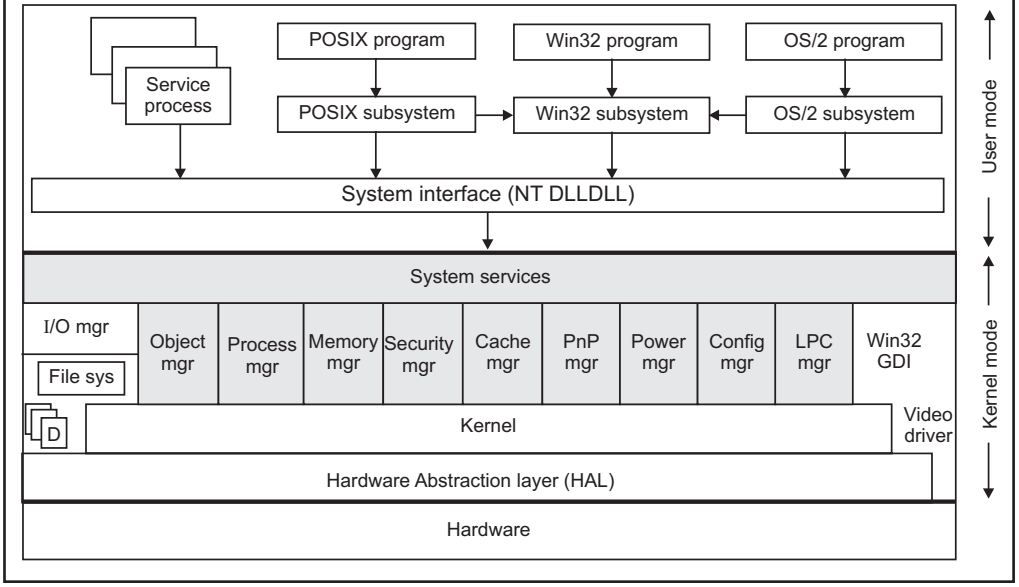
Nevertheless, there is still some structure in Windows 2000. It is divided into several layers, each one using the services of the ones beneath it. The structure is illustrated in Figure 11.1. One of the layers is divided horizontally into many modules. Each module has some particular function and a well-defined interface to the other modules.

The lowest two software layers, the HAL and the kernel, are written in C and in assembly language and are partly machine dependent. The upper ones are written entirely in C and are almost entirely machine independent. The drivers are written in C, or in a few cases C++.

Notes

Below we will first examine the various components of the system starting at the bottom and working our way up.

Figure 11.1: The structure of Windows 2000 (slightly simplified). The shaded area is the executive. The boxes indicated by D are device drivers. The service processes are system daemons.





Did u know?

The software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system (OS) acts as a host for application programs that are run on the machine.

11.1.1 Hardware Abstraction Layer

One of the goals of Windows 2000 (and Windows NT before it) was to make the operating system portable across platforms. Ideally, when a new machine comes along, it should be possible to just recompile the operating system with the new machine's compiler and have it run for the first time. Unfortunately, the upper layers of the operating system can be made completely portable (because they mostly deal with internal data structures), while the lower layers deal with device registers, interrupts, DMA, and other hardware features that differ appreciably from machine to machine. Even though most of the low-level code is written in C, it cannot just be scooped up from a Pentium, plopped down on, say, an Alpha, recompiled, and rebooted due to the many small hardware differences between the Pentium and the Alpha that have nothing to do with the different instruction sets and which cannot be hidden by the compiler.

Fully aware of this problem, Microsoft made a serious attempt to hide many of the machine dependencies in a thin layer at the bottom called the **HAL (Hardware Abstraction Layer)**. (The name HAL was inspired by the computer HAL in the late Stanley Kubrick's movie 2001: A Space Odyssey. Rumor has it that Kubrick chose the name "HAL" by taking the name of the then-dominant computer company—IBM—and subtracting 1 from each letter.)

Notes

The job of the HAL is to present the rest of the operating system with abstract hardware devices, in particular, devoid of the warts and idiosyncracies with which real hardware is so richly endowed. These devices are presented in the form of machine-independent services (procedure calls and macros) that the rest of the operating system and the drivers can use. By using the HAL services (which are identical on all Windows 2000 systems, no matter what the hardware is) and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new hardware. Porting the HAL itself is straightforward because all the machine-dependent codes are concentrated in one place and the goals of the port are well defined, namely, implement all of the HAL services.

The services chosen for inclusion in the HAL are those that relate to the chip set on the parentboard and which vary from machine to machine within reasonably predictable limits. In other words, it is designed to hide the differences between one vendor's parentboard and another one's, but not the differences between a Pentium and an Alpha. The HAL services include access to the device registers, bus-independent device addressing, interrupt handling and resetting, DMA transfers, control of the timers and real-time clock, low-level spin locks and multiprocessor synchronization, interfacing with the BIOS and its CMOS configuration memory. The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, or disks or for the memory management unit.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O versus I/O ports. Some machines have one and some have the other. How should a driver be programmed: to use memory-mapped I/O or not? Rather than forcing a choice, which would make the driver not portable to a machine that did it the other way, the hardware abstraction layer offers three procedures for driver writers to use for reading the device registers and another three for writing them:

```
uc = READ_PORT_UCHAR(port);         WRITE_PORT_UCHAR(port, uc);

us = READ_PORT_USHORT(port);        WRITE_PORT_USHORT(port, us);

ul = READ_PORT_ULONG(port);         WRITE_PORT_LONG(port, ul);
```

These procedures read and write unsigned 8-, 16-, and 32-bit integers, respectively, to the specified port. It is up to the hardware abstraction layer to decide whether memory-mapped I/O is needed here. In this way, a driver can be moved without modification between machines that differ in the way the device registers are implemented.

Drivers often need to access specific I/O devices for various purposes. At the hardware level, a device has one or more addresses on a certain bus. Since modern computers often have multiple buses (ISA, PCI, SCSI, USB, 1394, etc.), it can happen that two or more devices have the same bus address, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto system-wide logical addresses. In this way, drivers are not required to keep track of which device is on which bus. These logical addresses are analogous to the handles the operating system gives user programs to refer to files and other system resources. This mechanism also shields higher layers from properties of alternative bus structures and addressing conventions.

Interrupts have a similar problem—they are also bus dependent. Here, too, the HAL provides services to name the interrupts in a system-wide way and also provides services to allow drivers to attach interrupt service routines to interrupts in a portable way, without having to know anything about which interrupt vector is for which bus. Interrupt request level management is also handled in the HAL.

Another HAL service is setting up and managing DMA transfers in a device-independent way. Both the system—wide DMA engine and DMA engines on specific I/O cards can be handled.

Notes

Devices are referred by their logical addresses. The HAL also implements software scatter/gather (writing or reading from noncontiguous blocks of physical memory).

The HAL also manages clocks and timers in a portable way. Time is kept track of in units of 100 nsec starting at 1 January 1901, which is far more precise than MS-DOS's keeping track of time in units of 2 sec since 1 January 1980 and provides support for the many computer-related activities in the 17th, 18th, and 19th centuries. The time services decouple the drivers from the actual frequencies at which the clocks run.

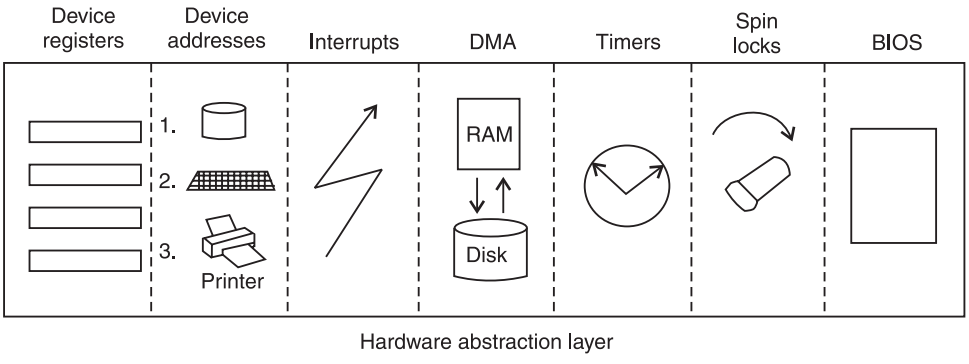
Kernel components sometimes need to synchronize at a very low level, especially to prevent race conditions in multiprocessor systems. The HAL provides some primitives to manage this synchronization, such as spin locks, in which one CPU simply waits for a resource held by another CPU to be released, particularly in situations where the resource is typically only held for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the BIOS and inspects the CMOS configuration memory, if any, to find out which buses and I/O devices the system contains and how they have been configured. This information is then put into the registry so other system components can look it up without having to understand how the BIOS or configuration memory work. A summary of some of the things the HAL does is given in Figure 11.2.

Since the HAL is highly-machine dependent, it must match the system it is installed on perfectly, so a variety of HALs are provided on the Windows 2000 CD-ROM. At system installation time, the appropriate one is selected and copied to the system directory \winNT\system32 on the hard disk as hal.dll. All subsequent boots use this version of the HAL. Removing this file will make the system unbootable.

Although the HAL is reasonably efficient, for multimedia applications, it may not be fast enough. For this reason, Microsoft also produced a software package called **DirectX**, which augments the HAL with additional procedures and allows user processes much more direct access to the hardware. DirectX is somewhat specialized, so we will not discuss it further in this unit.

Figure 11.2: Some of the Hardware Functions the HAL Manages





Hardware refers to a physical piece of a computer. This could be a hard drive, monitor, memory chip, or CPU. The key idea is that the item is something you can touch.

Notes

11.1.2 Kernel Layer

Above the hardware abstraction layer is a layer that contains what Microsoft calls the **kernel**, as well as the device drivers. Some early documentation refers to the kernel as the "microkernel," which it really never was because the memory manager, file system, and other major components resided in kernel space and ran in kernel mode from day one. The kernel is certainly not a microkernel now since virtually the entire operating system was put in kernel space starting with NT 4.0.

In the unit on UNIX, we used the term "kernel" to mean everything running in kernel mode. In this unit, we will reserve the term "kernel" for the part labeled as such in Figure 11.2 and call the totality of the code running in kernel mode the "operating system." Part of kernel (and much of the HAL) is permanently resident in main memory (i.e., is not paged). By adjusting its priority, it can control whether it can tolerate being preempted by I/O interrupts or not. Although a substantial fraction of the kernel is machine specific, most of it is nevertheless written in C, except where top performance overshadows all other concerns.

The purpose of the kernel is to make the rest of the operating system completely independent of the hardware, and thus highly portable. It picks up where the HAL leaves off. It accesses the hardware via the HAL and builds upon the extremely low-level HAL services to construct higher-level abstractions. For example, the HAL has calls to associate interrupt service procedures with interrupts, and set their priorities, but does little else in this area. The kernel, in contrast, provides a complete mechanism for doing context switches. It properly saves all the CPU registers, changes the page tables, flushes the CPU cache, and so on, so that when it is done, the previously running thread has been saved in tables in memory. It then sets up the new thread's memory map and loads its registers so the new thread can start running.

The code for thread scheduling is also in the kernel. When it is time to see if a new thread can run, for example, after a quantum runs out or after an I/O interrupt completes, the kernel chooses the thread and does the context switch necessary to run it. From the point of view of the rest of the operating system, thread switching is automatically handled by lower layers without any work on their part and in a portable way. The scheduling algorithm itself will be discussed later in this unit when we come to processes and threads.

In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel also has another key function: providing low-level support for two classes of objects: control objects and dispatcher objects. These objects are not the objects that user processes get handles to, but are internal objects upon which the executive builds the user objects.

Control objects are those objects that control the system, including primitive process objects, interrupt objects, and two somewhat strange objects called DPC and APC. A **DPC (Deferred Procedure Call)** object is used to split off the non-time-critical part of an interrupt service procedure from the time critical part. Generally, an interrupt service procedure saves a few volatile hardware registers associated with the interrupting I/O device so they do not get overwritten and re-enables the hardware, but saves the bulk of the processing for later.

For example, after a key is struck, the keyboard interrupt service procedure reads the key code from a register and re-enables the keyboard interrupt, but does not need to process the key immediately, especially if something more important (i.e., higher priority) is currently going on. As long as the key is processed within about 100 msec, the user will be none the wiser. DPCs are also used for timer expirations and other activities whose actual processing need not be instantaneous. The DPC queue is the mechanism for remembering that there is more work to do later.

Notes

Another kernel control object is the **APC (Asynchronous Procedure Call)**. APCs are like DPCs except that they execute in the context of a specific process. When processing a key press, it does not matter on whose context the DPC runs in because all that is going to happen is that the key code will be inspected and probably put in a kernel buffer. However, if an interrupt requires copying a buffer from kernel space to a buffer in some user process address space (e.g., as it may on completion of a read from the modem), then the copying procedure needs to run in the receiver's context. The receiver's context is needed so the page table will contain both the kernel buffer and the user buffer (all processes contain the entire kernel in their address spaces, as we will see later). For this reason, the kernel distinguishes between DPCs and APCs.

The other kind of kernel objects are **dispatcher objects**. These include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on. The reason that these have to be handled (in part) in the kernel is that they are intimately intertwined with thread scheduling, which is a kernel task. As a little aside, mutexes are called "mutants" in the code because they were required to implement the OS/2 semantics of not automatically unlocking themselves when a thread holding one exiled, something the Windows 2000 designers considered bizarre. (The OS/2 semantics are relevant because NT was originally conceived of as a replacement for OS/2, the operating system shipped on IBM's PC/2.)

11.1.3 Executive

Above the kernel and device drivers is the upper portion of the operating system, called the **executive**, shown as the shaded area in Figure 11.1. The executive is written in C, is architecture independent, and can be ported to new machines with relatively little effort. It consists of 10 components, each of which is just a collection of procedures that work together to accomplish some goal. There are no hard boundaries between the pieces and different authors describing the executive might even group the procedures differently into components. It should be noted that components on the same level can (and do) call each other extensively.

The **object manager** manages all objects known to the operating system. These include processes, threads, files, directories, semaphores, I/O devices, timers, and many others. The object manager allocates a block of virtual memory from kernel address space when an object is created and returns it to the free list when the object is deal-located. Its job is to keep track of all the objects.

To avoid any confusion, most of the executive components labeled "manager" in Figure 11.1 are not processes or threads, but merely collections of procedures that other threads can execute when in kernel mode. A few of them, such as the power manager and plug-and-play manager, really are independent threads though.

The object manager also manages a name space in which newly created objects may be placed so they can be referred to later. All other components of the executive use objects heavily to do their work. Objects are so central to the functioning of Windows 2000 that they will be discussed in detail in the next section.

The **I/O manager** provides a framework for managing I/O devices and provides generic I/O services. It provides the rest of the system with device-independent I/O, calling the appropriate driver to perform physical I/O. It is also home to all the device drivers (indicated by D in Figure 11.1). The file systems are technically device drivers under control of the I/O manager. Two different ones are present for the FAT and NTFS file systems, each one independent of the others and controlling different disk partitions. All the FAT file systems are managed by a single driver.

The **process manager** handles processes and threads, including their creation and termination. It deals with the mechanisms used to manage them, rather than policies about how they are

Notes

used. It builds upon the kernel process and thread objects and adds extra functionality to them. It is the key to multiprogramming in Windows 2000.

The **memory manager** implements Windows 2000's demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames. It thereby enforces the protection rules that restrict each process to only access those pages belonging to its address space and not to other processes' address spaces (except under special circumstances).

The **security manager** enforces Windows 2000's elaborate security mechanism, which meets the U.S. Department of Defense's Orange site C2 requirements. The Orange site specifies a large number of rules that a conforming system must meet, starting with authenticated login through how access control is handled, to the fact that virtual pages must be zeroed out before being reused.

The **cache manager** keeps the most recently used disk blocks in memory to speed up access to them in the (likely) event that they are needed again. Its job is to figure out which blocks are probably going to be needed again and which ones are not. It is possible to configure Windows 2000 with multiple file systems, in which case the cache manager works for all of them, so each one does not have to do its own cache management. When a block is needed, the cache manager is asked to supply it. If it does not have the block, the cache manager calls upon the appropriate file system to get it. Since files can be mapped into processes' address spaces, the cache manager must interact with the virtual memory manager to provide the necessary consistency. The amount of space devoted to caching is dynamic and can increase or decrease as demands on it change.

The **plug-and-play manager** is sent all notifications of newly attached devices. For some devices, a check is made at boot time and not thereafter. Other devices, for example, USB devices, can be attached at any time and their attachment triggers a message to the plug-and-play manager, which then locates and loads the appropriate driver.

The **power manager** rides herd on power usage. This consists of turning off the monitor and disks after they have been idle for a while. On laptops, the power manager monitors battery usage and takes action when the battery is about to run dry. Such action typically tells programs to save their files and prepare for a graceful shutdown.

The **configuration manager** is incharge of the registry. It adds new entries and looks up keys when asked to.

The **local procedure call manager** provides for a highly-efficient interprocess communication used between processes and their subsystems. Since this path is needed to carry out some system calls, efficiency is critical here, which is why the standard interprocess communication mechanisms are not used.

The Win32 GDI executive module handles certain system calls (but not all of them). It was originally in user space but was moved to kernel space in NT 4.0 to improve performance. The **GDI (Graphics Device Interface)** handles image management for the monitor and printers. It provides system calls to allow user programs to write on the monitor and printers in a device-independent way. It also contains the window manager and display driver. Prior to NT 4.0, it, too, was in user space but the performance was disappointing, so Microsoft moved it into the kernel to speed it up. It is worth mentioning that Figure 11.1 is not at all to scale. For example, the Win32 and graphics device interface module are larger than the rest of the executive combined.

At the top of the executive is a thin layer called **system services**. Its function is to provide an interface to the executive. It accepts the true Windows 2000 system calls and calls other parts of the executive to get them executed.

Notes

At boot time, Windows 2000 is loaded into memory as a collection of files. The main part of the operating system, consisting of the kernel and executive, is located in the file ntoskrnl.exe. The HAL is a shared library located in a separate file, hal.dll. The Win32 and graphics device interface are together in a third file, win32k.sys. Finally, many device drivers are also loaded. Most of these have extension .sys.

Actually, the ntoskrnl.exe file comes in uniprocessor and multiprocessor versions. Also, there are versions for the Xeon processor, which can have more than 4 GB of physical memory and the Pentium, which cannot. Finally, versions can consist of a free build (sold in stores and preinstalled by computer manufacturers) or a checked build (for debugging purposes). together there could be eight combinations, although two pairs were combined leaving only six. One of these is copied to ntoskrnl.exe when the system is installed.

The checked builds are worth a few words. When a new I/O device is installed on a PC, there is invariably a manufacturer-supplied driver that has to be installed to make it work. Suppose that an IEEE 1394 card is installed on a computer and appears to work fine. Two weeks later the system suddenly crashes. To whom does the owner blame? Microsoft?

The bug may indeed be Microsoft's, but some bugs are actually due to flakey drivers, over which Microsoft has no control and which are installed in kernel memory and have full access to all kernel tables as well as the entire hardware. In an attempt to reduce the number of irate customers on the phone, Microsoft tries to help driver writers debug their code by putting statements of the form. ASSERT(some condition) throughout the code. These statements make sanity checks on all parameters to internal kernel procedures (which may be freely called by drivers) and make many other checks as well. The free builds have ASSERT defined as a macro that does nothing, removing all the checks. The checked builds have defined it as:

#define ASSERT(a) if (!(a)) error(…)

causing all the checks to appear in the ntoskrnl.exe executable code and be carried out at run time. While this slows down the system enormously, it helps driver writers debug their drivers before they ship them to customers. The checked builds also have numerous other debugging features turned on.


Did u know?

A kernel can be contrasted with a shell, which is the outermost part of an operating system and a program that interacts with user commands.

11.1.4 Device Drivers

The last part of Figure 11.1 consists of the **device drivers**. Each device driver can control one or more I/O devices, but a device driver can also do things not related to a specific device, such as encrypting a data stream or even just providing access to kernel data structures. Device drivers are not part of the ntoskrnl.exe binary. The advantage of this approach is that once a driver has been installed on a system, it is added to a list in the registry and is loaded dynamically when the system boots. In this way, ntoskrnl.exe is the same for everyone, but every system is configured precisely for those devices it contains.

There are device drivers for macroscopically visible I/O devices such as disks and printers, but also for many internal devices and chips that practically no one has ever heard of. In addition, the file systems are also present as device drivers, as mentioned above. The largest device driver, the one for Win32, GDI, and video, is shown on the far right of Figure 11.1. It handles

Notes

many system calls and most of the graphics. Since customers can install new device drivers, they have the power to affect the kernel and corrupt the system. For this reason, drivers must be written with great care.

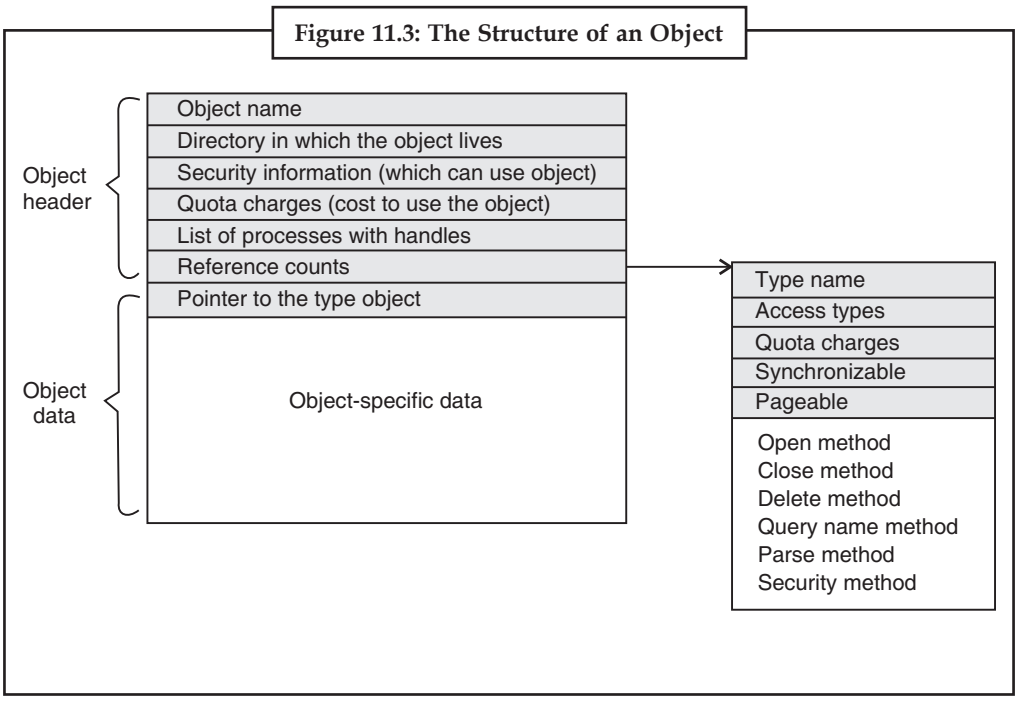
11.1.5 Implementation of Objects

Objects are probably the single most important concept in Windows 2000. They provide a uniform and consistent interface to all system resources and data structures such as processes, threads, semaphores, etc. This uniformity has various facets. First, all objects are named and accessed in the same way, using object handles. Second, because all accesses to the objects go through the object manager, it is possible to put all the security checks in one place and ensure that no process can make an end run around them. Third, sharing of objects among processes can be handled in a uniform way. Fourth, since all object opens and closes go through the object manager, it is easy to keep track of which objects are still in use and which can be safely deleted. Fifth, this uniform model for object management makes it easy to manage resource quotas in a straightforward way.

A key to understanding objects is to realize that an (executive) object is just some number of consecutive words in memory (i.e., in kernel virtual address space). An object is a data, structure in RAM, no more and no less. A file on disk is not an object, although an object (i.e., a data structure in kernel virtual address space) is created for a file when it is opened. A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or crashes) all objects are lost. In fact, when the system boots, there are no objects present at all (except for the idle and system processes, whose objects are hardwired into the ntoskrnl.exe file). All other objects are created on the fly as the system boots up and various initialization (and later user) programs run.

Objects have a structure, as shown in Figure 11.3. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in object space, security information (so a check can be made when an object is opened), and a list of processes with open handles to the object (if a certain debugging flag is enabled).

Figure 11.3: The Structure of an Object



Notes

Each object header also contains a quota charge field, which is the charge levied against a process for opening the object. If a file object costs 1 point and a process belongs to a job that has 10 file points worth of quota, the processes in that job can only open 10 files in total. In this way resource limits can be enforced for each object type separately.

Objects occupy valuable real estate—pieces of kernel virtual address space, so when an object is no longer needed it should be removed and its address space reclaimed. The mechanism for reclamation is to have a reference counter in each object's header. It counts the number of open handles held by processes. This counter is incremented every time the object is opened and decremented every time it is closed. When it hits 0, no more users hold handles to the object. When an object is acquired or released by an executive component, a second counter is incremented or decremented, even though no actual handle is issued. When both counters hit 0, no user process is using the object and no executive process is using the object, so the object can be removed and its memory freed.

The object manager needs to maintain dynamic data structures (its objects), but it is not the only part of the executive with this need. Other pieces also need to allocate and release chunks of kernel memory dynamically. To meet these needs, the executive maintains two page pools in kernel address space—for objects and for other dynamic data structures. Such pools operate as heaps, similar to the C language calls malloc and free for managing dynamic data. One pool is paged and the other is nonpaged (pinned in memory). Objects that are needed often are kept in the nonpaged pool; objects that are rarely accessed, such as registry keys and some security information, are kept in the paged pool. When memory is tight, the latter can be paged out and faulted back on demand. In fact, substantial portions of the operating system code and data structures are also pageable, to reduce memory consumption. Objects that may be needed when the system is running critical code (and when paging is not permitted) must go in the nonpaged pool. When a small amount of storage is needed, a page can be taken from either pool and then broken up into units as small as 8 bytes.

Objects are typed, which means each one has certain properties common to all objects of its type. The type is indicated by a pointer in the header to a type object, as shown in Figure 11.3. The type object information includes items such as the type name, whether a thread can wait on the object (yes for mutexes, no for open files), and whether new objects of this type go on the paged or nonpaged pool. Each object points to its type object.

The last thing a type object has is also the most important: pointers to the code for certain standard operations such as open, close, and delete. Whenever one of these operations is invoked on an object, the pointer to the type object is followed and the relevant code located and executed. This mechanism gives the system the opportunity to initialize new objects, and recover storage when they are deleted.

Executive components can create new types dynamically. There is no definite list of object types, but some of the more common ones are listed in Figure 11.4. Let us briefly go over the object types in Figure 11.4. Process and thread are obvious. There is one object for every process and every thread, which holds the main properties needed to manage the process or thread. The next three objects, semaphore, mutex, and event, all deal with interprocess synchronization. Semaphores and mutexes work as expected, but with various extra bells and whistles (e.g., maximum values and timeouts). Events can be in one of two states: signaled or nonsignaled. If a thread waits on an event that is in signaled state, the thread is released immediately. If the event is in nonsignaled state, it blocks until some other thread signals the event, which releases all blocked threads. An event can also be set up, so after a signal has been successfully waited for, it automatically reverts back to nonsignaled state, rather than staying in signaled state.

Notes

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging messages. Timers provide a way to block for a specific time interval. Queues are used to notify threads that a previously started asynchronous I/O operation has completed.

Figure 11.4: Some Common Executive Types Object Managed by Object Manager

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
Port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Structure used for mapping files onto virtual address space
Key	Registry key
Object directory	Directory for grouping objects within the object manager
Symbolic link	Pointer to another object by name
Device	I/O device object
Device driver	Each loaded device driver has its own object

Open file objects are created when a file is opened. Files that are not opened do not have objects managed by the object manager. Access tokens are security objects; they identify a user and tell what special privileges the user has, if any. Profiles are structures used for storing periodic samples of the program counter of a running thread to see where the program is spending its time.

Sections are the objects used by the memory system for handling memory-mapped files. They record which file (or part thereof) is mapped onto which memory addresses. Keys are registry keys and are used to relate names to values. Object directories are entirely local to the object manager. They provide a way to collect related objects together in exactly the same way directories work in the file system. Symbolic links are also similar to their file system counterparts—they allow a name in one part of the object name space to refer to an object in a different part of the object name space. Each known device has a device object that contains information about it and is used to refer to the device within the system. Finally, each device driver that has been loaded has an object in the object space.

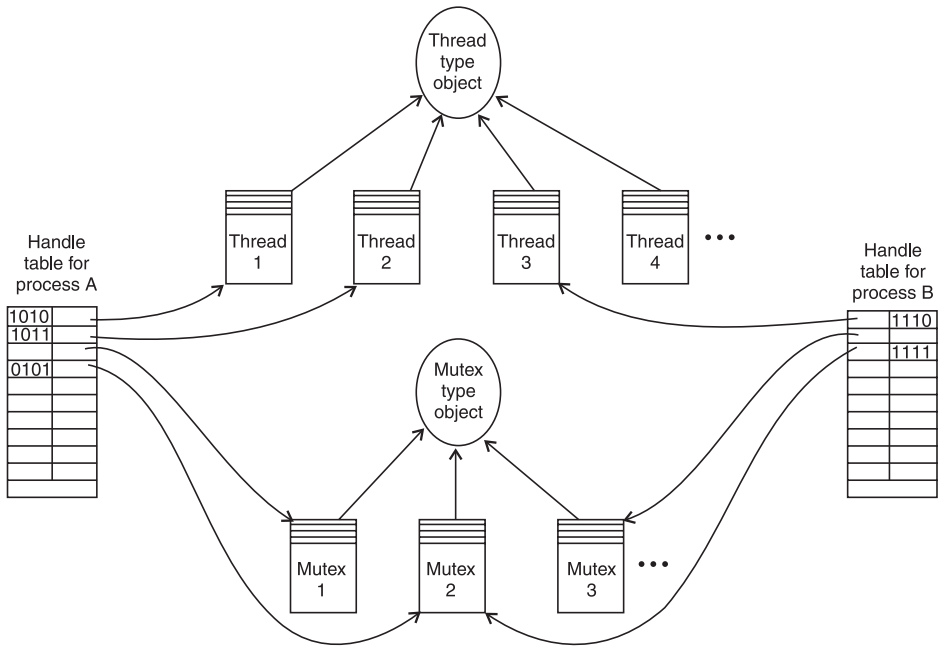
Notes

Users can create new objects or open existing objects by making Win32 calls such as CreateSemaphore or OpenSemaphore. These are the calls to library procedures that ultimately result in the appropriate system calls being made. The result of any successful call that creates or opens an object is a 64-bit handle table entry that is stored in the process' private handle table in kernel memory. The 32-bit index of the handle's position in the table is returned to the user to use on subsequent calls.

The 64-bit handle table entry in the kernel contains two 32-bit words. One word contains a 29-bit pointer to the object's header. The low-order 3 bits are used as flags (e.g., whether the handle is inherited by child processes). These bits are masked off before the pointer is followed. The other word contains a 32-bit rights mask. It is needed because permissions checking is done only at the time the object is created or opened. If a process has only read permission to an object, all the other rights bits in the mask will he 0s, giving the operating system the ability to reject any operation on the object other than reading it.

The handle tables for two processes and their relationships to some objects are illustrated in Figure 11.5. In this example, process A has access to threads 1 and 2 and access to mutexes 1 and 2. Process B has access to thread 3 and mutexes 2 and 3. The corresponding entries in the handle tables hold the rights to each of these objects. For example, process A might have the rights to lock and unlock its mutexes, but not the right to destroy them. Note that mutex 2 is shared by both processes allowing threads in them to synchronize. The other mutexes are not shared, which might mean that the threads within process A use mutex 1 for their internal synchronization and the threads within process and B use mutex 3 for their internal synchronization.

Figure 11.5: The Relationship between Handle Tables, Objects and Type Objects



11.1.6 Object Name Space

As objects are created and deleted during execution, the object manager needs a way to keep track of them. To do this job, it maintains a name space, in which all the objects in the system are located. The name space can be used by a process to locate and open a handle for some other process' object, provided it has been granted permission to do so. The object name space is one of three name spaces maintained by Windows 2000. The other ones are the file system

Notes

name space and the registry name space. All three are hierarchical name spaces with multiple levels of directories for organizing entries. The directory objects listed in Figure 11.4 provide the means to implement this hierarchical name space for objects.

Since executive objects are volatile (i.e., vanish when the computer is shut down, unlike file system and registry entries), when the system boots up, there are no objects in memory and the object name space is empty. During booting, various parts of the executive create directories and then fill them with objects. For example, as the plug-and-play manager discovers devices out there, it creates a device object for each one and enters this object into the name space. When the system is fully booted, all I/O devices, disk partitions, and other interesting discoveries are in the object name space.

Not all objects get entered by the Columbus method—just go look and see what you find. Some executive components look in the registry to see what to do. A key example here is device drivers. During bootup, the system looks in the registry to see which device drivers are needed. As they are loaded one by one, an object is created for each one and its name is inserted into the object space. Within the system, the driver is referred to by a pointer to its object.

Although the object name space is crucial to the entire operation of the system, few people know that it even exists because it is not visible to users without special viewing tools. One such viewing tool is winobj, available for free at www.sysinternals.com. When run, this tool depicts an object name space that typically contains the object directories listed in Figure 11.6 as well as a few others.

Figure 11.6: Some Typical Directories in the Object Name Space

Directory	Contents
??	Starting place for looking up MS-DOS devices like C:
Device	All discovered I/O devices
Driver	Objects corresponding to each loaded device driver
ObjectTypes	The type objects shown in Fig. 11.4
Windows	Objects for sending messages to all the windows
BaseNamedObjs	User-created objects such as semaphores, mutexes, etc.
Arcname	Partition names discovered by the boot loader
NLS	National language support objects
File System	File system driver objects and file system recognizer objects
Security	Objects belonging to the security system
KnownDLLs	Key shared libraries that are opened early and held open

The somewhat strangely named directory \??contains the names of all the MS-DOS-style device names, such as A: for the floppy disk and C: for the first hard disk. These names are actually symbolic finks to the directory \Devicewhere the device objects live. The name \?? was chosen to make it alphabetically first to speed up lookup of all path names beginning with a drive letter. The contents of the other object directories should be self explanatory.

Notes


Did u know?

In computing, a **device driver** or **software** driver is a computer program allowing higher-level computer programs to interact with a hardware device.

11.1.7 Environment Subsystems

Going back to Figure 11.1, we see that Windows 2000 consists of components in kernel mode and in user mode. We have now completed our examination of the kernel mode components; now it is time to look at the user mode components of which there are three kinds: DLLs, environment subsystems, and service processes. These components work together to provide each user process with an interface that is distinct from the Windows 2000 system call interface.

Windows 2000 supports three different documented APIs—Win32, POSIX, and OS/2. Each of these interfaces has a published list of library calls that programmers can use. The job of the DLLs (Dynamic Link Libraries) and environment subsystems is to implement the functionality of the published interface, thereby hiding the true system call interface from application programs. In particular, the Win32 interface is the official interface for Windows 2000, Windows NT, Windows 95/98/Me, and to a limited extent, Windows CE. By using the DLLs and Win32 environment subsystem, a program can be written to the Win32 specification and run unmodified on all these versions of Windows, even though the system calls are not the same on the various systems.

To see how these interfaces are implemented, let us look at Win32. A Win32 program normally contains many calls to Win32 API functions, for example, CreateWindow, DrawMenuBar, and OpenSemaphore. There are thousands of such calls, and most programs use a substantial number of them. One possible implementation would be to statically link every Win32 program with all the library procedures that it uses. If this were done, each binary program would contain one copy of each procedure that used in its executable binary.

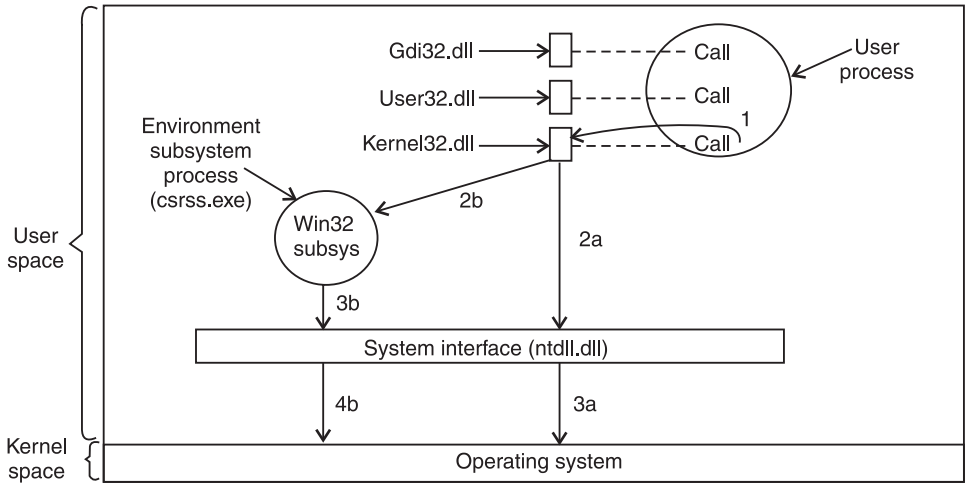
The trouble with this approach is that it wastes memory if the user has multiple programs open at once and they use many of the same library procedures. For example, Word, Excel, and Powerpoint all use exactly the same procedures for opening dialog boxes, drawing windows, displaying menus, managing the clipboard, etc., so if a user had all of them to open and active at once, there would be three (identical) copies of each of the libraries in memory. To avoid this problem, all versions of Windows support shared libraries, called **DLLs (Dynamic Link Libraries)**. Each DLL collects together a set of closely related library procedures and their data structures into a single file, usually (but not always) with extension .DLL. When an application is linked, the linker sees that some of the library procedures belong to DLLs and records this information in the executable's header. Calls to procedures in DLLs are made indirectly through a transfer vector in the caller's address space. Initially this vector is filled with 0s, since the addresses of the procedures to be called are not yet known.

When the application process is started, the DLLs that are needed are located (on disk or in memory) and mapped into the process' virtual address space. The transfer vector is then filled in with the correct addresses so that the procedures can then be called via the transfer vector with only a negligible loss of efficiency. The win here is that even though multiple application programs have the same DLL mapped in, only one copy of the DLL text is needed in physical memory (but each process gets its own copy of the private static data in the DLL). Windows 2000 uses DLLs extremely heavily for all aspects of the system.

Now we have enough background to see how the Win32 and other process interfaces are implemented. Each user process generally links with a number of DLLs that together implement the Win32 interface. To make an API call, one of the procedures in a DLL is called, shown as step 1 in Figure 11.7. What happens next depends on the Win32 API call? Different ones are implemented in different ways.

Notes

Figure 11.7: Various Routes Taken to Implement Win32 API Function Calls



In some cases, the DLL calls another DLL (ntdll.dll) that actually traps to the operating system. This path is shown as steps 2a and 3a in Figure 11.7. It is also possible that the DLL does all the work itself without making a system call at all. For other Win32 API calls, a different route is taken, namely, first a message is sent to the Win32 subsystem process (csrss.exe), which then does some work and then makes a system call (steps 2b, 3b, and 4b). Here, too, in some cases the environment subsystem does all the work in user space and just returns immediately. The message passing between the application process and the Win32 subsystem process has been carefully optimized for performance using a special local procedure call mechanism implemented by the executive and shown as LPC in Figure 11.1.

In the first version of Windows NT, virtually all the Win32 API calls took route 2b, 3b, 4b, putting a large chunk of the operating system in user space (e.g., the graphics). However, starting with NT 4.0, most of the codes were put into kernel mode (in the Win32/GDI driver in Figure 11.1) for performance reasons. In Windows 2000, only a small number of Win32 API calls (for example process and thread creation) take the long route. The other ones take the direct route, by passing the Win32 environment subsystem.

The three most important DLLs are shown in Figure 11.7, but they are not the only the ones. There are over 800 separate DLLs in the \winnt\system32 directory to talling 130 MB. To avoid any confusion, the number of DLL files are over 800; the number of API calls contained in them exceeds 13,000. (The 29 million lines of code had to compile into something, after all.) A few of the more important DLLs are listed in Figure 11.8. The number of exported functions (i.e., those visible outside the file) in each one is given, but these tend to change (meaning increase) over time. The number of exported functions in the first public release of ntdll.dll in Windows 2000 is 1179. These are the real system calls. The 1209 calls exported by ntoskrnl.exe are the functions available to device drivers and other code linked with the kernel. The list of exported functions in any .exe or .dll file can be viewed using the depends program in the platform SDK Kit.

Notes

Figure 11.8: Some key Windows 2000 files, the mode they run in, the number of exported function calls, and the main contents of each file. The calls in win32k.sys are not formally exported since win32k.sys is not called directly

File	Mode	Fcns	Contents
hal.dll	Kernel	95	Low-level hardware management, e.g., port I/O
ntoskrnl.exe	Kernel	1209	Windows 2000 operating system (kernel + executive)
win32k.sys	Kernel	-	Many system calls including most of the graphics
ntdll.dll	User	1179	Dispatcher from user mode to kernel mode
csrss.exe	User	0	Win32 environment subsystem process
kernel32.dll	User	823	Most of the core (nongraphics) system calls
gdi32.dll	User	543	Font, text, colour, brush, pen, bitmap, palette, drawing, etc. calls
user32.dll	User	695	Window, icon, menu, cursor, dialog, clipboard, etc. calls
advapi32.dll	User	557	Security, cryptography, registry, management calls

Although the Win32 process interface is the most important one, there are also two other ones—POSIX and OS/2. The POSIX environment provides minimal support for UNIX applications. It supports only the P1003.1 functionality and little else. It does not have threads, windowing, or networking, for example. In practice, porting any real UNIX program to Windows 2000 using this subsystem is close to impossible. It was included only because parts of the U.S. Government require operating systems for government computers to be P1003.1 compliant. This subsystem is not self-contained and uses the Win32 subsystem for the most of its work, but without exporting the full Win32 interface to its user programs (which would have made it usable, at no extra cost to Microsoft).

To allow UNIX users to migrate to Windows 2000, Microsoft has a product called Interix that provides a better degree of UNIX compatibility than the POSIX subsystem.

The OS/2 subsystem is similarly limited in functionality and does not support any graphical applications. In practice, it, too, is completely useless. Thus the original idea of having multiple operating system interfaces implemented by different processes in user space is essentially gone. What is left is a full Win32 implementation in kernel mode and little else.

Self Assessment

Multiple choice questions:

1. One of the goals of Windows 2000 (and Windows NT before it) was to make the operating system ..... .
   (*a*) portable (*b*) machine dependent
   (*c*) reliable (*d*) none of these
2. Microsoft made a serious attempt to hide many of the machine dependencies in a thin layer at the bottom called ..... .
   (*a*) Kernel (*b*) Hardware layer
   (*c*) Hardware Abstraction layer (*d*) Hardware data layer

Notes

3. The ..... provides a framework for managing I/O devices and provides generic I/O services.

   (*a*) I/O manager (*b*) I/O providers

   (*c*) Device manager (*d*) Device provider

4. The ..... keeps the most recently used disk blocks in memory to speed up access to them in the event that they are needed again.

   (*a*) Power manager (*b*) Memory manager

   (*c*) Process manager (*d*) Cache manager

5. Device drivers are not part of the ..... .

   (*a*) Kernel binary (*b*) krnl.exe binary

   (*c*) ntoskr.exe binary (*d*) ntoskrnl.exe binary

11.2 Summary

- Windows2000 consists of two major parts: the operating system itself, which runs in kernel mode, and the environment subsystems, which runs in user mode.
- Hardware Abstraction Layer, the way it manages clocks and timers in a portable way.
- The Kernel Layer, which includes the control objects and dispatcher objects.
- Executive includes the object manager, I/O manager, process manager, memory manager, security manager, cache manager, plug-and-play manager, power manager, configuration manager, and local procedure call manager.
- Device drivers which control one or more I/O devices.

11.3 Keywords

Control Objects: These are the objects that control the system, including primitive process objects, interrupt objects, and two somewhat strange objects called DPC and APC.

DirectX: HAL is reasonably efficient, for multimedia applications it may not be fast enough. For this reason, Microsoft also produced a software package called DirectX. It allows user processes much more direct access to the hardware.

Graphic Device Interface (GDI): It handles image management for the monitor and printers. It provides system calls to allow user programs to write on the monitor and printers in a device-independent way.

HAL (Hardware Abstract Layer): Microsoft made a serious attempt to hide many of the machine dependencies in a thin layer at the bottom that is called HAL (Hardware Abstraction Layer).

System Services: At the top of the executive is a thin layer called system services. Its function is to provide an interface to the executive. It accepts the true Windows 2000 system calls and calls other parts of the executive to have them executed.

Notes

11.4 Review Questions

1. Explain operating system structure with figure.
2. What is HAL? Describe the job of HAL with suitable example.
3. What is the purpose of the kernel in the operating system? How can we differentiate the kernel with HAL?
4. Where is the executive located in the operating system? Write the components of the executive.
5. Define the following terms:
   (*a*) cache manager
   (*b*) plug-and-play manager
   (*c*) GDI

Answers to Self Assessment

1. (*a*) 2. (*c*) 3. (*a*) 4. (*d*) 5. (*d*)

11.5 Further Readings



Books

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes. *Introduction to Operating Design and Implementation*, by Michael Kifer, Scoott A. Smolka.



Online link

wiley.com/coolege.silberschatz

Unit 12: Processes and Threads in Windows

Notes

CONTENTS

Objectives
Introduction
12.1 Processes and Threads in Windows 2000
12.1.1 Fundamental Concepts of Processes and Threads in Windows 2000
12.1.2 Job, Process, Thread and Fiber Management API Calls
12.1.3 Interprocess Communication
12.1.4 Implementation of Processes and Threads
12.1.5 Scheduling
12.1.6 MS-DOS Emulation
12.2 Booting Windows 2000
12.3 Memory Management
12.3.1 Fundamental Concepts of Memory Management
12.3.2 Memory Management System Calls
12.3.3 Implementation of Memory Management
12.3.4 Page Fault Handling
12.3.5 Page Replacement Algorithm
12.3.6 Physical Memory Management
12.4 Summary
12.5 Keywords
12.6 Review Questions
12.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain process and threads in windows
- Explain booting of Windows 2000
- Discuss memory management

Notes

Introduction

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Microsoft Windows supports preemptive multitasking, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

12.1 Processes and Threads in Windows 2000

Windows 2000 has a number of concepts for managing the CPU and grouping resources together. In the following sections we will examine these, discussing some of the relevant Win32 API calls, and show how these concepts are implemented.

12.1.1 Fundamental Concepts of Process and Threads in Windows 2000

Windows 2000 supports traditional processes, which can communicate and synchronize with one another, just as they can be in UNIX. Each process contains at least one thread, which in turn contains at least one fiber (lightweight thread). Furthermore, processes can be collected into jobs for certain resource management purposes. Together, jobs, processes, threads, and fibers provide a very general set of tools for managing parallelism and resources, both on uniprocessors (single-CPU machines) and on multiprocessors (multi CPU machines). A brief summary of these four concepts is given in Figure 12.1.

Figure 12.1: Basic Concepts Used for CPU and Resource Management

Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

Let us examine these concepts from the largest to the smallest. A job in Windows 2000 is a collection of one or more processes that are to be managed as a unit. In particular, there are quotas and resource limits associated with each job stored in the corresponding job object. The quotas include items such as the maximum number of processes (prevents any process from generating an unbounded number of children), the total CPU time available to each process individually and to all the processes combined, and the maximum memory usage, again, per process and total. Jobs can also impose security restrictions on the processes in the job, such as not being able to acquire administrator (superuser) power, even with the proper password.

Processes are more interesting and also more important than jobs. As in UNIX, processes are containers for resources. Every process has a 4 GB address space, with the user occupying the bottom 2 GB (optionally 3 GB on Advanced server and Datacenter server) and the operating system occupying the rest. Thus the operating system is present in every process' address,

Notes

although protected from tampering by the memory management unit hardware. A process has a process ID, one or more threads, a list of handles (managed in kernel mode), and an access token holding its security information. Processes are created using a Win32 call that takes as its input the name of an executable file, which defines the initial contents of the address space and creates the first thread.

Every process starts out with one thread, but new ones can be created dynamically. Threads forms the basis of CPU scheduling as the operating system always selects a thread to run, not a process. Consequently, every thread has a state (ready, running, blocked, etc.), whereas processes do not have states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space it is to start running at. Every thread has a thread ID, which is taken from the same space as the process IDs, so an ID can never be in use for both: a process and a thread at the same time. Process and thread IDs are multiples of four so they can be used as byte indices into kernel tables, the same as other objects.

A thread normally runs in user mode, but when it makes a system call it switches to kernel mode and continues to run as the same thread with the same properties and limits it had in user mode. Each thread has two stacks, one for use when it is in user mode and one for use when it is in kernel mode. In addition to a state, an ID, and two stacks, every thread has a context (in which to save its registers when it is not running), a private area for its own local variables, and possibly its own access token. If it has its own access token, this one overrides the process access token in order to let client threads pass their access rights to server threads who are doing work for them. When a thread is finished executing, it can exit. When the last thread still active in a process exit, the process terminates.

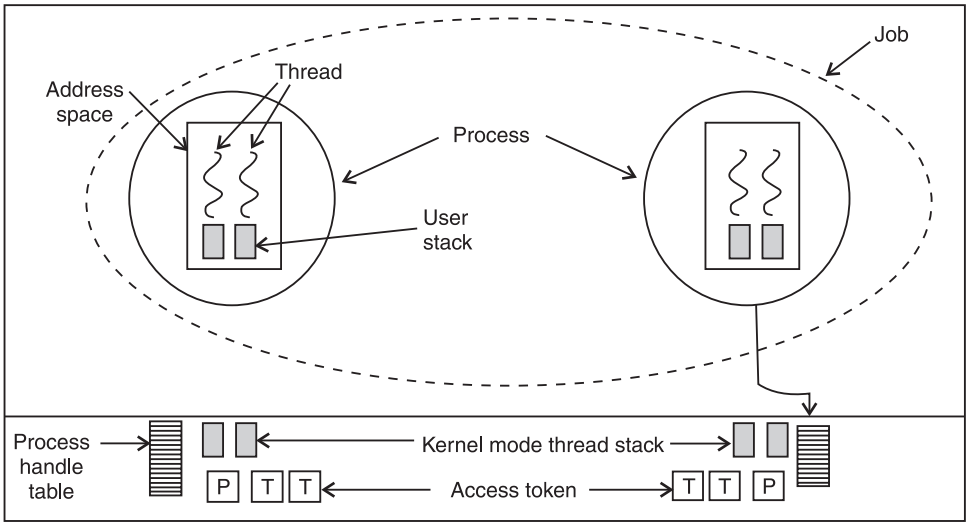
It is important to realize that threads are a scheduling concept, not a resource ownership concept. Any thread is able to access all the objects that belong to its process. All it has to do is to grab the handle and to make the appropriate Win32 call. There is no restriction on a thread that it cannot access an object because a different thread created or opened it. The system does not even keep track of which thread created which object. Once an object handle has been put in a process handle table, any thread in the process can use it.

In addition to the normal threads that run within user processes, Windows 2000 has a number of daemon threads that run only in kernel space and are not associated with any user process (they are associated with the special system or idle processes). Some perform administrative tasks, such as writing dirty pages to the disk, while others form a pool that can be assigned to a component of the executive or a driver that needs to get some work done asynchronously in the background. We will study some of these threads later when we come to the memory management.

Switching threads in Windows 2000 is relatively expensive because doing a thread switch requires entering and later leaving kernel mode. To provide very lightweight pseudoparallelism, Windows 2000 provides **fibers**, which are like threads, but are scheduled in user space by the program that created them (or its run-time system). Each thread can have multiple fibers, the same way a process can have multiple threads, except that when a fiber logically blocks, it puts itself on the queue of blocked fibers and selects another fiber to run in the context of its thread. The operating system is not aware of this transition because the thread keeps on running, even though it may be first running one fiber, then another. In fact, the operating system knows nothing at all about fibers, so there are no executive objects relating to fibers, as there are for jobs, processes, and threads. There are also no true system calls for managing fibers. However, there are Win32 API calls. These are among the Win32 API calls that do not make system calls, which we mentioned during the discussion of Figure 11.7 in previous Unit. The relationship between jobs, processes, and threads is illustrated in Figure 12.2.

Notes

Figure 12.2: The Relationship Between Jobs, Processes and Threads. Several Fibers can also be Multiplexed on One Thread (not shown)



Although we will not discuss it in much detail, Windows 2000 is capable of running on a symmetric multiprocessor system. This requirement means that the operating system code must be fully reentrant, that is, every procedure must be written in such a way that two or more CPUs may be changing its variables at once, without causing problems. In many cases, this means that code sections have to be protected by spin locks or mutexes to keep additional CPUs at bay until the first one is done (i.e., serialize access to critical regions). The number of CPUs the system can handle is governed by the licensing restrictions. There is no technical reason why Windows Professional cannot run on a 32-node multiprocessor, it is the same binary as Datacenter Server, after all.

The upper limit of 32 CPUs is a hard limit because word-length bitmaps are used to keep track of CPU usage in various ways. For example, one word-length bitmap keeps track of which of the (up to) 32 CPUs are currently idle, and another bitmap is used per process to list the CPUs this process is permitted to run on. The 64-bit version of Windows 2000 should be able to effortlessly support up to 64 CPUs; beyond that requires actually changing the code substantially (to use multiple words for the bitmaps).

12.1.2 Job, Process, Thread and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has 10 parameters, each of which has many options. This design is clearly much more complicated than the UNIX scheme, in which fork has no parameters, and exec has just three: pointers to the name of the file to execute: the (parsed) command line parameter array, and the environment strings. Roughly speaking, the 10 parameters to CreateProcess are as follows:

1. A pointer to the name of the executable file.
2. The command line itself (unparsed).
3. A pointer to a security descriptor for the process.
4. A pointer to a security descriptor for the initial thread.
5. A bit telling whether the new process inherits the creator's handles.
6. Miscellaneous flags (e.g., error mode, priority, debugging, consoles).

Notes

7. A pointer to the environment strings.
8. A pointer to the name of the new process' current working directory.
9. A pointer to a structure describing the initial window on the screen.
10. A pointer to a structure that returns 18 values to the caller.

Windows 2000 does not enforce any kind of parent-child or other hierarchy. All processes are created equally (no processes are created more equal). However, since 1 of the 18 parameters returned to the creating process is a handle to the new process (allowing considerable control over the new process), there is an implicit hierarchy in terms of who has a handle to whom. Although these handles cannot just be passed directly to other processes. There is a way for a process to make a duplicate handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in Windows 2000 is created with a single thread, but a process can create more threads later on. Thread creation is simpler than process creation—CreateThread has only six parameters instead of 10 such as:

1. The optional security descriptor.
2. The initial stack size.
3. The starting address.
4. A user-defined parameter.
5. The initial state of the thread (ready or blocked).
6. The thread's ID.

The kernel does the thread creation, so it is clearly aware of threads (i.e., they are not implemented purely in user space as is the case in some other systems).



Task How to differentiate between Job, Process, Thread and Fiber Management API Calls?

12.1.3 Interprocess Communication

Threads can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as might happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network; regular pipes cannot.

Mailslots are the features of Windows 2000, not present in UNIX. They are similar to pipes in some ways, but not all. For one thing, they are one-way, whereas pipes are two-way. They can also be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one receiver.

Notes

Sockets are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote machine reads from it. Sockets can also be used to connect processes on the same machine, but since they entail more overhead than pipes, they are generally only used in a networking context.

Remote procedure calls are a way for process A to have process B call a procedure in B's address space on A's behalf and return the result to A. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process.

Finally, processes can share memory by mapping onto the same file at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in producer-consumer problems can easily be implemented.

Just as Windows 2000 provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms work on threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore is created using the CreateSemaphore API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed on to another process so that multiple processes can be synchronized on the same semaphore. Calls for up and down are present, although they have the somewhat peculiar names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races).

Mutexes are also kernel objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking (WaitForSingleObject) and unlocking (ReleaseMutex). Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

The third synchronization mechanism is based on **critical sections**, (which we have called critical regions elsewhere in this site) which are similar to mutexes, except local to the address space of the creating thread. Because critical sections are not kernel objects, they do not have handles or security descriptors and cannot be passed between processes. Locking and unlocking is done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and only make kernel calls when blocking is needed, they are faster than mutexes.

The last synchronization mechanism uses kernel objects called **events** of which there are two kinds—**manual-reset events** and **auto-reset events**. Any event can be in one of two states; set and cleared. A thread can wait for an event to occur with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a manual-reset event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With an auto-reset event, if one or more threads are waiting, exactly one thread is released and the event is cleared. An alternative operation is PulseEvent, which is like SetEvent except that if nobody is waiting, the pulse is lost and the event is cleared. In contrast, a SetEvent that occurs with no waiting threads is remembered by leaving the event in set state so a subsequent thread waiting on it is released immediately.

Events, mutexes, and semaphores can all be named and stored in the file system, like named pipes. Two or more processes can be synchronized by opening the same event, mutex, or

Notes

semaphore, rather than having one of them create the object and then make duplicate handles for the others, although the latter approach is certainly an option as well.

The number of Win32 API calls dealing with processes, threads, and fibers is nearly 100, a substantial number of which deal with IPC in one form or another. A summary of the ones discussed above as well as some other important ones are given in Figure 12.3.

Figure 12.3: Some of the Win32 Calls for Managing Processes, Threads, and Fibers

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to non-signaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Most of the calls in Figure 12.3 were either discussed above or should be self-explanatory. Again note that not all of these are system calls. As we mentioned earlier, Windows 2000 knows nothing about fibers. They are entirely implemented in user space. As a consequence, the CreateFiber call does its work entirely in user space without making any system calls (unless it has to allocate some memory). Many other Win32 calls have this property as well, including EnterCriticalSection and LeaveCriticalSection as we noted above.

Notes



In computing, 'Inter-process communication' (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes.

12.1.4 Implementation of Processes and Threads

Processes and threads are more important and more elaborate than jobs and fibers, so we will concentrate on them here. A process is created when another process makes the Win32 CreateProcess call. This call invokes a (user-mode) procedure in kernel32.dll that creates the process in following steps using multiple system calls and other work.

1. The executable file given as a parameter is examined and opened. If it is a valid POSIX, OS/2, 16-bit Windows, or MS-DOS file, a special environment is set up for it. If it is a valid 32-bit Win32 .exe file, the registry is checked to see if it is special in some way (e.g., to be run under supervision of a debugger). All of this is done in user mode inside kernel32. dll.

2. A system call, NtCreateProcess, is made to create the empty process object and enter it into the object manager's name space. Both the kernel object and the executive object are created. In addition, the process manager creates a process control block for the object and initializes it with the process ID, quotas, access token, and various other fields. A section object is also created to keep track of the process' address space.

3. When kernel32.dll gets control back, it makes another system call, NtCreateThread, to create the initial thread. The thread's user and kernel stacks are also created. The stack size is given in the header of the executable file.

4. Kernel32.dll now sends a message to the Win32 environment subsystem telling it about the new process and passing it to the process and thread handles. The process and threads are entered into the subsystems tables so it has a complete list of all processes and threads. The subsystem then displays a cursor containing a pointer with an hourglass to tell the user that something is going on but that the cursor can be used in the meanwhile. When the process makes its first GUI call, usually to create a window, the cursor is removed (it times out after 2 seconds if no call is forthcoming).

5. At this point, the thread is able to run. It starts out by running a runtime system procedure to complete the initialization.

6. The run-time procedure sets the thread's priority, tells the loaded DLLs that a new thread is present, and does other housekeeping chores. Finally, it begins running the code of the process' main program.

Thread creation also consists of a number of steps, but we will not go into them in much detail. It starts when the running process executes CreateThread, which calls a procedure inside kernel32.dll. This procedure allocates a user stack within the calling process and then makes the NtCreateThread call to create an executive thread object, initialize it, and also create and initialize a thread control block. Again, the Win32 subsystem is notified and enters the new thread in its tables. Then the thread starts running and completes its own initialization.

When a process or thread is created, a handle is returned for it. This handle can be used to start, stop, kill, and inspect the process or thread. It is possible for the owner of a handle to pass the handle to another process in a controlled and secure way. This technique is used to allow debuggers to have control over the processes they are debugging.

Notes

12.1.5 Scheduling

Windows 2000 does not have a central scheduling thread. Instead, when a thread cannot run any more, the thread enters kernel mode and runs the scheduler itself to see which thread to switch to. The following conditions cause the currently running thread to execute the scheduler code:

1. The thread blocks on a semaphore, mutex, event, I/O, etc.
2. It signals an object (e.g., does an up on a semaphore).
3. The running thread's quantum expires.

In condition 1, the thread is already running in kernel mode to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it must save its own context, run the scheduler code to pick its successor, and load that thread's context to start it.

In condition 2, the running thread is in the kernel, too. However, after signaling some object, it can definitely continue because signaling an object never blocks. Still, the thread is required to run the scheduler to see if the result of its action has released a higher priority thread that is now free to run. If so, a thread switch occurs because Windows 2000 is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum).

In condition 3, a trap to kernel mode occurs, at which time the thread executes the scheduler code to see who runs next. Depending on what other threads are waiting, the same thread may be selected, in which case it gets a new quantum and continues running. Otherwise a thread switch happens.

The scheduler is also called under two other conditions:

1. An I/O operation completes.
2. A timed wait expires.

In the first condition, a thread may have been waiting on this I/O and is now released to run. A check has to be made to see if it should preempt the running thread since there is no guaranteed minimum run time. The scheduler is not run in the interrupt handler itself (since that may keep interrupts turned off too long). Instead a DPC is queued for slightly later, after the interrupt handler is done. In the second condition, a thread has done a down on a semaphore or blocked on some other object, but with a timeout that has now expired. Again it is necessary for the interrupt handler to queue a DPC to avoid having it run during the clock interrupt handler. If a thread has been made ready by this timeout, the scheduler will be run and if nothing more important is available, the DPC will run next.

Now we come to the actual scheduling algorithm. The Win32 API provides two hooks for processes to influence thread scheduling. These hooks largely determine the algorithm. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are: realtime, high, above normal, normal, below normal, and idle. Second, there is a call SetThreadPriority that sets the relative priority of some thread (possibly, but not necessarily, the calling thread) compared to the other threads in its process. The allowed values are: time critical, highest, above normal, normal, below normal, lowest, and idle. With six process classes and seven thread classes, a thread can have any one of 42 combinations. This is the input to the scheduling algorithm.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The 42 combinations are mapped onto the 32 priority classes according to the table of Figure 12.4. The number in the table determines the thread's **base priority**. In addition, every thread has a current priority, which may be higher (but not lower) than the base priority and that we will discuss shortly.

Notes

Figure 12.4: Mapping of Win32 Priorities to Windows 2000 Priorities

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win 32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

To use these priorities for scheduling, the system maintains an array with 32 entries, corresponding to priorities 0 through 31 derived from the table of Figure 12.4. Each array entry points to the head of a list of ready threads at the corresponding priority. The basic scheduling algorithm consists of searching the array from priority 31 down to priority 0. As soon as a non-empty slot is found, the thread at the head of the queue is selected and run for one quantum. If the quantum expires, the thread goes to the end of the queue at its priority level and the thread at the front is chosen next. In other words, when there are multiple threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the idle thread is run.

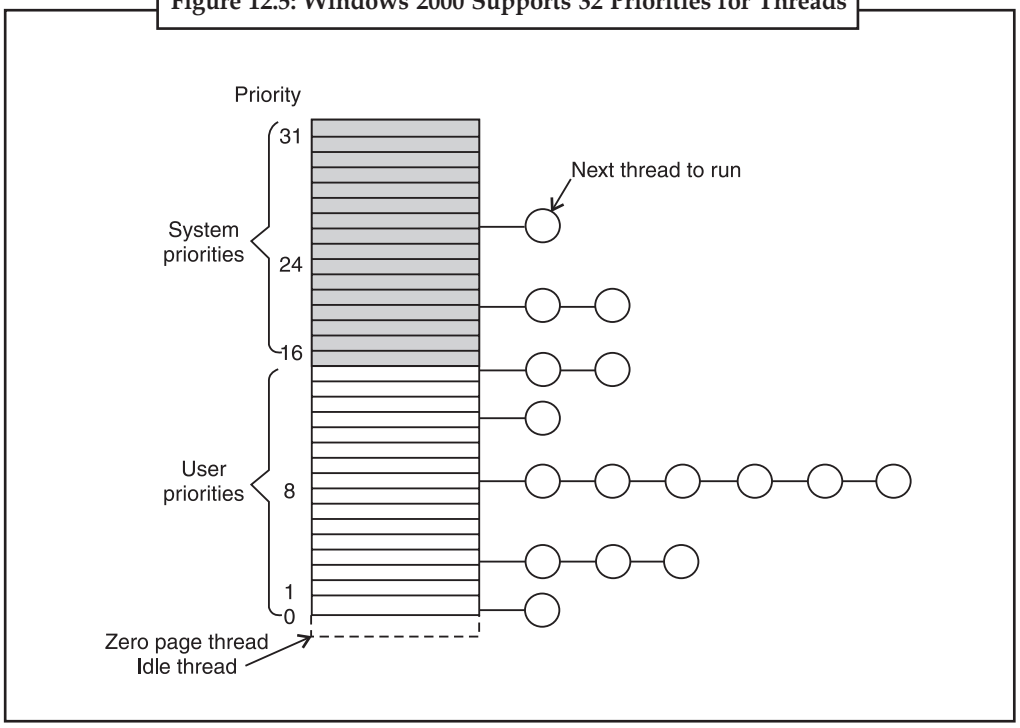
It should be noted that scheduling is done by picking a thread without regard to which process that thread belongs. Thus, the scheduler does not first pick a process and then pick a thread in that process. It only looks at the threads. It does not even know which thread belongs to which process. On a multiprocessor, each CPU schedules itself using the priority array. A spin lock is used to make sure that only one CPU at a time is inspecting the array.

The array of queue headers is shown in Figure 12.5. The figure shows that there are actually four categories of priorities: realtime, user, zero, and idle, which is effectively -1. These deserve some comment. Priorities 16-31 are called real time, but they are not. There are no guarantees given and no deadlines are met. They are simply higher priority than 0-15. However, priorities 16 through 31 are reserved for the system itself and for threads explicitly assigned those priorities by the system administrator. Ordinary users may not run there for a good reason. If a user thread were to run at a higher priority than, say, the keyboard or mouse thread and get into a loop, the keyboard or mouse thread would never run, effectively hanging the system.

User threads run at priorities 1-15. By setting the process and thread priorities, a user can determine which threads get preference. The zero thread runs in the background and eats up whatever CPU time nobody else wants. Its job is to zero pages for the memory manager. We will discuss its role later. If there is absolutely nothing to do, not even zero pages, the idle thread runs. It is not really a full blown thread though.

Notes

Figure 12.5: Windows 2000 Supports 32 Priorities for Threads



Over the course of time, some patches were made to the basic scheduling algorithm to improve system performance. Under certain specific conditions, the current priority of a user thread can be raised above the base priority (by the operating system), but never above priority 15. Since the array of Figure 12.5 is based on the current priority, changing this priority affects scheduling. No adjustments are ever made to threads running at priority 15 or higher.

Let us now see when a thread's priority is raised. First, when an I/O operation completes and releases a waiting thread, the priority is boosted to give it a chance to run again quickly and start more I/O. The idea here is to keep the I/O devices busy. The amount of boost depends on the I/O device, typically 1 for a disk, 2 for a serial line, 6 for the keyboard, and 8 for the sound card.

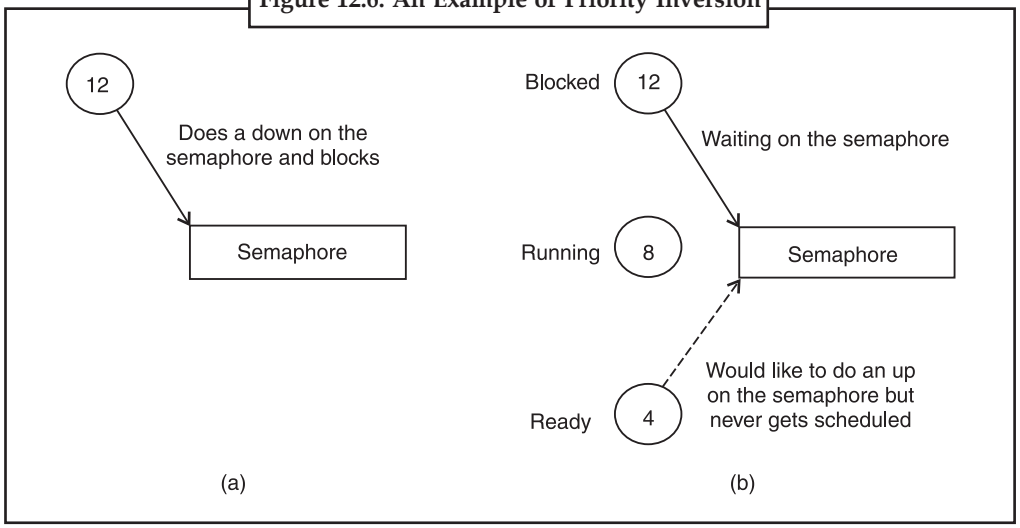
Second, if a thread was waiting on a semaphore, mutex, or other event, when it is released, it gets boosted by 2 units if it is in the foreground process (the process controlling the window to which keyboard input is sent) and 1 unit otherwise. This fix tends to raise interactive processes above the big crowd at level 8. Finally, if a GUI thread wakes up because window input is now available, it gets a boost for the same reason.

These boosts are not forever. They take effect immediately, but if a thread uses all of its next quantum, it loses one point and moves down one queue in the priority array. If it uses up another full quantum, it moves down to another level, and so on until it hits its base level, where it remains until it is boosted again. Clearly, if a thread wants good service, it should play a lot of music.

There is an other case in which the system fiddles with the priorities. Imagine that two threads are working together on a producer-consumer type problem. The producer's work is harder, so it gets a high priority, say 12, compared to the consumer's 4. At a certain point, the producer has filled up a shared buffer and blocks on a semaphore, as illustrated in Figure 12.6 (*a*).

Notes

Figure 12.6: An Example of Priority Inversion



Before the consumer gets a chance to run again, an unrelated thread at priority 8 becomes ready and starts running, as shown in Figure 12.6(*b*). As long as this thread wants to run, it will be able to, since it outruns the consumer and the producer, while higher, is blocked. Under these circumstances, the producer will never get to run again until the priority 8 thread gives up.

Windows 2000 solves this problem through what might be charitably called a big hack. The system keeps track of how long it has been since a ready thread ran last. If it exceeds a certain threshold, it is moved to priority 15 for two quanta. This may give it the opportunity to unblock the producer. After the two quanta are up, the boost is abruptly removed rather than decaying gradually. Probably a better solution would be to penalize threads that use up their quantum over and over by lowering their priority. After all, the problem was not caused by the starved thread, but by the greedy thread. This problem is well known under the name **priority inversion**.

An analogous problem happens if a priority 16 thread grabs a mutex and does not get a chance to run for a long time, starving more important system threads that are waiting for the mutex. This problem can be prevented within the operating system by having a thread that needs a mutex for a short time just disable scheduling while it is busy. On a multiprocessor, a spin lock should be used.

Before leaving the subject of scheduling, it is worth saying a couple of words about the quantum. On Windows 2000 Professional the default is 20 msec; on uniprocessor servers it is 120 msec; on multiprocessors various other values are used, depending on the clock frequency. The shorter quantum favors interactive users whereas the longer quantum reduces context switches and thus provides better efficiency. These defaults can be increased manually by 2x, 4x, or 6x if desired. As an aside, the size of the quantum was chosen a decade ago and not changed since although machines are now more than an order of magnitude faster. The numbers probably could be reduced by a factor of 5 to 10 with no harm and possibly better response time for interactive threads in a heavily loaded system.

One last patch to the scheduling algorithm says that when a new window becomes the foreground window, all of its threads get a longer quantum by an amount taken from the registry. This change gives them more CPU time, which usually translates to better service for the window that just moved to the foreground.


Did u know?

Scheduling is applied in procurement and production, in transportation and distribution, and in information processing and communication.

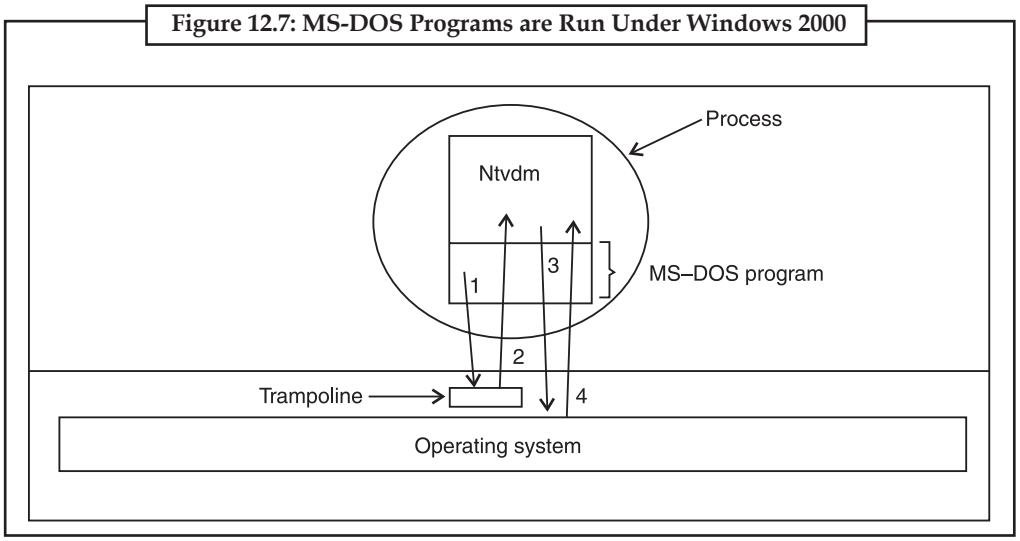
Notes

12.1.6 MS-DOS Emulation

One of the design goals of Windows 2000 was inherited from NT—try to run as many reasonable MS-DOS programs as possible. This goal is quite different from Windows 98's stated goal: run all old MS-DOS programs (to which we add—no matter how ill-behaved they may be).

The way Windows 2000 deals with ancient programs is to run them in a fully protected environment. When an MS-DOS program is started, a normal Win32 process is started and loaded with an MS-DOS emulation program, ntvdm(NT Virtual DOS Machine) that will monitor the MS-DOS program and carry out its system calls. Since MS-DOS only recognized memory up to 1 MB on the 8088 and only up to 16 MB with bank switching and other tricks on the 286, it is safe to put ntvdm high in the process' virtual address space where the program has no way to address it. This situation is shown in Figure 12.7.

Figure 12.7: MS-DOS Programs are Run Under Windows 2000



When the MS-DOS program is just executing normal instructions, it can run on the bare hardware since the Pentium includes all the 8088 and 286 instructions as subsets. The interesting part is what happens when the MS-DOS program wants to do I/O or interact with the system. A well-behaved program just makes a system call. In expectation of this, *ntvdm* instructs Windows 2000 to reflect all MS-DOS system calls back to it. In effect, the system call just bounces off the operating system and is caught by the emulator, as shown in steps 1 and 2 in Figure 12.7. Sometimes this technique is referred to as using a **trampoline**.

Once it gets control, the emulator figures out what the program was trying to do and issues its own Win32 calls to get the work done (step 3 and 4 in Figure 12.7). As long as the program is well behaved and just makes legal MS-DOS system calls, this technique works fine. The trouble is that some old MS-DOS programs bypassed the operating system and wrote directly to the video RAM, read directly from the keyboard, and so on, things that are impossible in a protected environment. To the extent that the illegal behavior causes a trap, there is some hope that the emulator can figure out what the program was trying to do and emulate it. If it does not know what the program wants, the program is just killed because 100 percent emulation was not a Windows 2000 requirement.

Self Assessment

Multiple choice questions:

1. What command is used to remove files?
   (*a*) dm (*b*) m
   (*c*) delete (*d*) erase

Notes

2. Each process contains at least ..... .
   (*a*) one thread (*b*) two threads
   (*c*) four threads (*d*) none of these.
3. A thread normally runs in ..... .
   (*a*) User mode (*b*) Safe mode.
   (*c*) both (*a*) and (*b*) (*d*) None of these
4. A semaphore is created using the ..... API function.
   (*a*) APISemaphore (*b*) CreateSemaphore
   (*c*) WaitSemaphore (*d*) None of these

12.2 Booting Windows 2000

Before Windows 2000 can start up, it must be booted. The boot process creates the initial processes that bring up the system. In this section, we will briefly discuss how the boot process works for Windows 2000. This short assembly language program reads the partition table to see which partition contains the bootable operating system. When it finds the operating system partition, it reads in the first sector of that partition, called the **boot sector**, and jumps to it. The program in the boot sector reads its partition's root directory, searching for a file called ntldr (another piece of archaeological evidence that Windows 2000 is really NT). If it finds that file, it reads the file into memory and executes it. Ntldr loads Windows 2000. As an aside, there are several versions of the boot sector, depending on whether the partition is formatted as FAT-16, FAT-32, or NTFS. When Windows 2000 is installed, the correct version of the master boot record and boot sector are written to disk.

Ntldr now reads a file called Boot.ini, which is the only configuration information not in the registry. It lists all the versions of hal.dll and ntoskrnl.exe available for booting in this partition. The file also provides many parameters, such as how many CPUs and how much RAM to use, whether to give user processes 2 GB or 3 GB, and what rate to set the real-time clock to. Ntldr then selects and loads hal.dll and ntoskrnl.exe files as well as bootvid.dll, the default video driver for writing on the display during the boot process. Ntldr next reads the registry to find out which drivers are needed to complete the boot (e.g., the keyboard and mouse drivers, but also dozens more for controlling various chips on the parentboard). Finally, it reads in all these drivers and passes control to ntoskrnl.exe.

Once started, the operating system does some general initialization and then calls the executive components to do their own initialization. For example, the object manager prepares its name space to allow other components call it to insert their objects into the name space. Many components also do specific things related to their function, such as the memory manager setting up the initial page tables and the plug-and-play manager finding out which I/O devices are present and loading their drivers. All in all, dozens of steps are involved, during which time the progress bar displayed on the screen is growing in length as steps are completed. The last step is creating the first true user process, the **session manager**, smss.exe. Once this process is started and running, booting is completed.

The session manager is a native Windows 2000 process. It makes true system calls and does not use the Win32 environment subsystem, which is not even running yet. In fact, one of its first duties is to start it (csrss.exe). It also reads the registry hives from disk and learns what else it is supposed to do. Typically its work includes entering many objects in the object manager's name space, creating any extra paging files needed, and opening important DLLs to have them around all the time. After it has done most of this work, it creates the login daemon, winlogon.exe.

At this point, the operating system is up and running. Now it is time to get the service processes (user space daemons) going on and allow users to log in. Winlogon.exe first creates the authentication manager (lsass.exe), and then the parent process of all the services (services.exe). The latter looks in the registry to find out which user space daemon processes are needed and what files they are in. It then starts creating them. The fact that the disk is generally being heavily

Notes

used after the first user has logged in (but has done nothing) is not the user's fault. The culprit is services.exe creating all the services. In addition, it also loads any remaining device drivers that have not yet been loaded. The hierarchy of initial processes and some typical services are shown in Figure 12.8.

Figure 12.8: The processes starting up during the boot phase. The ones above the line are always started. The ones below it are examples of services that could be started

	Process	Description
idle		Not really a process, but home to the idle thread
	system	Creates smss.exe & paging files; reads registry; opens DLLs
	smss.exe	First real proc; much initialization; creates csrss & winlogon
	csrss.exe	Win32 subsystem process
	winlogon.exe	Login daemon
	lsass.exe	Authentication manager
	services.exe	Looks in registry and starts services
	Printer server	Allows remote jobs to use the printer
	File server	Serves requests for local files
	Telnet daemon	Allows remote logins
	Incoming email handler	Accepts and stores inbound email
	Incoming fax handler	Accepts and prints inbound faxes
	DNS resolver	Internet domain name system server
	Event logger	Logs various system events
	Plug-and-play manager	Monitors hardware to see what is out there

Winlogon.exe is also responsible for all user logins. The actual login dialog is handled by a separate program in msgina.dll to make it possible for third parties to replace the standard login with faceprint identification or something else other than name and password. After a successful login, winlogon.exe gets the user's profile from the registry and from it determines which shell to run. Many people do not realize it, but the standard Windows desktop is justexplorer.exe with some options set. If desired, a user can select any other program as the shell, including the command prompt or even Word, by editing the registry. However, editing the registry is not for the faint of heart; a mistake here can make the system unusable.

12.3 Memory Management

Windows 2000 has an extremely sophisticated virtual memory system. It has a number of Win32 functions for using it and part of the executive plus six dedicated kernel threads for managing it. In the following sections, we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

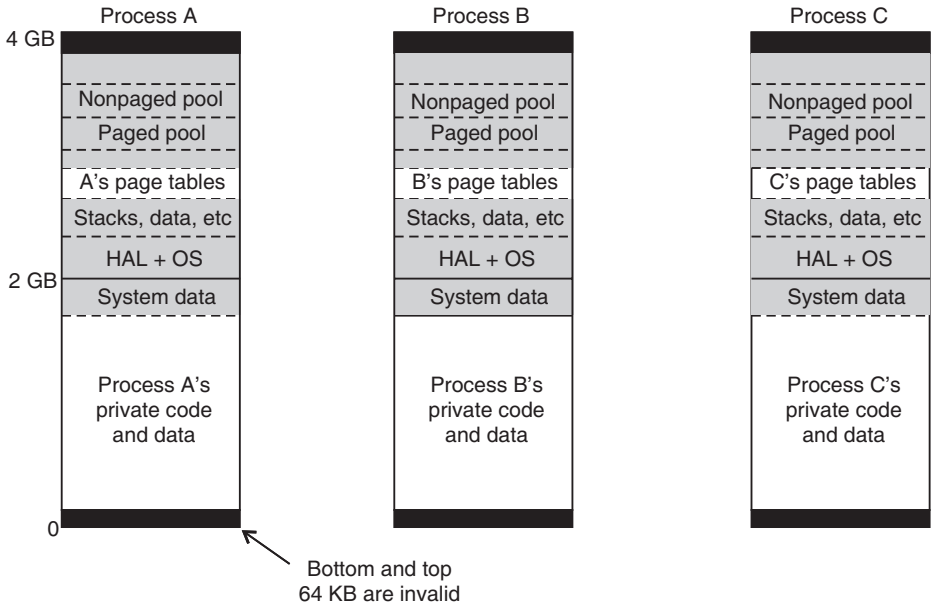
12.3.1 Fundamental Concepts of Memory Management

In Windows 2000, every user process has its own virtual address space. Virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB minus about

Notes

256 MB are available for the process' code and data; the upper 2 GB map onto kernel memory in a protected way. The virtual address space is demand paged, with a fixed page size (4 KB on the Pentium).

Figure 12.9: Virtual address space layout for three user processes. The white areas are private per process. The shaded areas are shared among all processes



The virtual address space layout for three user processes is shown in Figure 12.9 in slightly simplified form. The bottom and top 64 KB of each process' virtual address space is normally unmapped. This choice was made intentionally to help catch programming errors. Invalid pointers are often 0 or -1, so attempts to use them on Windows 2000 will cause an immediate trap instead of reading garbage or, worse yet, writing to an incorrect memory location. However, when old MS-DOS programs are being run in emulation mode, they can be mapped in.

Starting at 64 KB comes the user's private code and data. This extends up to almost 2 GB. The last piece of the bottom 2 GB contains some system counters and timers that are shared among all users read only. Making them visible here allows processes to access them without the overhead of a system call.

The upper 2 GB contains the operating system, including the code, data, and the paged and nonpaged pools (used for objects, etc.). The upper 2 GB is shared among all user processes, except for the page tables, which are each process' own page tables. The upper 2 GB of memory is not writable and mostly not even readable for user-mode processes. The reason for putting it here is that when a thread makes a system call, it traps into kernel mode and just keeps on running in the same thread. By making the whole operating system and all of its data structures (as well as the whole user process) visible within a thread's address space when it enters kernel mode, there is no need to change the memory map or flush the cache upon kernel entry. All that has to be done is switch over to the thread's kernel stack. The trade-off here is less private address space per process in return for faster system calls. Large database servers already feel cramped, which is why the 3-GB user space option is available on Advanced Server and Datacenter Server.

Each virtual page can be in one of three states: free, reserved, or committed. A **free page** is not currently in use and a reference to it causes a page fault. When a process is started, all of its pages are in free state until the program and initial data are mapped into its address space.

Notes

Once code or data is mapped onto a page, the page is said to be **committed**. A reference to a committed page is mapped using the virtual memory hardware and succeeds if the page is in main memory. If the page is not in main memory, a page fault occurs and the operating system finds and brings in the page from disk.

A virtual page can also be in **reserved** state, meaning it is not available for being mapped until the reservation is explicitly removed. For example, when a new thread is created, 1 MB of stack space is reserved in the process virtual address space, but only one page is committed. This technique means that the stack can eventually grow to 1 MB without fear that some other thread will allocate the needed contiguous piece of virtual address space out from under it. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable.

An interesting trade-off occurs with assignment of backing store to committed pages. A simple strategy would be to assign a page in one of the paging files to back up each committed page at the time the page was committed. This would guarantee that there was always a known place to write out each committed page should it be necessary to evict it from memory. The downside of this strategy is that the paging file might have to be as large as the union of all processes' virtual memory. On a large system that rarely ran out of memory and thus rarely paged, this approach would waste disk space.

To avoid wasting disk space, Windows 2000 committed pages that have no natural home on the disk (e.g., stack pages) are not assigned a disk page until the moment that they have to be paged out. This design makes the system more complex because the paging files maps may have to be fetched during a page fault, and fetching them may cause one or more additional page faults inside the page fault handler. On the other hand, no disk space need be allocated for pages that are never paged out.

Trade-offs like this (system complexity versus better performance or more features) tend to get resolved in favour of the latter because the value of better performance or more features is clear but the downside of complexity (a bigger maintenance headache and more crashes per year) is hard to quantify. Free and reserved pages never have shadow pages on disk and references to them always cause page faults.

The shadow pages on the disk are arranged into one or more paging files. There may be up to 16 paging files, possibly spread over 16 separate disks, for higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed. These files can be created at the maximum size at system installation time in order to reduce the chances that they are highly fragmented, but new ones can be created using the control panel later on. The operating system keeps track of which virtual page maps onto which part of which paging file. For (execute only) program text, the executable binary file (i.e., *.exe* or *.dll* file) contains the shadow pages; for data pages, the paging files are used.

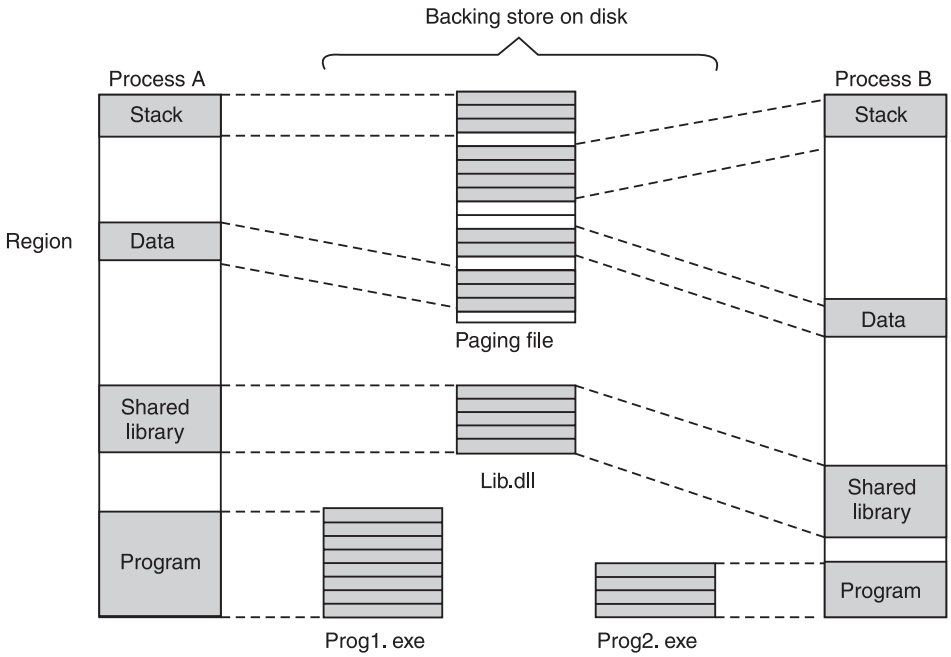
Windows 2000, like many versions of UNIX, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of consecutive pages). Once a file has been mapped onto the address space, it can be read or written using ordinary memory references. Memory-mapped files are implemented in the same way its other committed pages, only the shadow pages are in the users file instead of in the paging file. As a result, while a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the virtual address space). However, when the file is unmapped or is explicitly flushed, the disk version is brought up-to-date.

Windows 2000 explicitly allows two or more processes to map onto the same part of the same file at the same time, possibly at different virtual addresses, as shown in Figure 12.10. By reading and writing memory words, the processes can now communicate with each other and pass data back and forth at very high bandwidth, since no copying is required. Different processes may have different access permissions. Since all the processes using a mapped file share the same pages, changes made by one of them are immediately visible to all the others, even if the disk

Notes

file has not yet been updated. Care is also taken that if another process opens the file for normal reading, it sees the current pages in RAM, not stale pages from the disk.

Figure 12.10: Mapped Regions with their Shadow Pages on Disk. The *lib.dll* File is Mapped into Two Address Spaces at the Same Time.



It is worth noting that there is a problem if two programs share a DLL file and one of them changes the file's static data. If no special action is taken, the other one will see the changed data, which is probably not what is desired. The problem is solved by mapping all pages in as read only by secretly noting that some are really writable. When a write happens to a page that is mapped read only but is really writable, a private copy of the page is made and mapped in. Now it can be written safely without affecting other users or the original copy on disk. This technique is called **copy-on-write**.

Also it is worth noting that if program text is mapped into two address spaces at different addresses, a certain problem arises with addressing. What happens if the first instruction is JMP 300? If process one maps the program in at address 65,536, the code can easily be patched to read JMP 65836. But what happens if a second process now maps it in at 131,072? The JMP 65836 will go to address 65,836 instead of 131,372 and the program will fail. The solution is to use only relative offsets, not absolute virtual addresses in code that is to be shared. Fortunately, most machines have instructions using relative offsets as well as instructions using absolute addresses. Compilers can use the relative offset instructions, but they have to know in advance whether to use them or the absolute ones. The relative ones are not used all the time because the resulting code is usually less efficient. Usually, a compiler flag tells them which to use. The technique of making it possible to place a piece of code at any virtual address without relocation is called **position independent code**.

Years ago, when 16 bit (or 20 bit) virtual address spaces were standard, but machines had megabytes of physical memory, all kinds of tricks were thought of to allow programs to use more physical memory than to fit in the address space. Often these tricks went under the name of **bank switching,** in which a program could substitute some block of memory above the 16-bit or 20-bit limit for a block of its own memory. When 32-bit machines were introduced, people

Notes

thought they would have enough address space forever. They were wrong. The problem is back. Large programs often need more than the 2 GB or 3 GB of user address space Windows 2000 allocates to them, so bank switching is back, now called **address windowing extensions**. This facility allows programs to map into shuffle chunks of memory in and out of the user portion of the address space (and especially above the dreaded 4 GB boundary). Since it is only used on servers with more than 2 GB of physical memory, we will defer the discussion until the next edition of this site (by which time even entry-level desktop machines will be feeling the 32-bit pinch).

12.3.2 Memory Management System Calls

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Figure 12.11. All of them operate on a region consisting either of a single page or a sequence of two or more pages that are consecutive in the virtual address space.

Figure 12.11: The Principal Win32 API Functions for Managing Virtual Memory in Windows 2000

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

The first four API functions are used to allocate, free, protect and query regions of virtual address space. Allocated regions always begin on 64 KB boundaries to minimize porting problems to future architectures with pages larger than current ones (up to 64 KB). The actual amount of address space allocated can be less than 64 KB, but must be a multiple of the page size. The next two give a process the ability to hardware pages in memory so they will not be paged out and to undo this property. A real-time program might need this ability, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. Actually, the pages can be removed from memory, but only if the entire process is swapped out. When it is brought back, all the locked pages are reloaded before any thread can start running again. Although not shown in Figure 12.11, Windows 2000 also has API functions to allow a process to access the virtual memory of a different process over which it has been given control (i.e., for which it has a handle). The last four API functions listed are for managing memory-mapped files. To map a file, a file mapping object (see Figure 12.4) must first be created, with CreateFileMapping. This function returns a handle to the file mapping object and optionally enters a name for it

Notes

into the file system so another process can use it. The next two functions map and unmap files, respectively. The last one can be used by a process to map in a file currently also mapped in by a different process. In this way, two or more processes can share regions of their address spaces. This technique allows them to write in limited regions of each other's memory.

12.3.3 Implementation of Memory Management

Windows 2000 supports a single linear 4 GB demand-paged address space per process. Segmentation is not supported in any form. Theoretically, page sizes can be of any power of two up to 64 KB. On the Pentium they are fixed at 4 KB; on the Itanium they can be 8 KB or 16 KB. In addition, the operating system itself can use 4 MB pages to reduce page table space consumed.

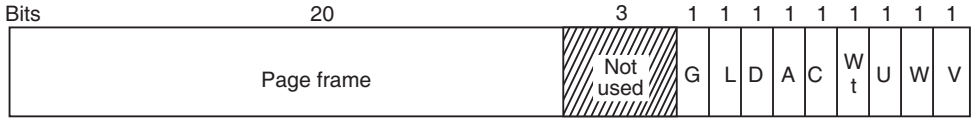
Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager deals with. When a region of virtual address space is allocated, as four of them have been for process *A* in Figure 12.10, the memory manager creates a **VAD (Virtual Address Descriptor)** for it, listing the range of addresses mapped, the backing store file and offset where it is mapped, and the protection code. When the first page is touched, the directory of page tables is created and a pointer to it inserted is in the VAD. In fact, an address space is completely defined by the list of its VADs. This scheme supports sparse address spaces because unused areas between the mapped regions use no resources.

12.3.4 Page Fault Handling

Windows 2000 does not use any form of prepaging. When a process starts, none of its pages are in memory. All of them are brought in dynamically as page faults occurs. On each page fault, a trap to the kernel occurs. The kernel builds a machine-independent descriptor telling what happened and passes this to the memory manager part or the executive. The memory manager then checks it for validity. If the faulted page falls within a committed or reserved region, it looks up the address in the list of VADs, finds (or creates) the page table, and looks up the relevant entry.

The page table entries are different for different architectures. For the Pentium, the entry for a mapped page is shown in Figure 12.12. Unmapped pages also have entries, but their format is somewhat different. For example, for an unmapped page that must be zeroed before it may be used, that fact is noted in the page table.

Figure 12.12: A Page Table Entry for a Mapped Page on the Pentium



G: Page is global to all processes
L : Large (4 MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled

Wt: Write through (no caching)
U : Page is accessible in user mode
W : Writing to the page permitted
V : Valid page table entry

Notes

The most important bits in the page table entry for purposes of the paging algorithm are the A and D bits. They are fed by the hardware and keep track of whether the page has been referenced or written on, respectively, since the last time they were cleared.

Page faults come in five categories:

1. The page referenced is not committed.
2. A protection violation occurred.
3. A shared page has been written.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are fatal errors from which there is no recovery for the faulting process. The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. The solution is to copy the page to a new physical page frame and map the same in read/write. This is how copy-on-write works. (If a shared page is marked writable in all the processes using it, it is not copy-on-write and no fault occurs when writing to it.) The fourth case requires allocating a new page frame and mapping in it. However, the security rules require that the page contain only 0s, to prevent the process from snooping on the previous owner of the page. Thus a page of 0s must be found, or if one is not available, another page frame must be allocated and zeroed on the spot. Finally, the fifth case is a normal page fault where page is located and mapped in.

The actual mechanics of getting and mapping pages is fairly standard, so we will not discuss this issue. The only noteworthy feature is that Windows 2000 does not read in isolated pages from the disk. Instead, it reads in runs of consecutive pages, usually about 1-8 pages, in an attempt to minimize the number of disk transfers. The run size is larger for code pages than for data pages.

12.3.5 Page Replacement Algorithm

Page replacement works like this. The system makes a serious attempt to maintain a substantial number of free pages in the memory so that when a page fault occurs, a free page can be claimed on the spot, without the need to first write some other page to disk. As a consequence of this strategy, most page faults can be satisfied with at most one disk operation (reading in the page), rather than sometimes two (writing back a dirty page and then reading in the needed page).

Of course, the pages on the free list have to come from somewhere, so the real page replacement algorithm is how pages get taken away from processes and put on the free list (actually, there are four free lists, but for the moment it is simplest to think of there being just one; we will come to the details later). Let us now have a look at how Windows 2000 frees pages. To start with, the entire paging system makes heavy use of the working set concept. Each process (not each thread) has a working set. This set consists of the mapped-in pages that are in memory and can be thus referenced without a page fault. The size and composition of the working set fluctuates as the process' threads run, of course.

Each process' working set is described by two parameters: the minimum size and the maximum size. Every process starts with the same minimum and maximum, but these bounds can change over time. The default initial minimum is in the range 20-50 and the default initial maximum is in the range 45-345, depending on the total amount of RAM. The system administrator can change these defaults, however.

Notes

If a page fault occurs and the working set is smaller than the minimum, the page is added. On the other hand, if a page fault occurs and the working set is larger than the maximum, a page is evicted from the working set (but not from memory) to make room for the new page. This algorithm means that Windows 2000 uses a local algorithm, to prevent one process from hurting others by hogging memory. However, the system does try to tune itself to some extent. For example, if it observes that one process is paging like crazy (and the others are not), the system may increase the size of its maximum working set, so that over time, the algorithm is a mix of local and global. There is an absolute limit on the working set size, however: even if there is only one process running, it may not take the last 512 pages, to leave some slack for new processes.

So far, so good, but the story is not over yet. Once a second, a dedicated kernel daemon thread, the **balance set manager**, checks to see if there are enough free pages. If there are not enough, it starts the **working set manager** thread to examine the working sets and recover more pages. The working set manager first determines the order to examine the processes in. Large processes that have been idle for a long time are considered before small active processes and the foreground process is considered last.

The working set manager then starts inspecting processes in the chosen order. If a process' working set is currently less than its minimum or it has incurred more than a certain number of page faults since the last inspection, it is passed over. Otherwise, one or more pages are removed. The target number of pages to remove is a complicated function of the total RAM size, how tight memory is, how the current working set size compares to the process' minimum and maximum, and other parameters. All the pages are examined in turn.

On a uniprocessor, if a page's reference bit is clear, a counter associated with the page is incremented. If the reference bit is set, the counter is set to zero. After the scan, the pages with the highest counters are removed from the working set. The thread continues examining processes until it has recovered enough pages, then it stops. If a complete pass through all processes still has not recovered enough pages, it makes another pass, trimming more aggressively, even reducing working sets below their minimum if necessary.

On a multiprocessor, looking at the reference bit does not work because although the current CPU may not have touched the page recently, some other one may have. Examining another CPU's reference bits is too expensive to do. Consequently, the reference bit is not examined and the oldest pages are removed.

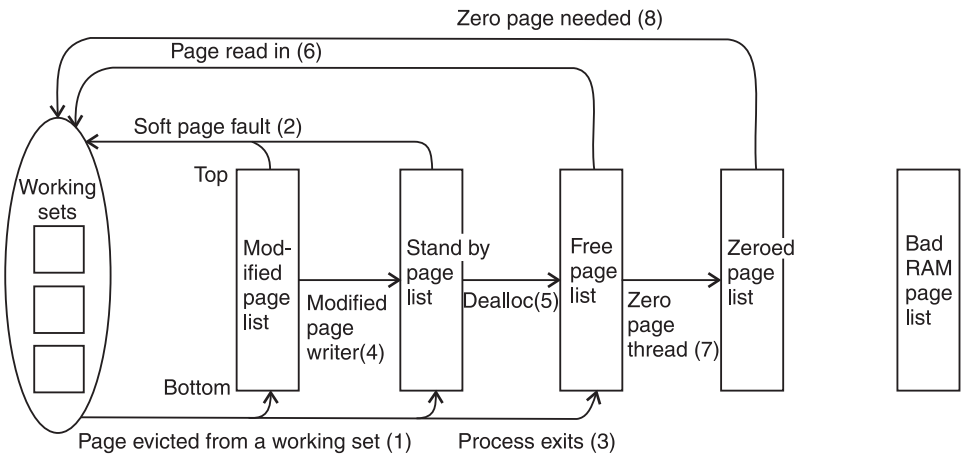
It should be noted that for page replacement purposes, the operating system itself is regarded as a process. It owns pages and also has a working set. This working set can be trimmed. However, parts of the code and the nonpaged pool are locked in memory and cannot be paged out under any circumstances.

12.3.6 Physical Memory Management

Above we mentioned that there were actually four free lists. Now it is time to see what all of them are for. Every page in memory is either in one or more working sets or on exactly one of these four lists, which are illustrated in Figure 12.13. The standby (clean) and modified (dirty) lists hold pages that have recently been evicted from a working set, are still in memory, and are still associated with the process that was using them. The difference between them is that clean pages have a valid copy on disk and can thus be abandoned at will, whereas dirty pages do not have an up-to-date copy on disk. The free list consists of clean pages that are no longer associated with any process. The pages on the zeroed page list are not associated with any process and are also filled with zeros. A fifth list holds any physically defective RAM pages that may exist to make sure that they are not used for anything.

Notes

Figure 12.13: The Various Page Lists and the Transitions between Them



Pages are moved between the working sets and the various lists by the working set manager and other kernel daemon threads. Let us examine the transitions. When the working set manager removes a page from a working set, the page goes on the bottom of the standby or modified list, depending on its state of cleanliness. This transition is shown as (1). Pages on both lists are still valid pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits its nonshared pages cannot be faulted back to it, so they go on the free list (3). These pages are no longer associated with any process.

Other transitions are caused by other daemon threads. Every 4 seconds the **swapper thread** runs and looks for processes all of whose threads have been idle for a certain number of seconds. If it finds any such processes, their kernel stacks are unpinned and their pages are moved to the standby or modified lists, also shown as (1).

Two other daemon threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If there are not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list. The former handles writes to mapped files and the latter handles writes to the paging files. The result of these writes is to transform dirty pages into clean pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block, if there is no room in memory to bring them in when a page has to be written, a deadlock could result. The other thread can solve the problem by writing the pages to a paging file, which never grows. Nobody ever said Windows 2000 was simple.

The other transitions in Figure 12.13 are as follows. If a process unmaps a page, the page is no longer associated with a process and can go on the free list (5), except for the case that it is shared. When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety. The situation is different when a stack grows.

In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel daemon thread, the **zero page thread**, runs at the lowest priority, erasing pages that are on the free list and putting them on the zeroed page list (7).

Notes

Whenever the CPU is idle and there are free pages, they might as well as be zeroed since a zeroed page is potentially more useful than a free page.

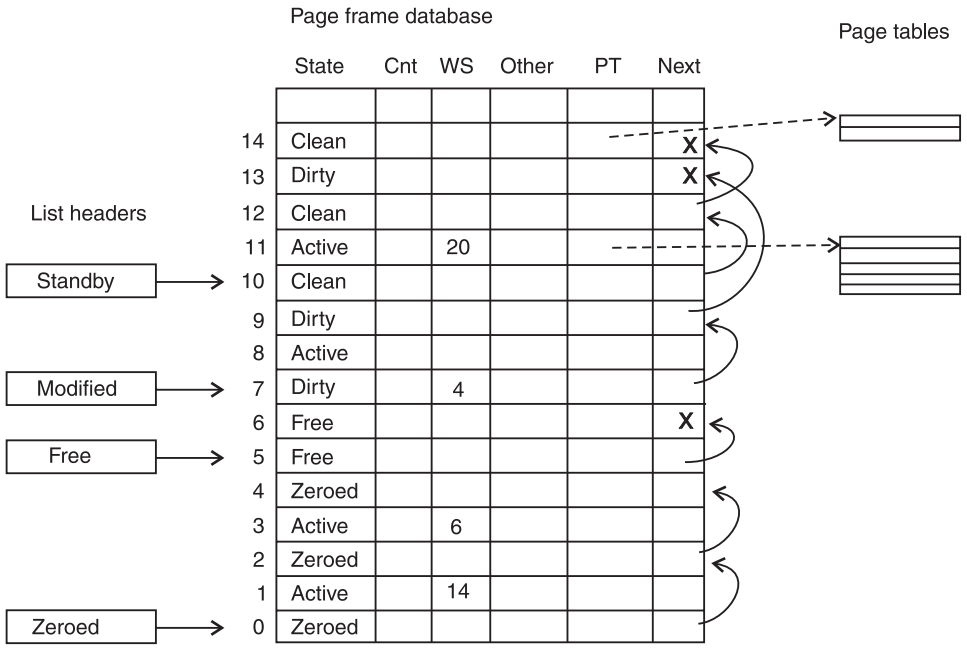
The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better? If the CPU is idle a lot and the zero page thread gets to run often, taking a zeroed page is better because there is no shortage of them. However, if the CPU is always busy and the disk is mostly idle, it is better to take a page from the standby list to avoid the CPU cost of having to zero another page later if a stack grows.

Another puzzle, how aggressively should the daemons move pages from the modified list to the standby list? Having clean pages around is better than having dirty pages around (since they can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly-cleaned page may be faulted back into a working set and dirtied again.

In general, Windows 2000 resolves these kinds of conflicts through complex heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings. Furthermore, the code is so complex that the designers are loathe to touch parts of it for fear of breaking something somewhere else in the system that nobody really understands any more.

To keep track of all the pages and all the lists, Windows maintains a page frame database with as many entries as there are RAM pages, as shown in Figure 12.14. This table is indexed by physical page frame number. The entries are fixed length, but different formats are used for different kinds of entries (e.g., valid versus invalid). Valid entries maintain the page's state and a count of how many page tables point to the page, so the system can tell when the page is no longer in use. Pages that are in a working set tell which one. There is also a pointer to the page table pointing to the page, if any (shared pages are handled specially), a link to the next page on the list (if any), and various other fields and flags, such as read in progress, write in progress, etc.

Figure 12.14: Some of the Major Fields in the Page Frame Database for a Valid Page



Notes

All in all, memory management is a highly complex subsystem with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple paging algorithm like clock or aging.


Did u know?

Memory management is the act of managing computer memory. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed.

Self Assessment

Fill in the blanks:

5. The actual login dialog is handled by a separate program in ..... .
6. In ..... every user process has its own virtual address space.
7. The ..... file is mapped into two address spaces at the same time.
8. Mapped page writer handles writes to ..... files.

True or False:

9. Once order or data is mapped onto a page, the page is said to be a free page.
10. Windows 2000 support a single linear 4 GB demand-page address space per process.

12.4 Summary

- Every process starts with one thread, but newer ones can be created dynamically.
- A thread normally runs in user mode.
- Windows 2000 does not enforce any kind of parent-child or other hierarchy.
- Threads creation is simpler than process creation. Create threads has only six parameters.
- Socket is like pipes, except that they normally connect process on different machines.
- In implementation of process, a process created when another process makes the win32 create process call.
- When an MS-DOS program started, a normal Win32 process is started and loaded with an MS-DOS emulation program.

12.5 Keywords

API: The Windows API, informally WinAPI, is Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems.

Boot Sector: A boot sector is a sector of a hard disk, floppy disk, or similar data storage device that contains code for booting programs (usually, but not necessarily, operating systems) stored in other parts of the disk.

Notes

Mailslots: Mailslot is a mechanism for one-way interprocess communications (IPC). Applications can store messages in a mailslot. The owner of the mailslot can retrieve messages that are stored there. These messages are typically sent over a network to either a specified computer or to all computers in a specified domain.

Mutexes: Mutex object is a synchronization object whose state is set to be signaled when it is not owned by any thread, and not signaled when it is owned.

Scheduling: Scheduling is the process of deciding how to commit resources between a variety of possible tasks.

Win32: The Windows API for developing 32-bit applications. Win32 is built into Windows 95 and Windows NT so applications that rely on the API (Win32 applications) should run equally well in both environments. It is also possible to run some Win32 applications under older 16-bit versions of windows by installing the Win32s runtime system.

12.6 Review Questions

1. Explain the process of threads in windows and discuss its basic fundamental.
2. Draw the diagram of relationship between jobs, process and threads. Explain.
3. Give the description on Inter-process communication.
4. Define the following terms:
   (*a*) Socket (*b*) Mailslots
   (*c*) Semaphore
5. Explain and discuss the scheduling in windows operating system.
6. How to use the in Windows MS-DOS Emulation?
7. Give and explain the booting process of windows system.
8. Give the working of Memory management in operating system.
9. Describe the following terms briefly:
   (*a*) Page fault (*b*) Virtual page
   (*c*) Default page (*d*) Page replacement

Answers to Self Assessment

1. (*b*) 2. (*a*) 3. (*a*) 4. (*b*)
5. msgina.dll 6. Windows 2000 7. lib.dill 8. mapped
9. False 10. True

12.7 Further Readings


Books

Operating Systems, by Harvey M. Deitel , Paul J. Deitel, David R. Choffnes, *Introduction to Operating Design and Implementation*, by Michael Kifer, Scoott A. Smolka.


Online link

wiley.com/coolege.silberschatz

Unit 13: Input/Output and Security of Windows

Notes

CONTENTS

Objectives

Introduction

13.1 Input/Output in Windows 2000

13.1.1 Fundamental Concepts of Input/Output in Windows 2000

13.1.2 Input/Output API Calls

13.1.3 Implementation of I/O

13.1.4 Device Drivers

13.2 The Windows 2000 File System

13.2.1 Fundamental Concepts of Windows 2000 File System

13.2.2 File System API Calls in Windows 2000

13.2.3 Implementation of the Windows 2000 File System

13.2.4 File System Structure

13.2.5 File Name Lookup

13.2.6 File Compression

13.2.7 File Encryption

13.3 Security in Windows 2000

13.3.1 Fundamental Concepts of Security in Windows 2000

13.3.2 Security API Calls

13.3.3 Implementation of Security

13.4 Caching in Windows 2000

13.5 Summary

13.6 Keywords

13.7 Review Questions

13.8 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Discuss input/output of Windows
- Explain Windows 2000 file system
- Explain security in Windows 2000
- Understand caching in Windows 2000

Introduction

A large part of the I/O manager's role is the management of communication between drivers. All drivers supply a standard set of services that the I/O manager can call. This uniform interface allows the I/O manager to communicate with all drivers in the same way, without any knowledge of how the devices they control actually work.

Drivers communicate with each other using data structures called I/O request packets. The drivers do not pass I/O request packets to each other directly. Instead, they pass the packets to the I/O manager, which delivers them to the appropriate destination drivers using the drivers' standard services. The packets passed at the various stages are different—it's a primary job of each layer to construct the appropriate request packets to pass to the next layer.

13.1 Input/Output in Windows 2000

The goal of the Windows 2000 I/O system is to provide a framework for efficiently handling a very wide variety of I/O devices. Current input devices include various kinds of keyboards, mice, touch pads, joysticks, scanners, still cameras, television cameras, bar code readers, microphones, and laboratory rats. Current output devices include monitors, printers, plotters, beamers, CD-recorders, and sound cards. Storage devices include floppy disks, IDE and SCSI hard disks, CD-ROMs, DVDs, Zip drives, and tape drives. Finally, other devices include clocks, networks, telephones, and camcorders. No doubt many new I/O devices will be invented in the years to come, so Windows 2000 has been designed with a general framework to which new devices can easily be attached. In the following sections we will examine some of the issues relating to I/O.

13.1.1 Fundamental Concepts of Input/Output in Windows 2000

The I/O manager is on intimate terms with the plug-and-play manager. The basic idea behind plug and play is that of an enumerable bus. Many buses, including PC Card, PCI, USB, IEEE 1394, and SCSI, have been designed so that the plug-and-play manager can send a request to each slot and ask the device there to identify itself. Having discovered that what is out there, the plug-and-play manager allocates hardware resources, such as interrupt levels, locates the appropriate drivers, and loads them into memory. As each one is loaded, a **driver object** is created for it. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB and IEEE 1394, it can happen at any moment, requiring close contact between the plug-and-play manager, the bus driver (which actually does the enumeration), and the I/O manager.

Notes

The I/O manager is also closely connected with the power manager. The power manager can put the computer into any of six states, roughly described as:

1. Fully operational.
2. Sleep 1: CPU power reduced, RAM and cache on; instant wake-up.
3. Sleep 2: CPU and RAM on; CPU cache off; continue from current PC.
4. Sleep 3: CPU and cache off; RAM on; restart from fixed address.
5. Hibernate: CPU, cache, and RAM off; restart from saved disk file.
6. Off: Everything off; full reboot required.

I/O devices can also be in various power states. Turning them on and off is handled by the power manager and I/O manager together. Note that states 2 through 6 are only used when the CPU has been idle for a shorter or longer time interval.

Somewhat surprisingly, all the file systems are technically I/O drivers. Requests for data blocks from user processes are initially sent to the cache manager. If the cache manager cannot satisfy the request from the cache, it has the I/O manager call the proper file system driver to go and get the block it needs from disk.

An interesting feature of Windows 2000 is its support for **dynamic disks**. These disks may span multiple partitions and even multiple disks and may be reconfigured on the fly, without even having to reboot. In this way, logical volumes are no longer constrained to a single partition or even a single disk so that a single file system may span multiple drives in a transparent way.

Another interesting aspect of Windows 2000 is its support for asynchronous I/O. It is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is especially important on servers. There are various ways the thread can find out that the I/O has completed. One is to specify an event object at the time the call is made and then wait on it eventually. Another is to specify a queue to which a completion event will be posted by the system when the I/O is done. A third is to provide a callback procedure that the system calls when the I/O has completed.

13.1.2 Input/Output API Calls

Windows 2000 has over 100 separate APIs for a wide variety of I/O devices, including mice, sound cards, telephones, tape drives, etc. Probably the most important is the graphics system, for which there are thousands of Win32 API calls. We began our discussion with the Window graphical system. Here we will continue, mentioning a few of the Win32 API categories, each of which has many calls. A brief summary of the categories are given in Figure 13.1.

Win32 calls exist to create, destroy, and manage windows. Windows have a vast number of styles and options that can be specified, including titles, borders, colours, sizes, and scroll bars. Windows can be fixed or movable, of constant size or resizable. Their properties can be queried and messages can be sent to them.

Many windows contain menus, so there are Win32 calls for creating and deleting menus and menu bars. Dynamic menus can be popped up and removed. Menu items can be highlighted, dimmed out, or cascaded.

Notes

Dialog boxes are popped up to inform the user of some event or ask a question. They may contain buttons, sliders, or text fields to be filled in. Sounds can also be associated with dialog boxes, for example for warning messages.

There are hundreds of drawing and painting functions available, ranging from setting a single pixel to doing complex region clipping operations. Many calls are provided for drawing lines and closed geometric figures of various kinds, with detailed control over textures, colours, widths, and many other attributes.

Figure 13.1: Some Categories of Win32 API Calls

API group	Description
Window management	Create, destroy, and manage windows
Menus	Create, destroy, and append to menus and menu bars
Dialog boxes	Pop up a dialog box and collect information
Painting and drawing	Display points, lines, and geometric figures
Text	Display text in some font, size, and colour
Bitmaps and icons	Placement of bitmaps and icons on the screen
Colours and palettes	Manage the set of colours available
The clipboard	Pass information from one application to another
Input	Get information from the mouse and keyboard

Another group of calls relates to displaying text. Actually, the text display call, TextOut, is straightforward. It is the management of the colour, point sizes, typefaces, character widths, glyphs, kerning, and other typesetting details where the complexity comes in. Fortunately, the rasterization of text (conversion to bitmaps) is generally done automatically.

Bitmaps are small rectangular blocks of pixels that can be placed on the screen using the BitBlt Win32 call. They are used for icons and occasionally text. Various calls are provided for creating, destroying, and managing icon objects.

Many displays use a colour mode with only 256 or 65,536 of the 2^{24} possible colours in order to represent each pixel with only 1 or 2 bytes, respectively. In these cases a colour palette is needed to determine which 256 or 65,536 colours are available. The calls in this group create, destroy, and manage palettes, select the nearest available colour to a given colour, and try to make colours on the screen match colours on colour printers.

Many Windows 2000 programs allow the user to select some data (e.g., a block of text, part of a drawing, a set of cells in a spreadsheet), put it on the clipboard, and allow it to be pasted into another application. The clipboard is generally used for this transmission. Many clipboard formats

Notes

are defined, including text, bitmaps, objects, and metafiles. The latter are sets of Win32 calls that when executed draw something, allowing arbitrary drawings to be cut and pasted. This group of calls puts things on the clipboard, takes things off the clipboard, and generally manages it.

Finally, we come to input. There are no Win32 calls for GUI applications for reading input from the keyboard because GUI applications are event driven. The main program consists of a big loop getting input messages. When the user types something interesting, a message is sent to the program telling it what just came in. On the other hand, there are calls relating to the mouse such as reading its (x, y) position and the state of its buttons. Some of the input calls are actually output calls, though, such as selecting a mouse cursor icon and moving it around the screen (basically, this is output to the screen). For nonGUI applications, it is possible to read from the keyboard.

13.1.3 Implementation of I/O

We could go on more or less indefinitely about the Win32 graphics calls, but now it is time to look at how the I/O manager implements graphics and other I/O functions. The main function of the I/O manager is to create a framework in which different I/O devices can operate. The basic structure of the framework is a set of device-independent procedures for certain aspects of I/O plus a set of loaded device drivers for communicating with the devices.

13.1.4 Device Drivers

To make sure that device drivers work well with the rest of Windows 2000, Microsoft has defined a **Windows Driver Model** that device drivers are expected to conform with. Furthermore, it also has provided a tool kit that is designed to help driver writers produce conformant drivers. In this section, we will briefly examine this model. Conformant drivers must meet all of the following requirements as well as some others:

1. Handle incoming I/O requests, which arrive in a standard format.
2. Be as object based as the rest of Windows 2000.
3. Allow plug-and-play devices to be dynamically added or removed.
4. Permit power management, where applicable.
5. Be configurable in terms of resource usage.
6. Be reentrant for use on multiprocessors.
7. Be portable across Windows 98 and Windows 2000.

I/O Requests are passed to drivers in the form of a standardized packet called an **IRP (I/O Request Packet)**. Conformant drivers must be able to handle them. Drivers must be object based in the sense of supporting a specific list of methods that the rest of the system can call. They must also correctly deal with other Windows 2000 objects when given an object handle to deal with.

Conformant drivers must fully support plug and play, which means that if a device managed by the driver is suddenly added or removed from the system, the driver must be prepared to accept this information and act on it, even in the case that the device currently being accessed is suddenly removed. Power management must also be supported for the devices for which this is relevant. For example, if the system decides it is now time to go into a low-power hibernation mode, all devices that are capable of doing this must do so to save energy. They must also wake up when told to do so.

Notes

Drivers must be configurable, which means not having any built-in assumptions about which interrupt lines or I/O ports certain devices use. For example, the printer port on the IBM PC and its successors has been at address 0x378 for more than 20 years and it is unlikely to change now. But a printer driver that has this address hard coded into it is not conformant.

Being multiprocessor safe is also a requirement as Windows 2000 was designed for use on multiprocessors. Concretely, this requirement means while a driver is actively running and processing one request on behalf of one CPU, a second request may come in on behalf of a different CPU. The second CPU may begin executing the driver code simultaneously with the first one. The driver must function correctly even when being executed concurrently by two or more CPUs, which implies that all sensitive data structures may only be accessed from inside critical regions. Just assuming that there will not be any other calls until the current one is finished is not permitted.

Finally, conformant drivers must work not only on Windows 2000 but also on Windows 98. It may be necessary to recompile the driver on each system however, and use of C preprocessor commands to isolate platform dependencies is permitted.

In UNIX, drivers are located by using their major device numbers. Windows 2000 uses a different scheme. At boot time, or when a new hot pluggable plug-and-play device is attached to the computer, Windows 2000 automatically detects it and calls the plug-and-play manager. The manager queries the device to find out what the manufacturer and model number are. Equipped with this knowledge, it looks on the hard disk in a certain directory to see if it has the driver. If it does not, it displays a dialog box asking the user to insert a floppy disk or CD-ROM with the driver. Once the driver is located, it is loaded into memory.

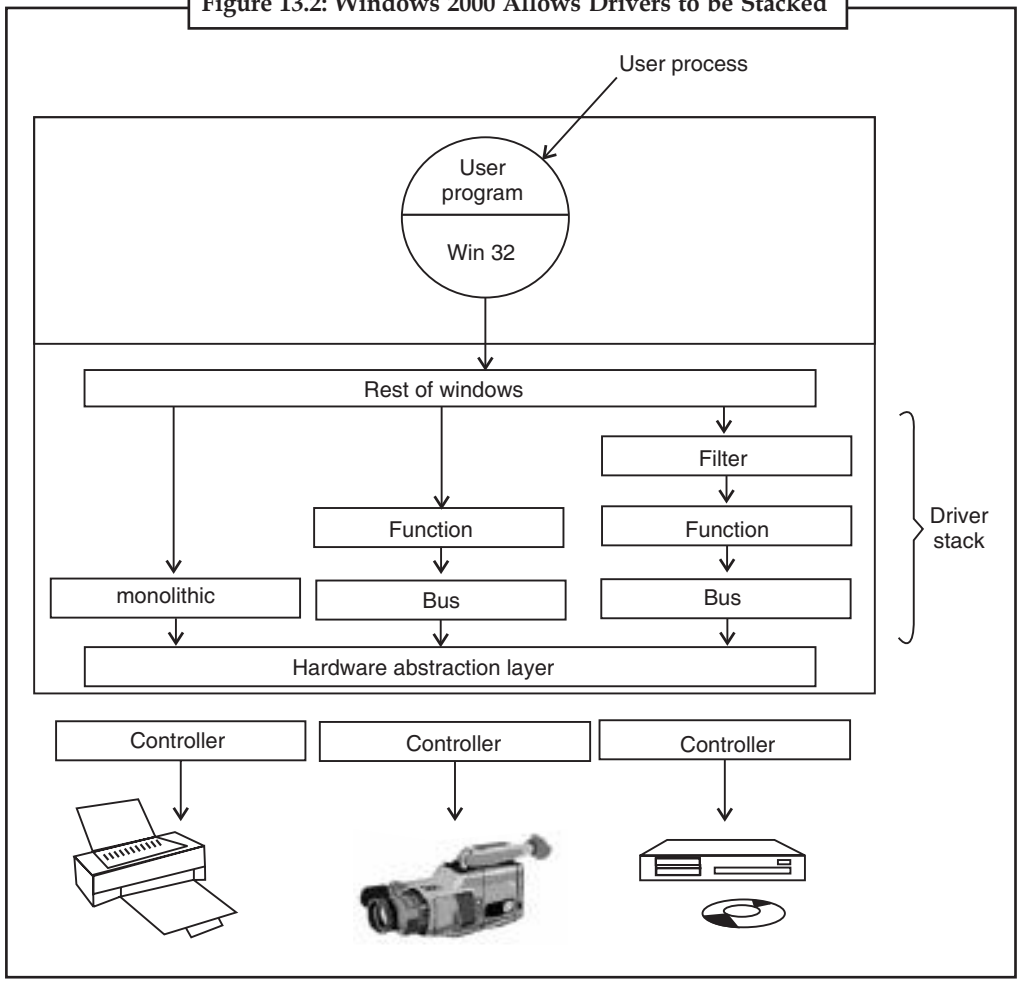
Each driver must supply a set of procedures that can be called to get its services. The first one, called *DriverEntry*, initializes the driver. It is called just after the driver is loaded. It may create tables and data structures, but must not touch the device yet. It also fills in some of the fields of the driver object created by the I/O manager when the driver was loaded. The fields in the driver object include pointers to all the other procedures that drivers must supply. In addition, for each device controlled by the driver (e.g., each IDE disk controlled by the IDE disk driver), a **device object** is created and initialized to point to the driver object. These driver objects are entered into a special directory,\??. Given a device object, the driver object can be located easily, and hence its methods can be called.

A second required procedure is *AddDevice*, which is called once (by the plug-and-play manager) for each device to be added. Once this has been accomplished, the driver is called with the first IRP, which sets up the interrupt vector and actually initializes the hardware. Other procedures that drivers must contain are the interrupt service procedure, various timer management procedures, a fast I/O path, DMA control, a way to cancel currently executing requests, and many more. All in all, Windows 2000 drivers are so complex that multiple sites have been written about them.

A driver in Windows 2000 may do all the work by itself, as the printer driver does in Figure 13.2 (just as an example). On the other hand, drivers may also be stacked, which means that a request may pass through a sequence of drivers, each doing part of the work. Two stacked drivers are also illustrated in Figure 13.2.

Notes

Figure 13.2: Windows 2000 Allows Drivers to be Stacked



One common use for stacked drivers is to separate out the bus management from the functional work of actually controlling the device. Bus management on the PCI bus is quite complicated on account of many kinds of modes and bus transactions, and by separating this work from the device-specific part, driver writers are freed from learning how to control the bus. They can just use the standard bus driver in their stack. Similarly, USB and SCSI drivers have a device-specific part and a generic part, with common drivers used for the generic part.

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. A filter driver performs some transformation on the data on the way up or down. For example, a filter driver could compress data on the way to the disk or encrypt data on the way to the network. Putting the filter here means that neither the application program nor the true device driver have to be aware of it and it works automatically for all data going to (or coming from) the device.

13.2 The Windows 2000 File System

Windows 2000 supports several file systems, the most important of which are **FAT-16**, **FAT-32** and **NTFS (NT File System)**. FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. NTFS is a new file system developed specifically for Windows NT and carried over to Windows 2000. It uses 64-bit disk addresses and can (theoretically) support disk partitions up to 264 bytes, although other considerations limit it to smaller sizes. Windows 2000 also supports

Notes

read-only file systems for CD-ROMs and DVDs. It is possible (even common) to have the same running system have access to multiple file system types available at the same time.

In this unit, we will treat the NTFS file system because it is a modern file system unencumbered by the need to be fully compatible with the MS-DOS file system, which was based on the CP/M file system designed for 8-inch floppy disks more than 20 years ago. Times have changed and 8-inch floppy disks are not quite state of the art any more. Neither are their file systems. Also, NTFS differs both in user interface and implementation in a number of ways from the UNIX file system, which makes it a good second example to study. NTFS is a large and complex system and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

13.2.1 Fundamental Concepts of Windows 2000 File System

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, τλ∈ is a perfectly legal file name. NTFS fully supports case sensitive names (so foo is different from Foo and FOO). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs restricted to using Win32 (e.g., for Windows 98 compatibility).

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each of which is represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in foo:stream1. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file was borrowed from the Apple Macintosh, in which files have two streams, the data fork and the resource fork. This concept was incorporated into NTFS to allow an NTFS server be able to serve Macintosh clients.

File streams can be used for purposes other than Macintosh compatibility. For example, a photo editing program could use the unnamed stream for the main image and a named stream for a small thumbnail version. This scheme is simpler than the traditional way of putting them in the same file one after another. Another use of streams is in word processing. These programs often make two versions of a document, a temporary one for use during editing and a final one when the user is done. By making the temporary one a named stream and the final one the unnamed stream, both versions automatically share a file name, security information, timestamps, etc. with no extra work.

The maximum stream length is 2^{64} bytes. To get some idea of how big a 2^{64}-byte stream is, imagine that the stream were written out in binary, with each of the 0s and 1s in each byte occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back. File pointers are used to keep track of where a process is in each stream, and these are 64 bits wide to handle the maximum length stream, which is about 18.4 exabytes.

The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. For graphical applications, no file handles are predefined. Standard input, standard output, and standard error have to be acquired explicitly if needed; in console mode they are preopened, however, Win32 also has a number of additional calls not present in UNIX.

Notes

13.2.2 File System API Calls in Windows 2000

The principal Win32 API functions for file management are listed in Figure 13.3. There are actually many more, but these give a reasonable first impression of the basic ones. Let us now examine these calls briefly. CreateFile can be used to create a new file and return a handle to it. This API function must also be used to open existing files as there is no FileOpen API function. We have not listed the parameters for the API functions because they are so voluminous. As an example, CreateFile has seven parameters, which are roughly summarized as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.
7. The handle of a file whose attributes should be cloned for the new file.

Figure 13.3: The Principal Win32 API Functions for File I/O. The Second Column Gives the Nearest UNIX Equivalent

Win32 API function	UNIX	Description
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

The next six API functions in Figure 13.3 are fairly similar to the corresponding UNIX system calls. The last two allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version. Such a code fragment (without any error checking) is shown in the program below. It has been designed to mimic our UNIX version. In practice, one would not have to program a copy file program since CopyFile is an API function (which executes something close to this program as a library procedure).

Notes

Program

```
/* Open files for input and output */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
/* Copy the file. */
do{
   s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
   if(s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while(s > 0 && count > 0);
 /* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Windows 2000 NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is \ however, instead of /, a fossil inherited from MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. Hard and symbolic links are supported, the former implemented by having multiple directory entries, as in UNIX, and the latter implemented using reparse points (discussed later in this unit). In addition, compression, encryption, and fault tolerance are also supported. These features and their implementations will be discussed later in this unit.

The major directory management API functions are given in Figure 13.4, again along with their nearest UNIX equivalents. The functions should be self explanatory.

Figure 13.4: The Principal Win32 API Functions for Directory Management. The Second Column Gives the Nearest UNIX Equivalent, When One Exists

Win32 API function	UNIX	Description
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile	rename	Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

13.2.3 Implementation of the Windows 2000 File System

NTFS is a highly complex and sophisticated file system. It was designed from scratch, rather than being an attempt to improve the old MS-DOS file system. Below we will examine a number of its features, starting with its structure, then moving on to file name lookup, file compression, and file encryption.

13.2.4 File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's

Notes

terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4 KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64 bit numbers.

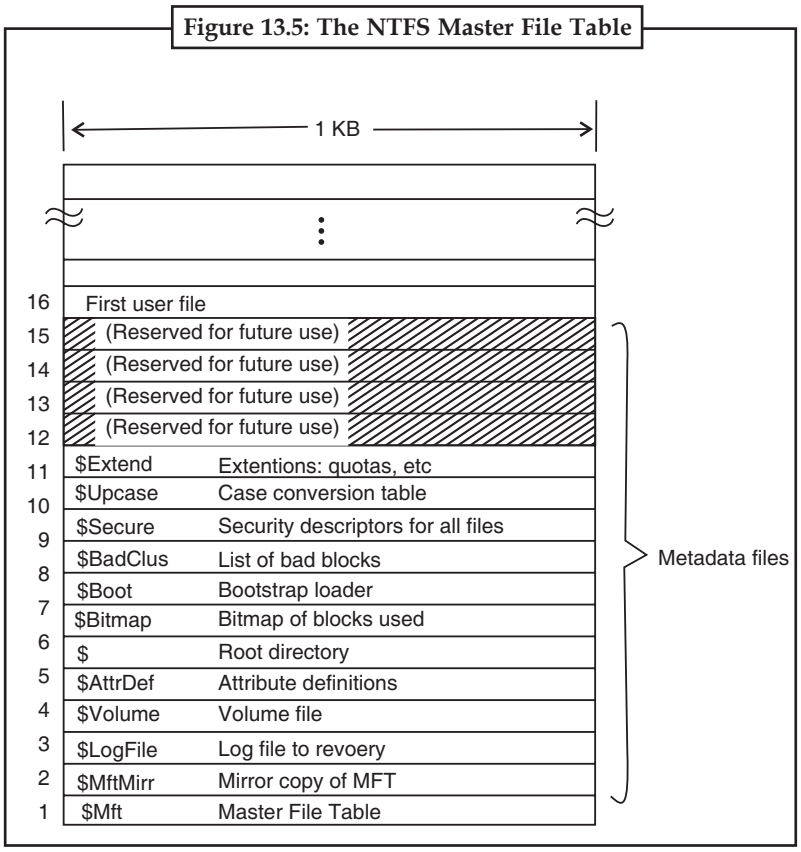
The main data structure in each volume is the **MFT (Master File Table)**, which is a linear sequence of fixed-size 1 KB records. Each MFT record describes one file or directory. It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the other MFT records. This overflow scheme dates back to CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Figure. 13.5. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is because some attribute values are variable length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record.

The first 16 MFT records are reserved for NTFS metadata files, as shown in Figure 13.5. Each of the records describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so the system can find the MFT file. Clearly, Windows 2000 needs a way to find the first block of the MFT file in order to find the rest of the file system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed at system installation time.

Figure 13.5: The NTFS Master File Table



Notes

Record 1 is a duplicate of the early part of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever goes bad. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation. Changes to file attributes are also logged here. In fact, the only changes not logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The $AttrDef file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last 4 MFT records are reserved for future use.

Each MFT record consists of a record header followed by a sequence of (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields. Following the record header comes the header of the first attribute, then the first attribute value, the second attribute header, the second attribute value, and so on.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Figure 13.6. Each MFT record consists of a sequence of attribute headers, each of which identifies the attribute it is heading and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in a separate disk block. Such an attribute is said to be a **non-resident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for non-resident attributes are longer because they contain information about where to find the attribute on disk.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and is always present. The file name is variable length in Unicode. In order to make files with nonMS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS name. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not used.

Notes

Figure 13.6: The Attributes Used in MFT Records

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

In NT 4.0, security information could be put in an attribute, but in Windows 2000 it all goes into a single file so that multiple files can share the same security descriptions. The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

The object ID attribute gives the file a unique name. This is sometimes needed internally. The reparse point tells the procedure parsing the file name to do something special. This mechanism is used for mounting and symbolic links. The two volume attributes are used only for volume identification. The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that everyone has been waiting for the data. The stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the file contained, or for files of only a few hundred bytes (and there are many of these), the file itself. Putting the actual file data in the MFT record is called an **immediate file**.

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually non-resident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular data.

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a file is placed in block 20 on the disk, then the system will try hard to place the second logical block in block 21,

Notes

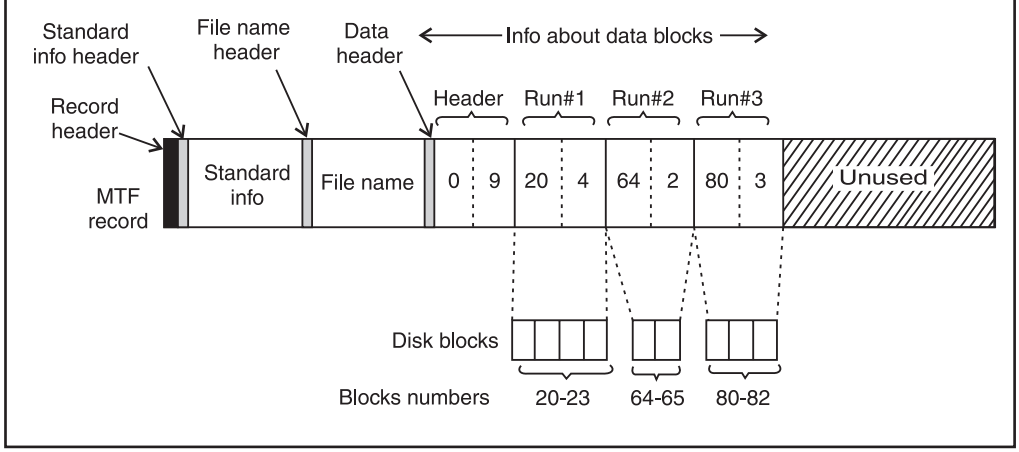
the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, if possible.

The blocks in a file are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a file with no holes in it, there will be only one such record. Files that are written in order from beginning to end all belong to this category. For a file with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a file could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros.

Each record begins with a header giving the offset of the first block within the file. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60, 80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block file is illustrated in Figure. 13.7.

Figure 13.7: An MFT Record for a Three-run, Nine-block File



In this figure we have an MFT record for a short file (short here means that all the information about the file blocks fits in one MFT record). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20-23, the second is blocks 64-65, and the third is blocks 80-82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs are there depends on how good a job the disk block allocator did in finding runs of consecutive blocks when the file was created. For a *n*-block file, the number of runs can be anything from 1 up to and including *n*.

Several comments are worth making here. First, there is no upper limit to the size of files that can be represented this way. In the absence of address compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20 MB file consisting of 20 separate runs of 1 million 1 KB blocks each fits easily in one MFT record, whereas a 60 KB file scattered into 60 isolated blocks does not.

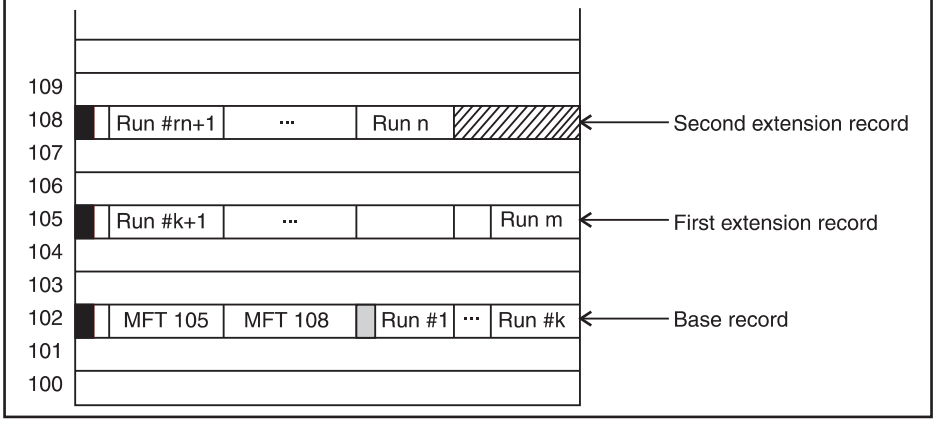
Second, while the straightforward way of representing each pair takes 2 × 8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple

Notes

high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy—all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple—use two or more MFT records. In Figure 13.8 we see a file whose base record is in MFT record 102. It has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first k data runs.

Figure 13.8: A File that Requires Three MFT Records to Store All Its Runs



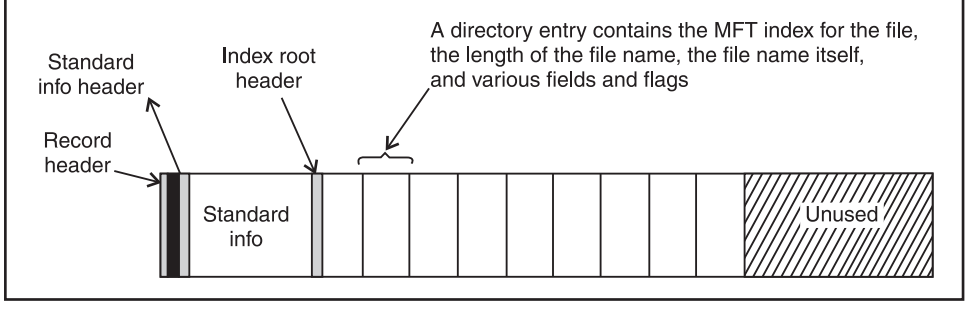
Note that Figure 13.8 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for "overspecifying" this information is to make seeking more efficient: to find the block at a given file offset, it is only necessary to examine the record headers, not the run pairs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way many MFT records can be used to handle large fragmented files.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the list of extension MFT records is made nonresident (i.e., stored on disk instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Figure 13.9. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

Figure 13.9: The MFT Record for a Small Directory



Notes

Large directories use a different format. Instead of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

13.2.5 File Name Lookup

We now have enough information to see how file name lookup occurs. When a user program wants to open a file, it typically makes a call like CreateFile("C:\maria\web.htm", …)

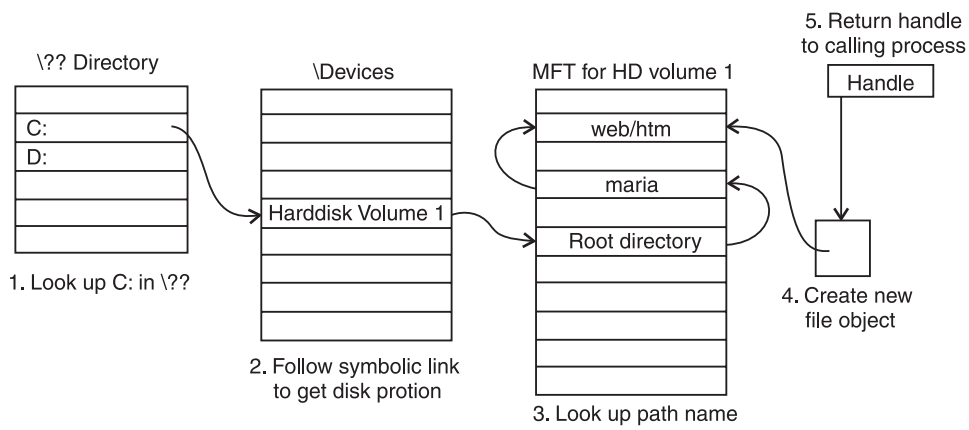
This call goes to the user-level shared library, *kernel32.dll* where \?? is pre-pended to the file name giving

\??\C:\maria\web.htm

It is the name that is passed as a parameter to the system call NtFileCreate.

Then the operating system starts the search at the root of the object manager's name space. It then looks in the directory \?? to find C:, which it will find. This file is a symbolic link to another part of the object manager's name space, the directory \Device. The link typically ends at an object whose name is something like \Device\HarddiskVolume1. This object corresponds to the first partition of the first hard disk. From this object it is possible to determine which MFT to use, namely the one on this partition. These steps are shown in Figure 13.10.

Figure 13.10: Steps in Looking up the File *C:\maria\web.htm*



The parsing of the file name continues now at the root directory, whose blocks can be found from entry 5 in the MFT. The string "maria" is now looked up in the root directory, which returns the index into the MFT for the directory *maria*. This directory is then searched for the string "web.htm". If successful, the result is a new object created by the object manager. The object, which is unnamed, contains the index of the MFT record for the file. A handle to this object is returned to the calling process. On subsequent ReadFile calls, the handle is provided, which allows the object manager to find the index and then the contents of the MFT record for the file. If a thread in a second process opens the file again, it gets a handle to a new file object.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. It is possible to tag a file or directory as a reparse point and associate a block of data with it. When the file or directory is encountered during a file name parse, exception processing is triggered and the block of data is interpreted. It can do various things, including redirecting the search to a different part of the directory hierarchy or even to a different partition. This mechanism is used to support both symbolic links and mounted file systems.

Notes

13.2.6 File Compression

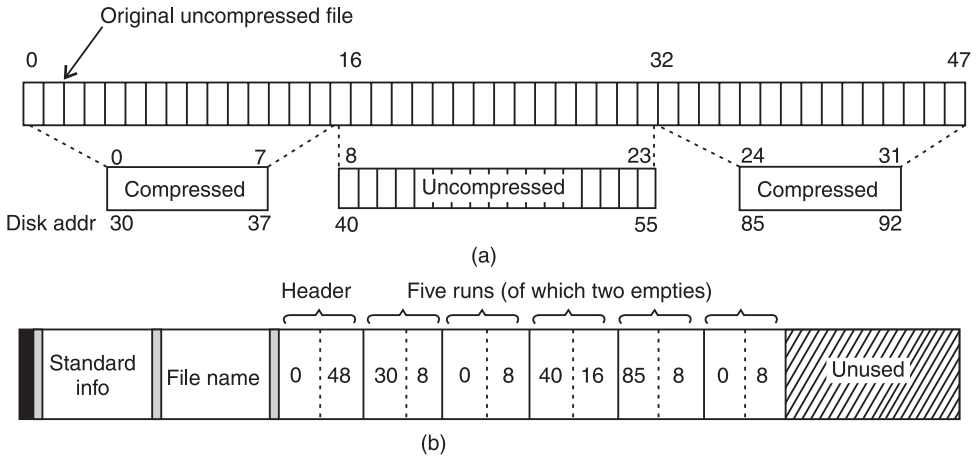
NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware of the fact that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting data can be stored in 15 or fewer blocks, the compressed data are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16-31 are examined to see if they can be compressed to 15 blocks or less, and so on.

Figure 13.11 shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The "missing" blocks are stored in the MFT entry with disk address 0 as shown in Figure 13.11. Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

When the file is read back, NTFS has to know which runs are compressed and which are not. It sees that based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since it contains the boot sector, using it for data is impossible anyway.

Figure 13.11: (*a*) An Example of a 48-Blocks File being Compressed to 32 Blocks (*b*) The MFT Record for the File after Compression



Random access to compressed files is possible, but tricky. Suppose that a process does a seek to block 35 in Figure 13.11. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive.

Notes

13.2.7 File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, etc. where the owners do not especially want revealed to anyone. Information loss can happen when a laptop computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows 2000 security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system. Even the simple act of going to the bathroom and leaving the computer unattended and logged in can be a huge security breach.

Windows 2000 addresses these problem by having an option to encrypt files, so even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows 2000 encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and new files moved to them or created in them to be encrypted as well. The actual encryption and decryption is not done by NTFS itself, but by a driver called **EFS (Encrypting File System)**, which is positioned between NTFS and the user process. In this way, application programs are unaware of encryption and NTFS itself is only partially involved in it.

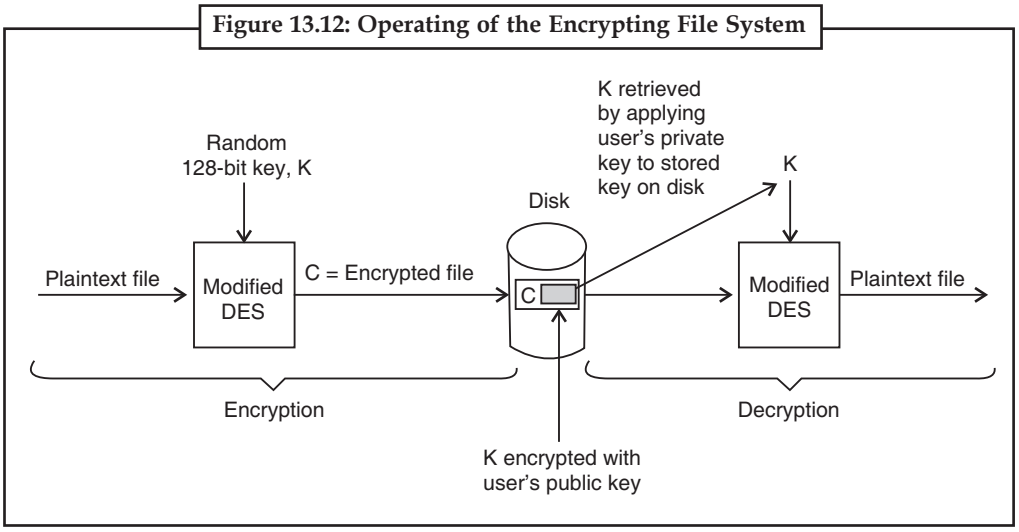
To understand how the encrypting file system works, it is necessary to understand how modern cryptography works.

Now let us see how Windows 2000 encrypts files. When the user asks a file to be encrypted, a random 128-bit file key is generated and used to encrypt the file block by block using a symmetric algorithm parametrized by this key. Each new file encrypted gets a different 128-bit random file key, so no two files use the same encryption key, which increases security in ease one key is compromised. The current encryption algorithm is a variant of **DES (Data Encryption Standard)**, but the EFS architecture supports the addition of new algorithms in the future. Encrypting each block independently of all the others is necessary to make random access still possible.

The file key has to be stored somewhere so the file can be decrypted later. If it were just stored on the disk in plaintext, then someone who stole or found the computer could easily decrypt the file, defeating the purpose of encrypting the files. For this reason, the file keys must all be encrypted before they are stored on the disk. Public-key cryptography is used for this purpose.

After the file is encrypted, the location of the user's public key is looked up using information in the registry. There is no danger of storing the public key's location in the registry because if a thief steals the computer and finds the public key, there is no way to deduce the private key from it. The 128 bit random file key is now encrypted with the public key and the result stored on disk along with the file, as shown in Figure 13.12.

Figure 13.12: Operating of the Encrypting File System



Notes

To decrypt a file, the encrypted 128-bit random file key is fetched from disk. However, to decrypt it and retrieve the file key, the user must present the private key. Ideally, this should be stored on a smart card, external to the computer, and only inserted in a reader when a file has to be decrypted. Although Windows 2000 supports smart cards, it does not store private keys on them.

Instead, the first time a user encrypts a file using EFS, Windows 2000 generates a (private key, public key) pair and stores the private key on disk encrypted using a symmetric encryption algorithm. The key used for the symmetric algorithm is derived either from the user's login password or from a key stored on the smart card, if smart card login is enabled. In this way, EFS can decrypt the private key at login time and keep it within its own virtual address space during normal operation so it can decrypt the 128-bit file keys as needed without further disk accesses. When the computer is shut down, the private key is erased from EFS' virtual address space so anyone stealing the computer will not have access to the private key.

A complication occurs when multiple users need access to the same encrypted file. Currently the shared use of encrypted files by multiple users is not supported. However, the EFS architecture could support sharing in the future by encrypting each file's key multiple times, once with the public key of each authorized user. All of these encrypted versions of the file key could be attached to the file.

The potential need to share encrypted files is one reason why this two-key system is used. If all files were encrypted by their owner's key, there would be no way to share any files. By using a different key to encrypt each file, this problem can be solved.

Having a random file key per file but encrypting it with the owner's symmetric key does not work because having the symmetric encryption key just lying around in plain view would ruin the security — generating the decryption key from the encryption key is too easy. Thus (slow) public-key cryptography is needed to encrypt the file keys. Because the encryption key is public anyway, having it lying around is not dangerous.

The other reason the two-key system being used is performance. Using public-key cryptography to encrypt each file would be too slow. It is much more efficient to use symmetric-key cryptography to encrypt the data and public-key cryptography to encrypt the symmetric file key.

Self Assessment

Multiple choice questions:

1. The I/O manager is on intimate terms with ..... manager.
   (*a*) plug-and-play (*b*) play
   (*c*) plug (*d*) none of these
2. Windows is based on ..... system.
   (*a*) character base (*b*) graphics base
   (*c*) event (*d*) none of these
3. Windows 2000 supports most important file which are ..... .
   (*a*) FAT 16 (*b*) FAT 32
   (*c*) NTFS (*d*) All of these
4. Windows 2000 NTF is used in ..... file system.
   (*a*) vertical (*b*) hierarchical
   (*c*) both (*d*) none of these

Notes

13.3 Security in Windows 2000

Having just looked at encryption in the file system, this is a good time to examine security in general. NT was designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange site. This standard requires operating systems to have certain properties in order to be classified as secured enough for certain kinds of military work. Although Windows 2000 was not specifically designed for C2 compliance, it inherits many security properties from NT, including the following:

1. Secure login with antispoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

Let us review these items briefly (none of which are met by Windows 98, incidentally).

Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and the user is told that login has failed. Windows 2000 prevents this attack by instructing users to hit CTRL-ALT-DEL to log in. This key sequence is always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address space protection simply means that each process has its own protected virtual address space not accessible by any unauthorized process. The next item means that when a stack grows, the pages mapped in are initialized to zero so processes cannot find any old information put there by the previous owner. Finally, security auditing allows the administrator to produce a log of certain security-related events.

In the next section we will describe the basic concepts behind Windows 2000 security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented.

13.3.1 Fundamental Concepts of Security in Windows 2000

Every Windows 2000 user (and group) is identified by a **SID (Security ID)**. SIDs are binary numbers with a short header followed by a long random component. Each SID is intended to be unique worldwide. When a user starts up a process, the process and its threads run under the user's SID. Most of the security system is designed to make sure that each object can be accessed only by threads with authorized SIDs.

Each process has an **access token** that specifies its SID and other properties. It is normally assigned at login time by winlogon and is shown in Figure 13.13, although processes should call GetTokenInformation to acquire this information since it may change in the future. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not used. The Groups fields specify the groups to which the process belongs: this is needed for POSIX conformance. The default **DACL (Discretionary ACL)** is

Notes

the access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDs are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

Finally, the privileges listed, if any, give the process special powers, such as the right to shut the machine down or access files to which access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. In summary, the access token tells who owns the process and which defaults and powers are associated with it.

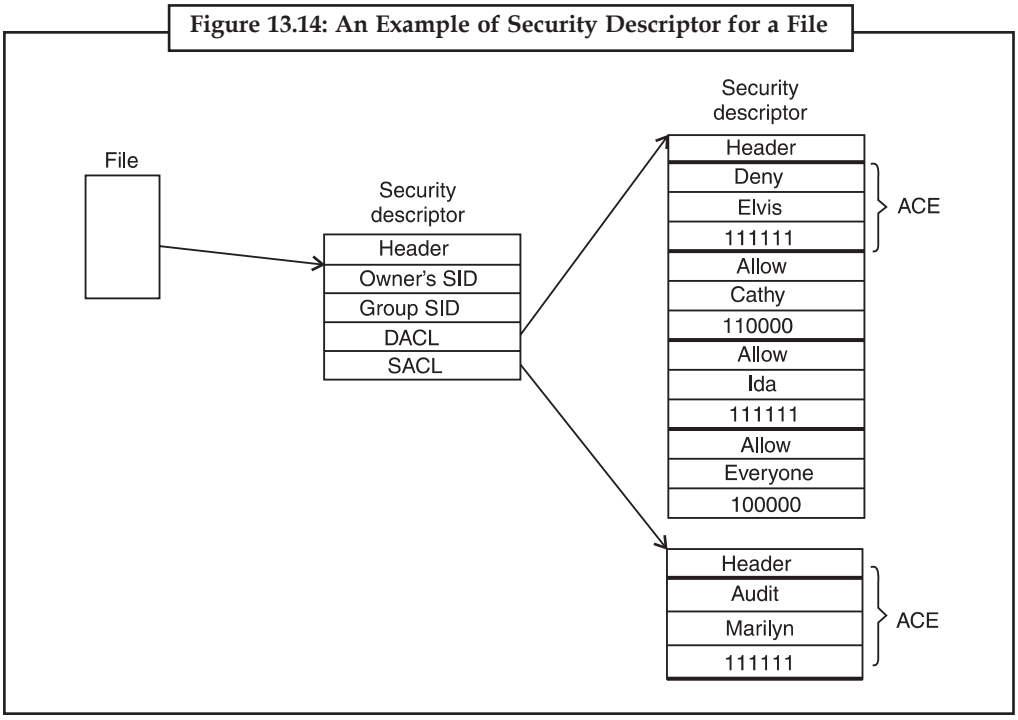
Figure 13.13: Structure of an Access Token

Header	Expiration time	Groups	Default DACL	User SID	Group SID	Restricted SIDs	Privileges

When a user logs in, winlogon gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process' access token initially applies to all the threads in the process. However, a thread can acquire a different access token during execution, in which case the thread's access token overrides the process' access token. In particular, a client thread can pass its access token to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**.

Another basic concept is the **security descriptor**. Every object has a security descriptor associated with it that tells who can perform which operations on it. A security descriptor consists of a header followed by a DACL with one or more **ACEs (Access Control Elements)**. The two main kinds of elements are Allow and Deny. An allow element specifies a SID and a bitmap that specifies which operations processes with that SID may perform on the object. A deny element works the same way, except a match means the caller may not perform the operation. For example, Ida has a file whose security descriptor specifies that everyone has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full access. This simple example is illustrated in Figure 13.14. The SID everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

Figure 13.14: An Example of Security Descriptor for a File



Notes

In addition to the DACL, a security descriptor also has a **SACL (System Access Control List)**, which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the system-wide security event log. In Figure 13.14, every operation that Marilyn performs on the file will be logged. Windows 2000 provides additional auditing features to log sensitive accesses.

13.3.2 Security API Calls

Most of the Windows 2000 access control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateFile, or other object creation call. This security descriptor then becomes the security descriptor attached to the object, as we saw in Figure 13.14. If no security descriptor is provided in the object creation call, the default security in the caller's access token (see Figure 13.13) is used instead.

Many of the Win32 API security calls relate to the management of security descriptors, so we will focus on those here. The most important calls are listed in Figure 13.15. To create a security descriptor, storage for it is first allocated and then initialized using InitializeSecurityDescriptor. This call fills in the header. If the owner SID is not known, it can be looked up by name using LookupAccountSid. It can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the caller's groups, but the system administrator can fill in any SIDs.

Figure 13.15: The Principal Win32 API Functions for Security

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

At this point the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce, and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, more like on an existing ACL than on one being constructed for the first time. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

Notes

13.3.3 Implementation of Security

Security in a standalone Windows 2000 system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this site). Logging in is handled bywinlogon and authentication is handled by lsass and msgina.dll. The result of a successful login is a new shell with its associated access token. This process uses the SECURITY and SAM keys in the registry. The former sets the general security policy and the latter contains the security information for the individual users.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security manager checks to see if the caller has all the rights required. It performs this check by looking at the caller's access token and the DACL associated with the object it goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. If all the rights of the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Any log entries required by the SACL are made.

13.4 Caching in Windows 2000

The Windows 2000 cache manager does caching for performance reasons, conceptually similar to caches in other operating systems. However, its design has some unusual properties that are worth looking at briefly.

The cache manager's job is to keep file system blocks that have been used recently in memory to reduce access time on any subsequent reference. Windows 2000 has a single integrated cache that works for all the file systems in use, including NTFS, FAT-32, FAT-16, and even CD-ROM file systems. This means that the file systems do not need to maintain their own caches.

As a consequence of the design goal to have a single integrated cache despite the presence of multiple file systems, the cache manager is located in an unusual position in the system. It is not a part of the file system because there are multiple independent file systems that may have nothing in common. Instead it operates at a higher level than the file systems, which are technically drivers under control of the I/O manager.

The Windows 2000 cache is organized by the virtual block, not by the physical block. The traditional file caches keep track of blocks by two-part addresses of the form (partition, block), where the first member denotes a device and partition and the second member is a block number within that partition. The Windows 2000 cache manager does not do that. Instead, it uses (file, offset) to refer to a block. The reason for this unorthodox arrangement is that when a request comes in to the cache manager, it is specified as (file, offset) because that is all the calling process knows. If cache blocks had been labeled by a (partition, block) tag, the cache manager would have no way of knowing which (file, offset) block corresponds to which (partition, block) block since it is the file systems that maintain those mappings.

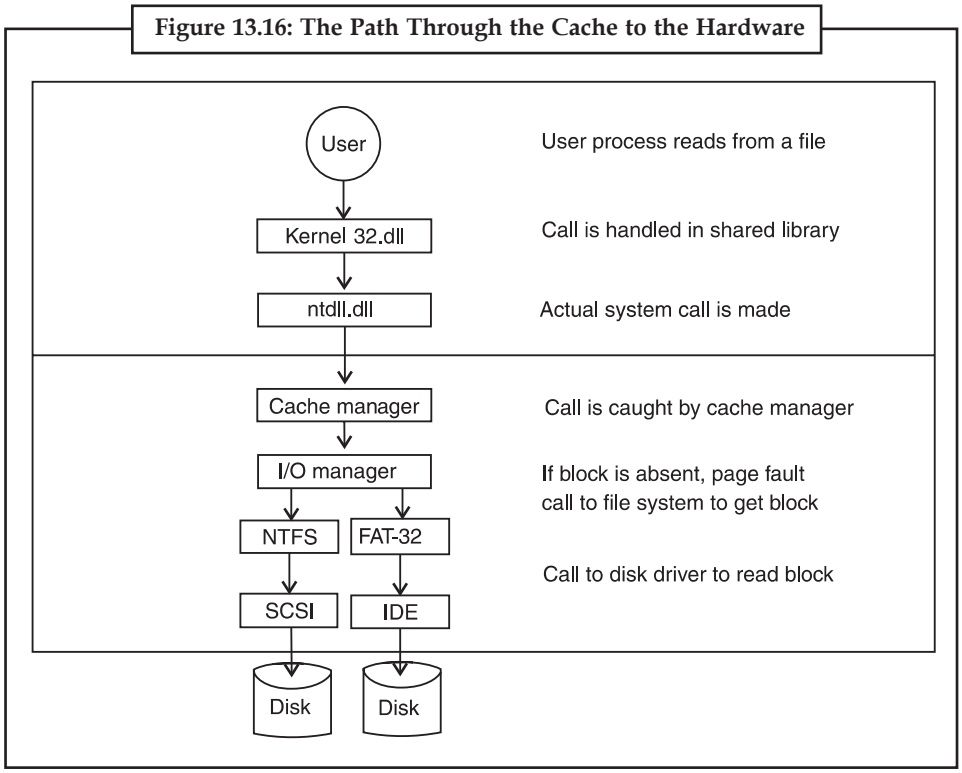
Notes

Let us now examine how the cache manager works. When a file is referenced, the cache manager maps a 256 KB chunk of kernel virtual address space onto the file. If the file is larger than 256 KB, only a portion of the file is mapped. The total amount of virtual address space the cache manager can use is determined at boot time and depends on the amount of RAM present. If the cache manager runs out of 256 KB chunks of virtual address space, it must unmap an old file before mapping in a new one.

Once a file is mapped, the cache manager can satisfy requests for its blocks by just copying from kernel virtual address space to the user buffer. If the block copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual away. The cache manager is not even aware of whether the block was in the cache or not. The copy always succeeds.

The operation of the cache manager is shown in Figure 13.16 in the case of an NTFS file system on a SCSI disk and a FAT-32 file system on an IDE disk. When a process does a read on a file, the request is directed to the cache manager. If the block needed is in the cache, it is copied to the user immediately. If it is not in the cache, the cache manager gets a page fault when trying to copy it. When the page fault has been handled, the block is copied to the calling process.

Figure 13.16: The Path Through the Cache to the Hardware



As a consequence of this design, the cache manager does not know how many of its mapped pages are in physical memory or even how large its cache is. Only the memory manager knows for sure. This approach allows the memory manager to dynamically trade off the size of the cache against memory for user pages. If there is little file activity but there are many processes active, the memory manager can use most of physical memory for process pages. On the other hand, if there is a lot of file activity and few processes, more physical memory can be devoted to the cache.

Another property the cache manager has is that coherence is maintained between memory-mapped files and the files that are open for reading and writing. Consider, for example, a situation

Notes

in which one process opens some file for reading and writing and a second process maps that file onto its address space. What happens if the second process writes on the file directly and immediately thereafter the first one reads in a block that was just changed? Does it get stale data? The answer is no. In both cases open files and mapped files, the cache manager maps a 256 KB piece of its virtual address space onto the file. The file is only mapped once, no matter how many processes have it open or mapped. In the case of a mapped file, both the cache manager and the user process share pages in memory. When a read request has to be satisfied, the cache manager just copies the page from memory to the user buffer, which always reflects the exact current state of the file because the cache manager is using the same pages as the process that has the file mapped in.


Caution

Before installing the operating system the customer setting and the data base have to be saved.

Self Assessment

Fill in the blanks:

5. In a file, encryption is done by a driver called ..... .
6. The Windows 2000 cache is organised by ..... block.
7. ..... hard disk drive is a non-volatile, random access device for digital data.

True or False:

8. The zip drive is a medium-capacity removable disk storage system.
9. SIDs are binary numbers with a short header followed by a long random component.

13.5 Summary

- The I/O manager is on intimate terms with the plug-and-play manager. The basic plug and play is that of an enumerable bus.
- Microsoft has defined a Windows Driver model that device drivers are expected to conform with.
- Windows 2000 supports some important files like FAT-16, FAT-32 and NTFS.
- Secure login means that the system administrator can require all user to have password in order to log in.
- Win32 API security calls relate to the management of security descriptors.
- The caches manager jobs keep file systems blocks that have been used recently in memory to reduce access time on any subsequent reference.

13.6 Keywords

Bitmaps: Bitmap or pixmap is a type of memory organization or image file format used to store digital images.

Dynamic Disks: The Dynamic Disk is a physical disk that manages its volumes by using LDM database.

Notes

FAT: File Allocation Table (FAT) is a computer file system architecture now widely used on many computer systems and most memory cards, such as those used with digital cameras.

NTFS: NTFS is the standard file system of Windows NT, including its later versions Windows 2000, Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista and Windows 7.

SCSI Hard Disk: A hard disk drive (HDD) is a non-volatile, random access device for digital data. It features rotating rigid platters on a motor-driven spindle within a protective enclosure.

Zip Driver: The Zip drive is a medium-capacity removable disk storage system that was introduced by Iomega in late 1994. Originally, Zip disks launched with capacities of 100 MB, but later versions increased this to first 250 MB and then 750 MB.

13.7 Review Questions

1. Explain the concept of Input/output in Windows 2000.
2. Discuss the following terms in brief.
   (*a*) NTFS (*b*) FAT
   (*c*) CD-ROM (*d*) SCSI
3. Give the explanation about the SCSI hard disk.
4. What are I/O API calls in windows?
5. Explain the implementation of I/O. Give example.
6. How to use the file system in windows? Give the example and discuss in brief.
7. Explain the concept of security in Windows 2000. Discuss security API calls.

Answers to Self Assessment

1. (*a*) 2. (*b*) 3. (*d*) 4. (*b*) 5. EFS
6. virtual 7. SCSI 8. True 9. True

13.8 Further Readings


Books

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes. *Introduction to Operating Design and Implementation,* by Michael Kifer, Scoott A. Smolka.


Online link

wiley.com/coolege.silberschatz

Unit 14: Case Study of Linux Operating System

Notes

CONTENTS

Objectives

Introduction

14.1 Overview of Linux
- 14.1.1 Linux Goals
- 14.1.2 Interfaces to Linux
- 14.1.3 Shell
- 14.1.4 Linux Utility Programs
- 14.1.5 Kernel Structure

14.2 Processes in Linux
- 14.2.1 Fundamental Concepts of Processes in Linux
- 14.2.2 Process Management System Calls in Linux
- 14.2.3 Implementation of Processes and Threads in Linux
- 14.2.4 Threads in Linux
- 14.2.5 Scheduling in Linux

14.3 Booting Linux

14.4 Memory Management in Linux
- 14.4.1 Fundamental Concepts of Memory Management in Linux
- 14.4.2 Memory Management System Calls in Linux
- 14.4.3 Implementation of Memory Management in Linux
- 14.4.4 Physical Memory Management
- 14.4.5 Memory Allocation Mechanisms
- 14.4.6 Virtual Address Space Representation
- 14.4.7 Paging in Linux
- 14.4.8 Page Replacement Algorithm

14.5 Input/Output in Linux
- 14.5.1 Fundamental Concepts of Input/Output in Linux
- 14.5.2 Networking
- 14.5.3 Input/Output System Calls in Linux
- 14.5.4 Implementation of Input/Output in Linux
- 14.5.5 Modules in Linux

14.6 Linux File System
- 14.6.1 Fundamental Concepts of Linux File System
- 14.6.2 File System Calls in Linux
- 14.6.3 Implementation of the Linux File System
- 14.6.4 Linux Virtual File System
- 14.6.5 Linux Extended File System—Ext2
- 14.6.6 Linux Ext3 File System
- 14.6.7 /proc File System
- 14.6.8 NFS—The Network File System

Notes

14.7 Security in Linux
14.8 Summary
14.9 Keywords
14.10 Review Questions
14.11 Further Readings

Objectives

After studying this unit, you will be able:

- Discuss overview of Linux
- Explain process of Linux
- Understand of Booting process of Linux
- Explain Memory management of Linux
- Explain Input/output of Linux
- Discuss Linux file system
- Understand Security of Linux

Introduction

During the early years of MINIX development and discussion on the Internet, many people requested (or in many cases, demanded) more and better features, to which the author often said "No" (to keep the system small enough for students to understand completely in a one-semester university course). This continuous "No" irked many users. At this time, FreeBSD was not available, so that was not an option. After a number of years went by like this, a Finnish student, Linus Torvalds, decided to write another UNIX clone, named **Linux**, which would be a full-blown production system with many features that MINIX was initially lacking.

The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed numerous ideas from MINIX, ranging from the structure of the source tree to the layout of the file system. However, it was a monolithic rather than a microkernel design, with the entire operating system in the kernel. The code size totaled 9,300 lines of C and 950 lines of assembler, roughly similar to MINIX version in size and also roughly comparable in functionality.

Linux rapidly grew in size and evolved into a full production UNIX clone as virtual memory, a more sophisticated file system, and many other features were added. Although it originally ran only on the 386 (and even had embedded 386 assembly codes in the middle of C procedures), it was quickly ported to other platforms and now runs on a wide variety of machines, just as UNIX does. One difference with UNIX does stand out however—Linux makes use of many special features of the gcc compiler and would need a lot of work before it would compile with an ANSI standard C compiler.

The next major release of Linux was version 1.0, issued in 1994. It was about 165,000 lines of code and included a new file system, memory-mapped files and BSD-compatible networking with sockets and TCP/IP. It also included many new device drivers. Several minor revisions followed in the next two years. By this time, Linux was sufficiently compatible with UNIX that a vast amount of UNIX software was ported to Linux, making it far more useful than it would have otherwise been. In addition, a large number of people were attracted to Linux and began working on the code and extending it in many ways under Torvalds' general supervision.

The next major release, 2.0 was made in 1996. It consisted of about 470,000 lines of C and 8000 lines of assembly code. It included support for 64 bit architectures, symmetric multiprogramming, new networking protocols and numerous other features. A large fraction of the total code mass was taken up by an extensive collection of device drivers. Additional

Notes

releases followed frequently. The version numbers of the Linux kernel consist of four numbers—*A, B, C* and *D*, the first number denotes the kernel version. The second number denotes the major revision. Prior to the 2.6 kernel, even revision numbers corresponded to stable kernel releases, whereas odd ones corresponded to unstable revisions, under development. With the 2.6 kernel that is no longer the case. The third number corresponds to the minor revisions, such as support for new drivers. The fourth number corresponds to minor bug fixes or security patches. A large array of standard UNIX software has been ported to Linux, including the X Window System and a great deal of networking software. Two different GUIs (GNOME and KDE) have also been written for Linux. In short, it has grown to a full-blown UNIX clone with all the bells and whistles a UNIX lover might want.

One unusual feature of Linux is its business model—it is free software. It can be downloaded from various sites on the Internet, for example: *www.kernel.org*. Linux comes with a license devised by Richard Stallman, founder of the Free Software Foundation. Despite the fact that Linux is free, this license, the **GPL (GNU Public License)**, is longer than Microsoft's Windows license and specifies what you can and cannot do with the code. Users may use, copy, modify, and redistribute the source and binary code freely. The main restriction is that all works derived from the Linux kernel may not be sold or redistributed in binary form only; the source code must either be shipped with the product or be made available on request.

Although Torvalds still controls the kernel fairly closely, a large amount of user-level software has been written by numerous other programmers, many of them originally migrated over from the MINIX, BSD, and GNU online communities. However, as Linux evolves, a steadily smaller fraction of the Linux community want to hack source code (witness hundreds of books telling how to install and use Linux and only a handful discussing the code or how it works). Also, many Linux users now forego the free distribution on the Internet to buy one of many CD-ROM distributions available from numerous competing commercial companies.

A Website listing the current top 100 top Linux distributions is *www.distrowatch.org*. As more and more software companies start selling their own versions of Linux and more and more hardware companies offer to preinstall it on the computers they ship, the line between commercial software and free software is beginning to blur substantially. As a footnote to the Linux story, it is interesting to note that just as the Linux bandwagon was gaining steam, it got a big boost from an unexpected source AT&T. In 1992, Berkeley, by now running out of funding, decided to terminate BSD development with one final release, 4.4BSD, (which later formed the basis of FreeBSD). Since this version contained essentially no AT&T code, Berkeley issued the software under an open source license (not GPL) that let everybody do whatever they wanted with it except one thing—sue the University of California. The AT&T subsidiary controlling UNIX promptly reacted by—you guessed it—suing the University of California. It simultaneously sued a company, BSDI, set up by the BSD developers to package the system and sell support, much as Red Hat and other companies now do for Linux. Since virtually no AT&T code was involved, the lawsuit was based on copyright and trademark infringement, including items such as BSDI's 1-800-ITS-UNIX telephone number. Although the case was eventually settled out of court, this legal action kept FreeBSD off the market just long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been a serious competition between two free, open source UNIX systems: the reigning champion, BSD, a mature and stable system with a large academic following dating back to 1977 versus the vigorousyoung challenger, Linux, just two years old but with a grow by following among individual users. Who knows how this battle of the free UNICES would have turned out?



Did u know?

Linux began in 1991 with the commencement of a personal project by a Finnish student, Linus Torvalds.

Notes

14.1 Overview of Linux

In this section, we will provide a general introduction to Linux and how it is used, for the benefit of readers not already familiar with it. Nearly all of this material applies to just about all UNIX variants with only small deviations. Although Linux has several graphical interfaces, the focus here is how Linux appears to a programmer working in a shell window on X. Subsequent sections will focus on system calls and how it works inside.

14.1.1 Linux Goals

UNIX was always an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways. The model of a group of experienced programmers working together closely to produce advanced software is obviously very different from the personal computer model of a single beginner working alone with a word processor, and this difference is reflected throughout UNIX from start to finish. It is only natural that Linux inherited many of these goals, even though the first version was for a personal computer. What is it that good programmers want in a system? To start with, most like their systems to be simple, elegant, and consistent. For example, at the lowest level, a file should just be a collection of bytes. Having different classes of files for sequential access, random access, keyed access, remote access, etc. (as mainframes do) just gets in the way.

Similarly, if the command

ls A*

means list all the files beginning with ''A'' then the command

rm A*

should mean remove all the files beginning with "A" and not remove the one file whose name consists of an ''A'' and an asterisk. This characteristic is sometimes called the principle of least surprise.

Another thing that experienced programmers generally want is power and flexibility. This means that a system should have a small number of basic elements that can be combined in an infinite variety of ways to suit the application. One of the basic guidelines behind Linux is that every program should do just one thing and do it well. Thus compilers do not produce listings, because other programs can do much better.

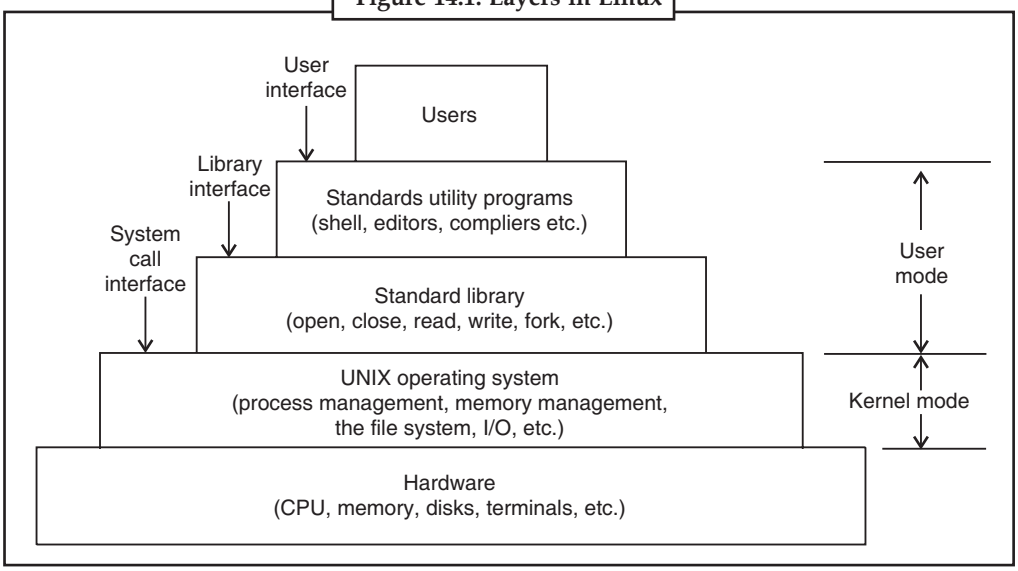
Finally, most programmers have a strong dislike for useless redundancy. Why type copy when *cp* is enough? To extract all the lines containing the string "ard" from the file *f*, the Linux programmer types *grep* ard *f*. The opposite approach is to have the programmer first select the *grep* program (with no arguments), and then have *grep* announce itself by saying: "Hi, I'm *grep*, I look for patterns in files. Please enter your pattern." After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and ask if that is correct. While this kind of user interface may or may not be suitable for rank novices, it irritates skilled programmers to no end.

14.1.2 Interfaces to Linux

A Linux system can be regarded as a kind of pyramid, as illustrated in Figure 14.1. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

Notes

Figure 14.1: Layers in Linux



Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language, but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the *read* library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls. In addition to the operating system and system call library, all versions of Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between Linux versions.

These include the command processor (shell), compilers, editors, text processing programs, and file manipulation utilities. It is the programs that a user at the keyboard invokes.

Thus we can speak of three different interfaces to Linux—the true system call interface, the library interface, and the interface formed by the set of standard utility programs. Most personal computer versions of Linux have replaced this keyboard-oriented user interface with a mouse-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes Linux so popular and has allowed it to survive numerous changes in the underlying technology so well. The GUI for Linux is similar to the first GUIs developed for UNIX systems in the 1970s, and popularized by Macintosh and later Windows for PC platforms. The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. Popular desktop environments for Linux include GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment). GUIs on Linux are supported by the X Windowing System, or commonly X11 or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as keyboards, mouse, screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, *xlib*, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of X11 by enriching the windows view, providing buttons, menus, icons, and other options. The X server can be started manually, from a command line, but is typically started during the boot process by a display manager,

Notes

which displays the graphical login screen where a user enters his username and password. When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open file, drag and drop to copy files from one location to another, etc. In addition, users may invoke a terminal emulator program, or *xterm*.

14.1.3 Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command line interface, called the **shell**. Often they start one or more shell windows from the graphical user interface and just work in them. The shell command line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (bash). It is heavily based on the original UNIX shell, Bourne shell, and in fact its name is an acronym for Bourne Again Shell. Many new shells are also in use (ksh, csh, etc.), however, bash is the default shell in most Linux systems. When the shell starts up, it initializes itself, then types a **prompt** character, often a per cent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, assumes it is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write to the monitor and the power to execute other programs. Commands may take arguments, which are passed to the called program as character strings.

For example, the command line

cp src dest

invokes the cp program with two arguments, src and dest. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy dest.

Not all arguments are file names. In *head* 20 file the first argument, 20, tells *head* to print the first 20 lines of file, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called **flags**, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command head 20 file is perfectly legal, and tells *head* to first print the initial 10 lines of a file called *20*, and then print the initial 10 lines of a second file called *file*. Most Linux commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts **magic characters**, sometimes called **wild cards**. An asterisk, for example, matches all possible strings, so

ls *.c

tells *ls* to list all the files whose name ends in *.c* If files named *x.c*, *y.c*, and *z.c* all exist, the above command is equivalent to typing

ls x.c y.c z.c

Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so

ls [ape]*

lists all files beginning with ''a'', ''p'', or ''e''. A program like the shell does not have to open the terminal (keyboard and monitor) in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called **standard input**

Notes

(for reading), a file called **standard output** (for writing normal output), and a file called **standard error** (for writing error messages). Normally, all three default to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many Linux programs read from standard input and write to standard output as the default. For example, sort invokes the *sort* program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen. It is also possible to redirect standard input and output, as is often useful. The syntax for redirecting standard input uses a less than sign (<) followed by the input file name. Similarly, standard output is redirected using a greater than sign (>). It is permitted to redirect both in the same command. For example, the command sort <in >out causes *sort* to take its input from the file *in* and write its output to the file *out*. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a **filter**. Consider the following command line consisting of three separate commands:

sort <in >temp; head –30 <temp; rm temp

It first runs *sort*, taking the input from *in* and writing the output to *temp*. When that has been completed, the shell runs *head*, telling it to print the first 30 lines of *temp* and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed. It frequently occurs that the first program in a command line produces output that is used as the input on the next program. In the above example, we used the file *temp* to hold this output. However, Linux provides a simpler construction to do the same thing.

In

sort <in | head –30

the vertical bar, called the **pipe symbol**, says to take the output from *sort* and use it as the input to *head*, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a **pipeline**, may contain arbitrarily many commands. A four-component pipeline is shown by the following example:

grep ter *.t | sort | head –20 | tail –5 >foo

Here all the lines containing the string ''ter'' in all the files ending in *.t* are written to standard output, where they are sorted. The first 20 of these are selected out by *head* which passes then to *tail*, which writes the last five (i.e., lines 16 to 20 in the sorted list) to *foo*. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus wc -l <a >b & runs the word count program, *wc*, to count the number of lines (*-l* flag) in its input, a, writing the result to *b*, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by sort <x | head & Multiple pipelines can run in the background simultaneously.

It is possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell commands are called **shell scripts**. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use if, for, while, and case constructs. Thus a shell script is really a program

Notes

written in shell language. The Berkeley C shell is an alternative shell that has been designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, various other people have written and distributed a variety of other shells.

14.1.4 Linux Utility Programs

The command-line (shell) user interface to Linux consists of a large number of standard utility programs. These programs can be divided into six categories as follows:

1. File and directory manipulation commands.
2. Filters.
3. Program development tools such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

The POSIX 1003.2 standard specifies the syntax and semantics of just under 100 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all Linux systems.

In addition to these standard utilities, there are many application programs as well, such as Web browsers, image viewers, etc. Let us consider some examples of these programs, starting with file and directory manipulation. cp a b copies file *a* to *b*, leaving the original file intact. In contrast, mv a b copies *a* to *b* but removes the original. In fact, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using cat, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the rm command. The chmod command allows the owner to change the rights bits to modify access permissions. Directories can be created with mkdir and removed with rmdir. To see a list of the files in a directory, it can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more. We have already seen several filters: grep extracts lines containing a given pattern from standard input or one or more input files; sort sorts its input and writes it on standard output; head extracts the initial lines of its input; tail extracts the final lines of its input. Other filters defined by 1003.2 are cut and paste, which allow columns of text to be cut and pasted into files; od which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; tr, which does character translation (e.g., lower case to upper case), and pr which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include gcc, which calls the C compiler, and ar, which collects library procedures into archive files. Another important tool is make, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are **header files**, which contain type, variable, macro, and other declarations. Source files often include these using a special include directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it, and recompile them. The function of make is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all Linux programs, except the smallest ones, are set up to be

Notes

compiled with make. A selection of the POSIX utility programs is listed in Figure 14.2, along with a short description of each one. All Linux systems have these programs, and many more.

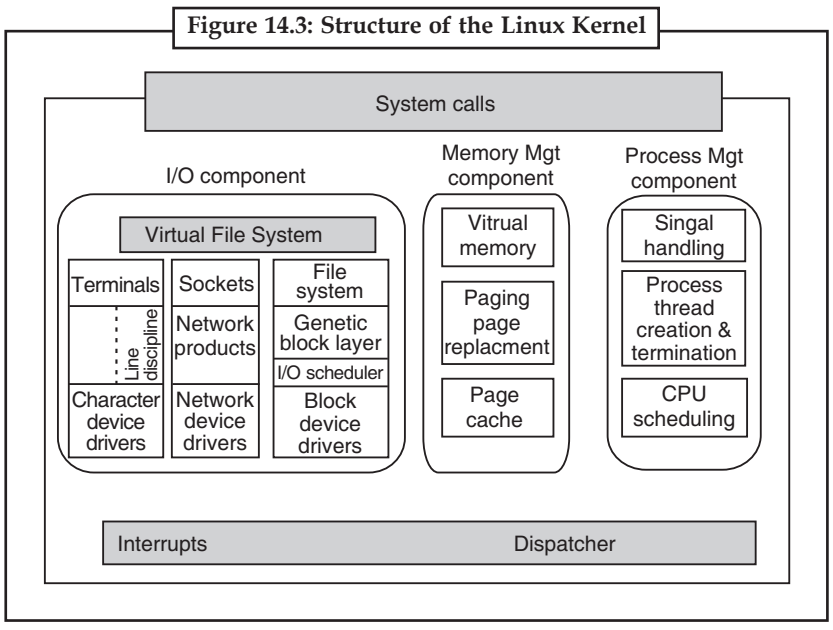
Figure 14.2: A Few of the Common Linux Utility Program Required by POSIX

Program	Typical use
catt	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file or lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

14.1.5 Kernel Structure

In Figure 14.1, we saw the overall structure of a Linux system. Now let us zoom in and look more closely at the kernel before examining the various parts.

Figure 14.3: Structure of the Linux Kernel



Notes

The kernel sits directly on the hardware and enables interactions with various devices, the system memory and controls CPU accesses. At the lowest level, as shown in Figure 14.3, it contains interrupt handlers which are the primary way for interacting with devices, and low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is the time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling. Next, we divide the various kernel subsystems into three main components.

The I/O component in Figure 14.3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a Virtual File System layer. That is, at the top level, performing a read operation to a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as either character device drivers or block device drivers, with the main difference that seeks and random accesses are allowed on block devices and not on character devices.

Technically, network devices are character devices, but they are handled somewhat differently that it is probably clearer to separate them, as has been done in the figure. Above the device driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like vi and emacs, want every key stroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, and allows users to edit the whole line before hitting ENTER to send it to the program.

In this case the character stream from the terminal device is passed through a so called line discipline, and appropriate formatting is applied. Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, always including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later. On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk operation requests in a way that tries to conserve wasteful disk head movement, or to meet some other system policy.

At the very top of the block device columns are the file systems. Linux may have, and it does in fact, multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block device layer provides an abstraction used by all file systems.

To the right in Figure 14.3 are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory management tasks include maintaining the virtual to physical memory mappings, maintaining a cache of recently accessed pages and implementing a good page replacement policy, and on-demand bringing in new pages of needed code and data into memory. The key responsibility of the process management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

Notes

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may only have an in-memory representation, such as files providing some runtime resource usage information. In addition, the virtual memory systems may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist. In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Figure 14.3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, cause a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

14.2 Processes in Linux

In the previous sections, we started out by looking at Linux as viewed from the keyboard, that is, what the user sees in an xterm window. We gave examples of shell commands and utility programs that are frequently used. We ended with a brief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts that Linux supports, namely, processes, memory, the file system, and input/output. These notions are important because the system calls—the interface to the operating system itself—manipulate them. For example, system calls exist to create processes and threads, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of Linux in existence, there are some differences between them. In this unit, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certain sections (especially implementation sections), the discussion may not apply equally to every version.

14.2.1 Fundamental Concepts of Process in Linux

The main active entities in a Linux system are the processes. Linux processes are very similar to the classical sequential processes. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Linux allows a process to create additional threads once it starts executing.

Linux is a multiprogramming system, so multiple, independent processes may be running at the same time. Each user may have several active processes at once; so on a large system, there may be hundreds or even thousands of processes running. In fact, on most of the single-user workstations, even when the user is absent, dozens of background processes, called **daemons**, are running. These are started by a shell script when the system is booted. (''Daemon'' is a variant spelling of ''demon,'' which is a self-employed evil spirit.) A typical daemon is the *cron daemon*. It wakes up once a minute to check if there is any work for it to do. If so, it does the work. Then it goes back to sleep until it is time for the next check. This daemon is needed because it is possible in Linux to schedule activities minutes, hours, days, or even months in the future. For example, suppose a user has a dentist appointment at 3 o'clock next Tuesday. He can make an entry in the cron daemon's database telling the daemon to beep at him at, say, 2:30. When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process. The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 a.m., or reminding forgetful users

Notes

every year on October 31 to stock up on trick-or-treat goodies for Halloween. Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth. Daemons are straightforward to implement in Linux because each one is a separate process, independent of all other processes. Processes are created in Linux in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process** and the new process is called the **child process**. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward. Changes made to the file by either one will be visible to the other. This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file as well. The fact that the memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty— How do the processes know which one should run the parent code and which one should run the child code? The secret is that the fork system call returns a 0 to the child and a nonzero value, the child's **PID (Process Identifier)** to the parent. Both processes normally check the return value, and act accordingly. As shown in Figure 14.4, processes are named by their PIDs. When a process is created, the parent is given the child's PID, as mentioned above.

Figure 14.4: Process Creation in Linux

mechanical, photocopying, recording, or likewise

```
pid = fork();                    /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error()           /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                 /* parent code goes here. /*/
} else {
                                 /* child code goes here. /*/
}
```

If the child wants to know its own PID, there is a system call, getpid, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants. Processes in Linux can communicate with each other using a form of message passing. It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read. These channels are called **pipes**. Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available. Shell pipelines are implemented with pipes. When the shell sees a line like sort <f | head it creates two processes, sort and head, and sets up a pipe between them in such a way that sort's standard output is connected to head's standard input. In this way, all the data that sort writes go directly to head, instead of going to a file. If the pipe fills up, the system stops running sort until head has removed some data from the pipe.

Processes can also communicate in another way—software interrupts. A process can send what is called a **signal** to another process. Processes can tell the system what they want to happen when a signal arrives. The choices are to ignore it, to catch it, or to let the signal kill the process (the default for most signals). If a process elects to catch signals sent to it, it must specify a signal handling procedure. When a signal arrives, control will abruptly switch to the handler. When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts. A process can only send signals to members of its **process group**,

Notes

which consists of its parent (and further ancestors), siblings, and children (and further descendants). A process may also send a signal to all members of its process group with a single system call. Signals are also used for other purposes. For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0, it gets a SIGFPE (floating-point exception) signal. The signals that are required by POSIX are listed in Figure 14.5. Many Linux systems have additional signals as well, but programs using them may not be portable to other versions of Linux and UNIX in general.

Figure 14.5: The Signals Required by POSIX

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purpose

14.2.2 Process Management System Calls in Linux

Let us now look at the Linux system calls dealing with process management. The main ones are listed in Figure 14.6. Fork is a good place to start the discussion. The Fork system call, supported also by other traditional UNIX systems, is the main way to create a new process in Linux systems. It creates an exact duplicate of the original process, including all the file descriptors, registers and everything else. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the entire parent address space is copied to create the child, subsequent changes in one of them do not affect the other one. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. Using the returned PID, the two processes can see which is the parent and which is the child. In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists).

Notes

Figure 14.6: Some System Calls Relating to Processes

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, & statloc, opts)	Wait for a child to terminate
S = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
S = sigaction(sig. & act, & oldact)	Define action to take on signals
S = sigreturn (&context)	Return from a signal
S = sigprocmask(how, &set, &old)	Examine or change the signal mask
S = sigpending (set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signals mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	suspend the caller until the next signal

Waitpid has three parameters. The return code s is ~ 1 If an error has occurred, pid is a process ID, and residual is the remaining time in the previous alarm. The parameters are what the name suggests. First one allows the caller to wait for a specific child. If it is ~ 1, any old child (i.e., the first child to terminate) will do. The second parameter is the address of a variable that will be set to the child's exit status (normal or abnormal termination and exit value). The third one determines whether the caller blocks or returns if no child is already terminated. In the case of the shell, the child process must execute the command typed by the user. It does this by using the exec system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of fork, waitpid, and exec is shown in Figure. 14.7. In the most general case, exec has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library procedures, including execl, execv, execle, and execve, are provided to allow the parameters to be omitted or specified in various ways. All of these procedures invoke the same underlying system call. Although the system call is exec, there is no library procedure with this name; one of the others must be used. Let us consider the case of a command typed to the shell such as cp file1 file2 used to copy file1 to file2. After the shell has been forked, the child locates and executes the file cp and passes its information about the files to be copied. The main program of cp (and many other programs) contains the function declaration where argc is a count of the number of items on the command line, including the program name. For the example above, argc is 3. The second parameter, argv, is a pointer to an array. Element i of that array is a pointer to the i-th string on the command line. In this example, argv[0] would point to the string ''cp''. Similarly, argv[1] would point to the 5-character string ''file1'' and argv[2] would point to the 5-character string ''file2''. The third parameter of main, envp, is a pointer to the environment, an main(argc, argv, envp) array of strings containing assignments of

Notes

the form name = value used to pass information such as the terminal type and home directory name to a program. In Figure 14.7, no environment is passed to the child, so the third parameter of execve is a zero in this case.

Figure 14.7: A Highly Simplified Shell

```
while (TRUE) {ty                          /* repeat forever /*/
      type_prompt( );                     /* display prompt on the screen */
      read_command(command, params);      /* read input line from keyboard */

      pid = fork( );                      /* fork off a child process */
      if (pid < 0) {                      /* error condition */
            printf("Unable to fork0);     /* error condition */
            continue;

      }

      if (pid) !=0) {
            waitpid (–1, &status, 0);     /* parent waits for child */
      } else {
            execve(command, params, 0);   /* child does the work */
      }
}
```

If exec seems complicated, do not despair; it is the most complex system call. All the rest are much simpler. As an example of a simple one, consider exit, which processes should be used when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable status of the waitpid system call. The low-order byte of status contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child's exit status (0 to 255), as specified in the child's call to exit. For example, if a parent process executes the statement n = waitpid(~1, &status, 0); it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to exit, the parent will be awakened with n set to the child's PID and status set to 0x0400 (0x as a prefix means hexadecimal in C).

The low-order byte of *status* relates to signals; the next one is the value the child returned in its call to exit. If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**. When the parent finally waits for it, the process terminates.

Several system calls relate to signals, which are used in a variety of ways. For example, if a user accidently tells a text editor to display the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. The usual choice is for the user to hit some special key (e.g., DEL or CTRLC), which sends a signal to the editor. The editor catches the signal and stops the print-out.

To announce its willingness to catch this (or any other) signal, the process can use the sigaction system call. The first parameter is the signal to be caught (see Figure. 14.5). The second is a pointer to a structure giving a pointer to the signal handling procedure, as well as some other bits and flags. The third one points to a structure where the system returns information about signal handling currently in effect, in case it must be restored later. The signal handler may run for as long as it wants to. In practice, though, signal handlers are usually fairly short. When the signal handling procedure is done, it returns to the point from which it was interrupted. The sigaction system call can also be used to cause a signal to be ignored, or to restore the default action, which is killing the process. Hitting the DEL key is not the only way to send a signal. The

Notes

kill system call allows a process to signal another related process. The choice of the name ''kill'' for this system call is not an especially good one, since most processes send signals to other ones with the intention that they might be caught. For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the alarm system call has been provided. The parameter specifies an interval, in seconds, after which a SIGALRM signal is sent to the process. A process may have only one alarm outstanding at any instant. If an alarm call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is cancelled by the second call to alarm. If the parameter to alarm is zero, any pending alarm signal is cancelled. If an alarm signal is not caught, the default action is taken and the signaled process is killed. Technically, alarm signals may be ignored, but that is a pointless thing to do.

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided instruction program that is testing reading, speed and comprehension. It displays some text on the screen and then calls alarm to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that a background process or other user might need. A better solution is to use the pause system call, which tells Linux to suspend the process until the next signal arrives.



Task Give the command write file in the Linux operating system.

14.2.3 Implementation of Processes and Threads in Linux

A process in Linux is like an iceberg that what you see is the part above the water, but there is also an important part underneath. Every process has a user part that runs the user program. However, when one of its threads makes a system call, it traps to kernel mode and begins running in kernel context, with a different memory map and full access to all machine resources. It is still the same thread, but now with more power and also its own kernel mode stack and kernel mode program counter. These are important because a system call can block part way through, for example, waiting for a disk operation to complete. The program counter and registers are then saved so the thread can be restarted in kernel mode later.

The Linux kernel internally represents processes as **tasks**, via the structure task struct. Unlike other OS approaches, which make a distinction between a process, lightweight process and thread, Linux uses the task structure to represent any execution context. Therefore, a single-threaded process will be represented with one task structure; a multi-threaded process will have one task structure for each of the user-level threads. Finally, the kernel itself is multi-threaded, and has kernel level threads which are not associated with any user process and are executing kernel code. We will return to the treatment of multi-threaded processes (and threads in general) later in this section. For each process, a process descriptor of type task_ struct is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open file descriptors, etc. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation.

For compatibility with other UNIX systems, Linux identifies processes via the Process Identifier (PID). The kernel organizes all processes in a doubly linked list of task structures. In addition to accessing process descriptors by traversing the linked lists, the PID can be mapped to the address of the task structure, and the process information can be accessed immediately.

Notes

The task structure contains a variety of fields. Some of these fields contain pointers to other data structures or segments, such as those containing information about open files. Some of these segments are related to the user-level structure of the process which is not of interest when the user process in not runnable. Therefore, these may be swapped or paged out, in order not to waste memory on information that is not needed. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable. The information in the process descriptor falls into the following broad categories:

1. **Scheduling Parameters:** Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.
2. **Memory Image:** Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
3. **Signals:** Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
4. **Machine Registers:** When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
5. **System Call State:** Information about the current system call, including the parameters, and results.
6. **File Descriptor Table:** When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
7. **Accounting:** Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
8. **Kernel Stack:** A fixed stack for use by the kernel part of the process.
9. **Miscellaneous:** Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Keeping this information in mind, it is now easy to explain how processes are created in Linux. The mechanism for creating a new process is actually fairly straightforward. A new process descriptor and user area are created for the child process and filled in largely from the parent. The child is given a PID, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

When a fork system call is executed, the calling process traps to the kernel and creates a task structure and few other accompanying data structures, such as the kernel mode stack and a thread_ info structure. This structure is allocated at a fixed offset from the process' end-of-stack, and contains few process parameters, along with the address of the process descriptor. By storing the process descriptor's address at a fixed location, Linux needs only few efficient operations to locate the task structure for a running process. The majority of the process descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, and updates the PID hash table entry to point to the new task structure. In case of collisions in the hash table, process descriptors may be chained. It also sets the fields in thetask_ struct to point to the corresponding previous/next process on the task array. In principle, it should

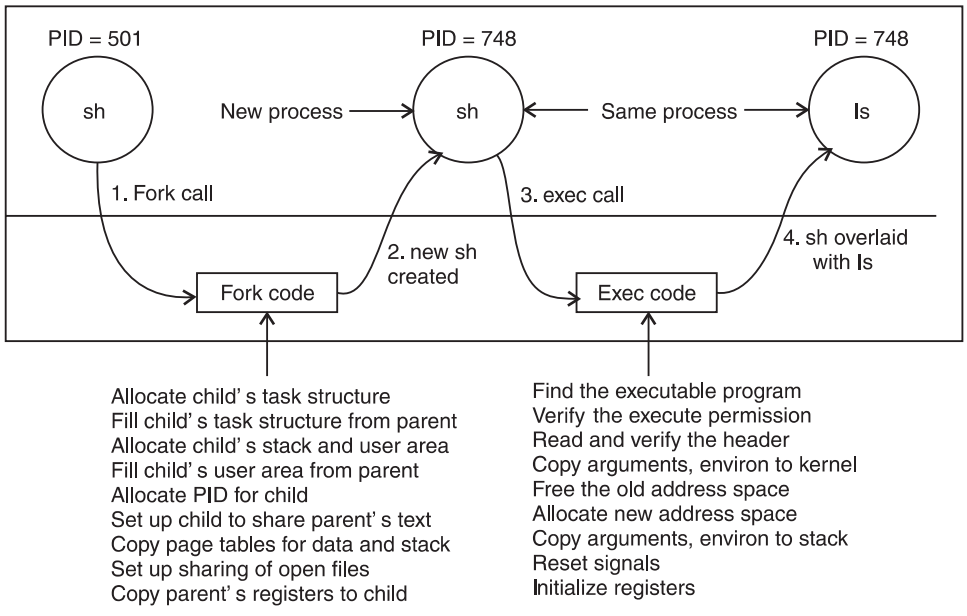
Notes

now allocate memory for the child's data and stack segments, and to make exact copies of the parents' segments, since the semantics of fork say that no memory is shared between parent and child. The text segment may either be copied or shared since it is read only. At this point, the child is ready to run.

However, copying memory is expensive, so all modern Linux systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever the child tries to write on a page, it gets a faulty protection. The kernel sees this and then allocates a new copy of the page to the child and marks it read/write. In this way, only pages that are actually written have to be copied. This mechanism is called **copy on write**. It has the additional benefit of not requiring two copies of the program in memory, thus saving in RAM. After the child process starts running, the code running there (a copy of the shell) does an exec system call giving the command name as a parameter. The kernel now finds and verifies the executable file, copies the arguments and environment strings to the kernel, and releases the old address space and its page tables. Now the new address space must be created and filled in. If the system supports mapped files, as Linux and other UNIX-based systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running. Figure 14.8 illustrates the steps described above through the following example:

A user types a command, ls on the terminal, the shell creates a new process by forking off a clone of itself. The new shell then calls exec to overlay its memory with the contents of the executable file *ls*.

Figure 14.8: The Steps in Executing the Command ls Typed to the Shell



14.2.4 Threads in Linux

We discussed threads in a general way. Here, we will focus on kernel threads in Linux, particularly focusing on the differences in the Linux thread models and other UNIX systems.

Notes

In order to better understand the unique capabilities provided by the Linux model, we start with a discussion of some of the challenging decisions present in multithreaded systems. The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider fork. Suppose that a process with multiple (kernel) threads does a fork system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process? The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread cannot be blocked when calling fork. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the fork. The mutex will never be released and the one thread will hang forever. Numerous other problems exist too. There is no simple solution.

File I/O is another problem area. Suppose that one thread is blocked reading from a file and another thread closes the file or does an lseek to change the current file pointer. What happens next? Who knows? Signal handling is another thorny issue. Should signals be directed at a specific thread or at the process in general? A SIGFPE (floating-point exception) should probably be caught by the thread that caused it. What if it does not catch it? Should just that thread be killed, or all threads? Now consider the SIGINT signal, generated by the user at the keyboard. Which thread should catch that? Should all threads share a common set of signal masks? All solutions to these and other problems usually cause something to break somewhere. Getting the semantics of threads right (not to mention the code) is a nontrivial business. Linux supports kernel threads in an interesting way that is worth looking at. The implementation is based on ideas from 4.4BSD, but kernel threads were not enabled in that distribution because Berkeley ran out of money before the C library could be rewritten to solve the problems discussed above.

Historically, processes were resource containers and threads were the units of execution. A process contained one or more threads that shared the address space, open files, signal handlers, alarms, and everything else. Everything was clear and simple as described above. In 2000, Linux introduced a powerful new system call, clone that blurred the distinction between processes and threads and possibly even inverted the primacy of the two concepts. Clone is not present in any other version of UNIX. Classically, when a new thread was created, the original thread(s) and the new one shared everything but their registers. In particular, file descriptors for open files, signal handlers, alarms, and other global properties were per process, not per thread. What clone did was to make it possible for each of these aspects and others to be process specific or thread specific is called as follows:

pid = clone(function, stack_ ptr, sharing_ flags, arg);

The call creates a new thread, either in the current process or in a new process, depending on sharing_ flags. If the new thread is in the current process, it shares the address space with existing threads and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as POSIX fork.

In both cases, the new thread begins executing at function, which is called with arg as its only parameter. Also in both cases, the new thread gets its own private stack, with the stack pointer initialized to stack_ ptr. The sharing flags parameter is a bitmap that allows a much finer grain of sharing than traditional UNIX systems. Each of the bits can be set independently of the other ones, and each of them determines whether the new thread copies some data structure

Notes

of shares it with the calling thread. Figure 14.9 shows some of the items that can be be shared or copied according to bits in sharing flags.

Figure 14.9: Bits in the Sharing Flags Bitmap

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler tale	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

The CLONE_ VM bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own address space. Having its own address space means that the effect of its STORE instructions are not visible to the existing threads. This behaviour is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

The CLONE_ FS bit controls sharing of the root and working directories and of the umask flag. Even if the new thread has its own address space, if this bit is set, the old and new threads share working directories. This means that a call to chdir by one thread changes the working directory of the other thread, even though the other thread may have its own address space. In UNIX, a call to chdir by a thread always changes the working directory for other threads in its process, but never for threads in another process. Thus this bit enables a kind of sharing not possible in traditional UNIX versions. The CLONE_ FILES bit is analogous to the CLONE_ FS bit. If set, the new thread shares its file descriptors with the old ones, so calls to lseek by one thread are visible to the other ones, again as normally holds for threads within the same process but not for threads in different processes. Similarly, CLONE_ SIGHAND enables or disables the sharing of the signal handler table between the old and new threads. If the table is shared, even among threads in different address spaces, then changing a handler in one thread affects the handlers in the others. Finally, CLONE_ PID controls whether the new thread gets its own PID or shares its parent's PID. This feature is needed during system booting. User processes are not permitted to enable it.

Finally, every process has a parent. The CLONE_ PARENT bit controls who the parent of the new thread is. It can either be the same as the calling thread (in which case the new thread is a sibling of the caller) or it can be the calling thread itself, in which case the new thread is a child of the caller. There are a few other bits that control other items, but they are less important. This fine-grained sharing is possible because Linux maintains separate data structures for the various items. (scheduling parameters, memory image, and so on). The task structure just points to these data structures, so it is easy to make a new task structure for each cloned thread and have it either point to the old thread's scheduling, memory, and other data structures or to copies of them. The fact that such fine-grained sharing is possible does not mean that it is useful, however, especially since traditional UNIX versions do not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

Notes

The Linux thread model raises another difficulty. UNIX systems associate a single PID with a process, independent of whether it is single- or multi-threaded. In order to be compatible with other UNIX systems, Linux distinguishes between a process identifier (PID) and a task identifier (TID). Both fields are stored in the task structure. When clone is used to create a new process which shares nothing with its creator, PID is set to a new value, otherwise, the task receives a new TID, but inherits the PID. In this manner all threads in a process will receive the same PID as the first thread in the process.

14.2.5 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes. Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly-readied real-time FIFO thread with higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. None of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard (''real-time'' extensions to UNIX) which uses those names. The real time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level. The conventional, non-real-time threads are scheduled according to the following algorithm. Internally, the non-real-time threads are associated with priority levels from 100 to 139, i.e., Linux internally distinguishes among 140 priority levels (for real-time and non-real-time tasks). Like for the real-time round-robin threads, Linux associates time quantum values for each of the non-real time priority levels. The quantum is a number of clock ticks the thread may continue to run for. In the current Linux version, the clock runs at 1000Hz and each tick is 1ms, which is called a **jiffy**. Like most UNIX systems, Linux associates a nice value with each thread. The default is 0 but this can be changed using the nice (value) system call, where value ranges from –20 to +19. This value determines the static priority of each thread. A user computing ?to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from ?20 to ?1). Deducing the reason for this rule is left as an exercise for the reader.

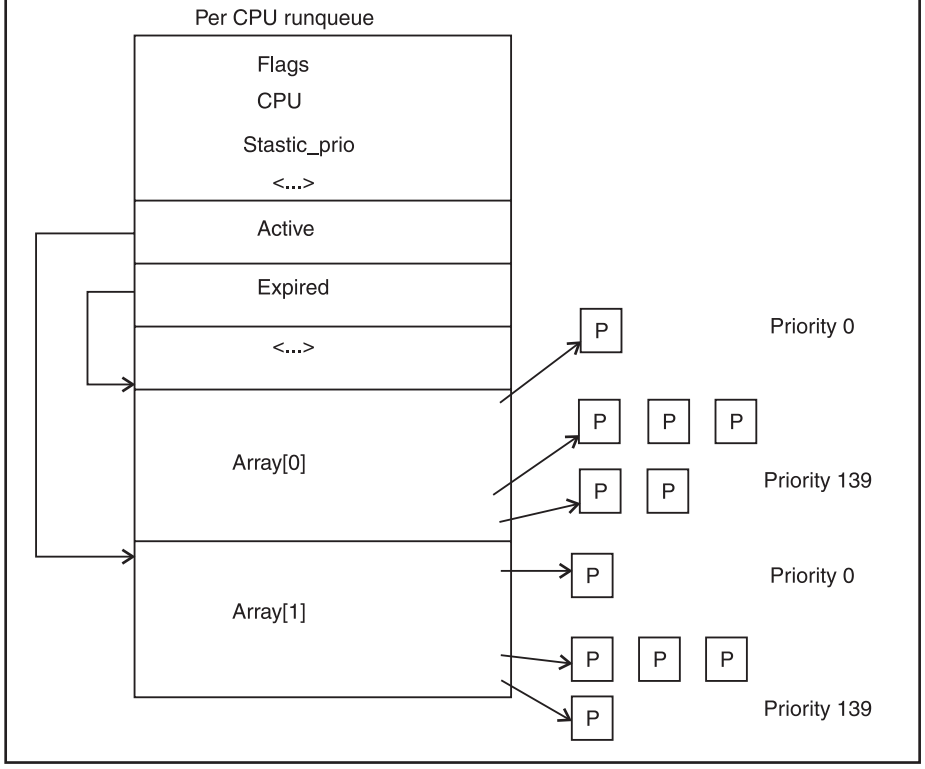
A key data structure used by the Linux scheduler is a **runqueue**. A runqueue is associated with each CPU in the system, and among other information, it maintains two arrays, *active* and *expired*. As shown in Figure 14.10, each of these fields is a pointer to an array of 140 list heads, each corresponding to a different priority.

The list head points to a doubly-linked list of processes at a given priority. The basic operation of the scheduler can be described as follows. The scheduler selects a task from the highest priority active array. If that task's timeslice (quantum) expires, it is moved to an expired list (potentially at a different priority level). If the task blocks, for instance to wait for an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already consumed. Once its timeslice is fully exhausted, it will also be placed on an expired array. When there are no more tasks in any of the active

Notes

arrays, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely to happen). Different priority levels are assigned different timeslice values. Linux assigns higher quanta to higher priority processes. For instance, tasks running at priority level 100 will receive the time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec. The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait for a second between read calls will slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service.

Figure 14.10: Illustration of Linux Runqueue and Priority Arrays



In this light, CPU-bound processes basically get any service that is left over when all the I/O bound and interactive processes are blocked. Since Linux (or any other OS) does not know apriori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads. The maximum priority bonus is ~5, since lower priority values correspond to higher priority received by the scheduler. The maximum priority penalty is +5. More specifically, the scheduler maintains a sleep_ avg variable associated with each task. Whenever a task is awaken, this variable is incremented, whenever a task is preempted or its quantum expires, this variable is decremented by the corresponding value. This value is used to dynamically map the task's bonus to values from ~5 to +5. The Linux scheduler recalculates the new priority level as a thread is moved from the active to the expired list. The scheduling algorithm described in this section refers to the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Earlier algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks. Since the description presented in the above

Notes

paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant O(1) time, independent of the number of processes in the system.

In addition, the scheduler includes features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify or modify the affinity requirements of a selected thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

In fact, the kernel code contains synchronization variables in numerous locations. Earlier Linux kernels had just one **Big Kernel Lock (BLK)**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs to execute kernel code concurrently. Hence, many new synchonization points were introduced at much finer granularity.

Self Assessment

Multiple choice questions:

1. A command line interface is called ..... .

   (*a*) Shell (*b*) kernel

   (*c*) Linux (*d*) None of these

2. The forking process is called the ..... .

   (*a*) Memory management (*b*) Parent process

   (*c*) Loading process (*d*) None of these

Fill in the blanks:

3. The function of Linux system is to ..... the hardware.
4. A full desktop environment contains a ..... which controls the placement and appearance of window.
5. To make it easy to specify multiple file names, the shell accepts ..... .

14.3 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS' boot process may rely on access to disks, screens, keyboards, etc. Next, the first sector of the boot disk, the **Master Boot Record (MBR)** is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called *boot* from the boot device, usually an IDE or SCSI disk. The boot program first copies itself to a fixed high memory address to free up low memory for the operating system.

Notes

Once moved, boot reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB** Bootloader**GRand**Unified Other popular bootloaders, such as Intel's LILO, do not rely on any specific filesystem. Instead, they need a block map, and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded. Then it reads in the operating system kernel and jumps to it. At this point, boot has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system. The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system's cockpit flight recorder (the black box investigators look for after a plane crash). Next the kernel data structures are allocated. Most are fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

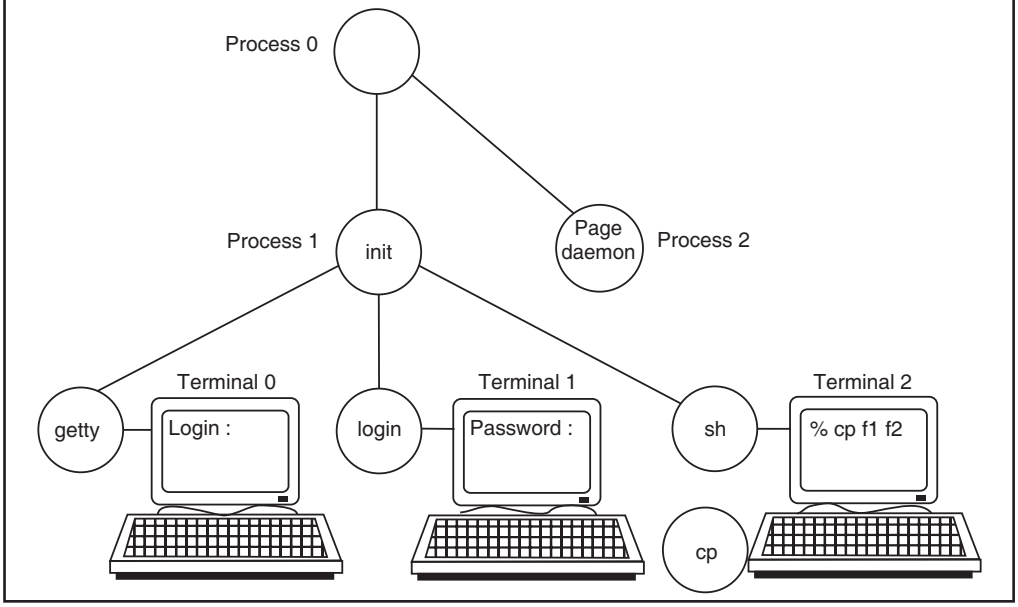
At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux can device drivers do not need to be statically linked and may be loaded dynamically (as can all versions of MS-DOS and Windows, incidentally). The arguments for and against dynamically loading drivers are interesting and worth stating briefly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secured machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are sufficiently rare having to relink the system when a new hardware device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating init (process 1) and the page daemon (process 2). Init checks its flags to see if it is supposed to come up single user or multiuser.

In the former case, it forks off a process that execs the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, /etc/rc, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads /etc/ttys, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then execs a program called getty. Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then types login: on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, getty terminates by executing /bin/login, the login program. Login then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, /etc/passwd. If it is correct, login replaces itself with the user's shell, which then waits for the first command. If it is incorrect, login just asks for another user name. This mechanism is illustrated in Figure 14.11 for a system with three terminals.

Notes

Figure 14.11: The Sequence of Processes Used to Boot Some Linux Systems



In the Figure 14.11, the getty process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so getty has overwritten itself with login, which is asking for the password. A successful login has already occurredon terminal 2, causing the shell to type the prompt (%). The user then typed cp f1 f2 which has caused the shell to fork off a child process and have that process exec the cp program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed cc instead of cp, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compilers passes.

14.4 Memory Management in Linux

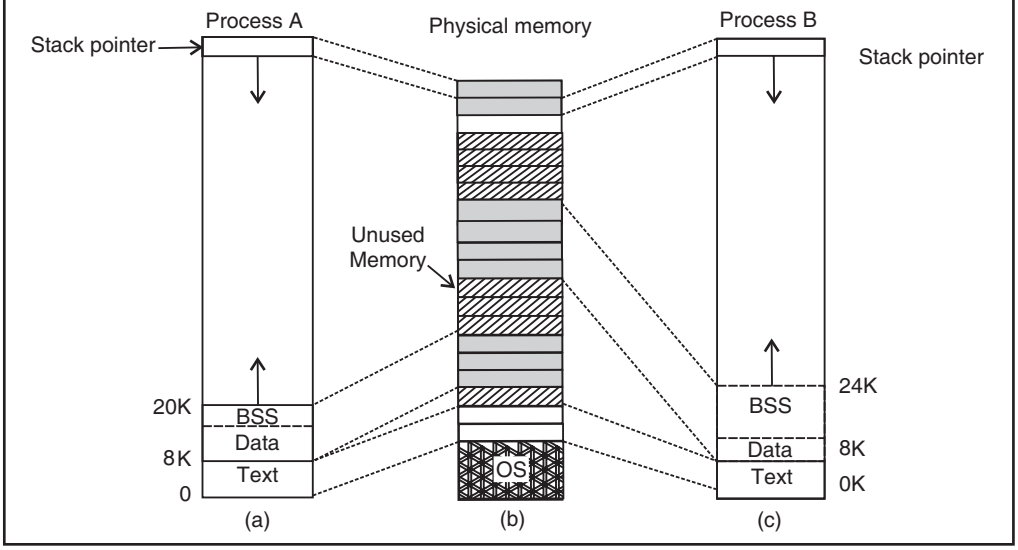
The Linux memory model is straightforward to make programs portable and to make it possible to implement Linux on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to sophisticated paging hardware. This is an area of the design that has been barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

14.4.1 Fundamental Concepts of Memory Management in Linux

Every Linux process has an address space logically consisting of three segments: text, data, and stack. An example process' address space is depicted in Figure 14.12 as process *A*. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self-modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way. The **data segment** contains storage for all the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS** (historically called **Block Started by Symbol**). The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started. For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized data are not all that different from program text both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

Notes

Figure 14.12: (*a*) Process *A*'s Virtual Address Space (*b*) Physical Memory (*c*) Process *B*'s Virtual Address Space



The existence of uninitialized data is actually just an optimization. When a global variable is not explicitly initialized, the semantics of the C language say that its initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0. However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables follows the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is to put a word in the header telling how many bytes to allocate. To make this point more explicit, consider Figure 14.12 again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file. In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static zero page, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy on write mechanism kicks in, and an actual page frame is allocated to the process. Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, brk, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure malloc, commonly used to allocate memory, makes heavy use of this system call. The process address space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called **heap**.

The third segment is the stack segment. On most of the machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32 bit × 86 platforms, the stack starts at address 0 × C0000000, which is the 3 GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault normally occurs, and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

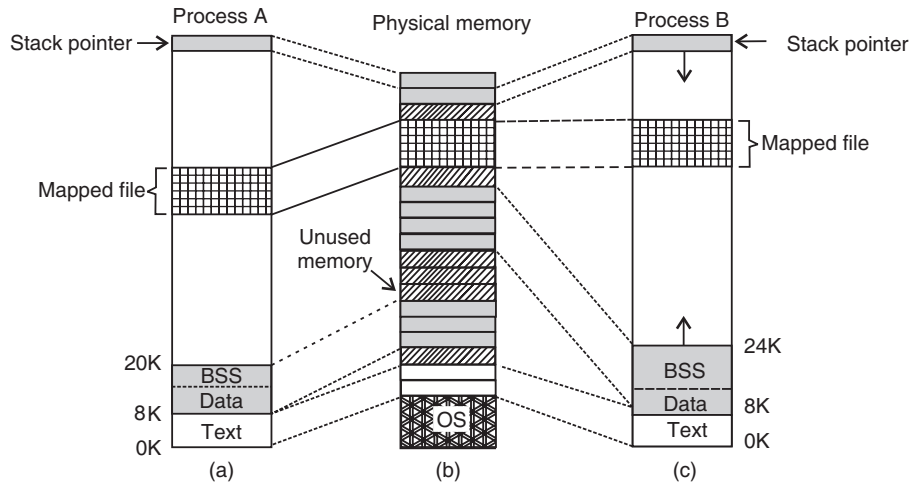
Notes

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way a program can discover its arguments. For example, when the command cp src dest is typed, the cp program is run with the string ''cp src dest'' on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier. When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, most Linux systems support **shared text segments**. In Figure 14.12, we see two processes, *A* and *B*, that have the same text segment. In Figure 14.12, we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual memory hardware. Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 232 bits of address space for instructions and an additional 232 bits of address space for the data and stack segments to share. A jump to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition of dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as read and write. Shared libraries are accessed by mapping them in using this mechanism. In Figure 14.13, we see a file that is mapped into two processes at the same time, at different virtual addresses.

Figure 14.13: Two Processes can Share a Mapped File



We can map in the same file at the same time. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high bandwidth way for multiple processes to share memory. In the most extreme case, two or more processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, making two address spaces exactly correspond is never done, however.

Notes

14.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the malloc library procedure (defined by the ANSI C standard). How malloc is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as **passing the buck**. In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Figure 14.14. Brk specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

Figure 14.14: Some System Calls Relating to Memory Management

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

The return code s is ~1 if an error has occurred; a and addr are memory addresses, len is a length, prot controls protection, flags are miscellaneous bits, fd is a file descriptor, and offset is a file offset. The mmap and munmap system calls control memory-mapped files. The first parameter to mmap, addr, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in a. The second parameter, len, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, prot, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, flags, controls whether the file is private or sharable, and whether addr is a requirement or merely a hint. The fifth parameter, fd, is the file descriptor for the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, offset tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do. The other call, unmap, removes a a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

14.4.3 Implementation of Memory Management in Linux

Each Linux process on a 32 bit machine typically gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The kernel memory typically resides in low physical memory, however it is mapped in the top 1 GB of each process virtual address space, between addresses 0×C0000000 and 0×FFFFFFFF (3-4 GB). The address space is created when the process is created and is overwritten on an exec system call.

In order to allow multiple processes to share the underlying physical memory Linux monitors the use of the physical memory, allocates more memory as needed by user processes or kernel components, dynamically maps portions of the physical memory into the address space of different processes, and dynamically brings in and out of memory program executables, files and other state, necessary to utilize the platform resources efficiently and to ensure execution progress. The remainder of this unit describes the implementation of various mechanisms in the Linux kernel which are responsible for these operations.

Notes

14.4.4 Physical Memory Management

Due to idiosyncratic hardware limitations on many systems, not all physical memory can be treated identically. Linux distinguishes between three memory zones:

1. ZONE_ DMA — pages that can be used for DMA operations.

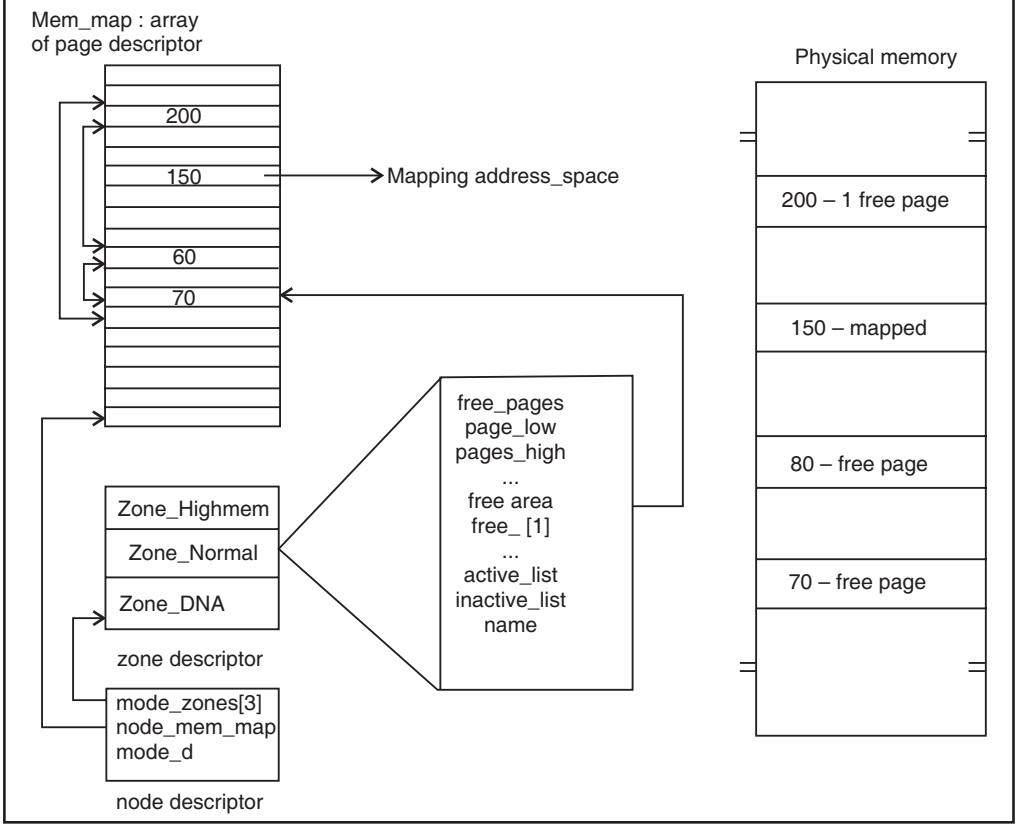
2. ZONE_ NORMAL — normal, regularly-mapped pages.

3. ZONE_ HIGHMEM — pages with high memory addresses, which are not permanently mapped.

The exact boundaries and layout of the memory zones is architecture dependent. On x86 hardware, certain devices can perform DMA operations only in the first 16 MB of address space, hence ZONE_ DMA is in the range 0–16 MB. In addition, the hardware cannot directly map memory addresses above 896 MB, hence ZONE_ HIGHMEM is anything above this mark. ZONE_ NORMAL is anything in between. Therefore, on x86 platforms, the first 896 MB of the Linux address space are directly mapped, whereas the remaining 128 MB of the kernel address space are used to access high memory regions. The kernel maintains a zone structure for each of the three zones, and can perform memory allocations for the three zones separately.

Main memory in Linux consists of three parts. The first two parts, the kernel and memory map, are **pinned** in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page table page, or be on the free list. The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones, free page frame s, etc. The information is organized as follows. First of all, Linux maintains an array of **page descriptors**, of type page for each physical page frame in the system, called mem_ map. Each page descriptor contains a pointer to the address space it belongs to, in case the page is not free, a pair of pointers which allow it to form doubly-linked lists with other descriptors, for instance to keep together all free page frames, and few other fields. In Figure 14.15, the page descriptor for page 150 contains a mapping to the address.

Figure 14.15: Linux Main Memory Representation

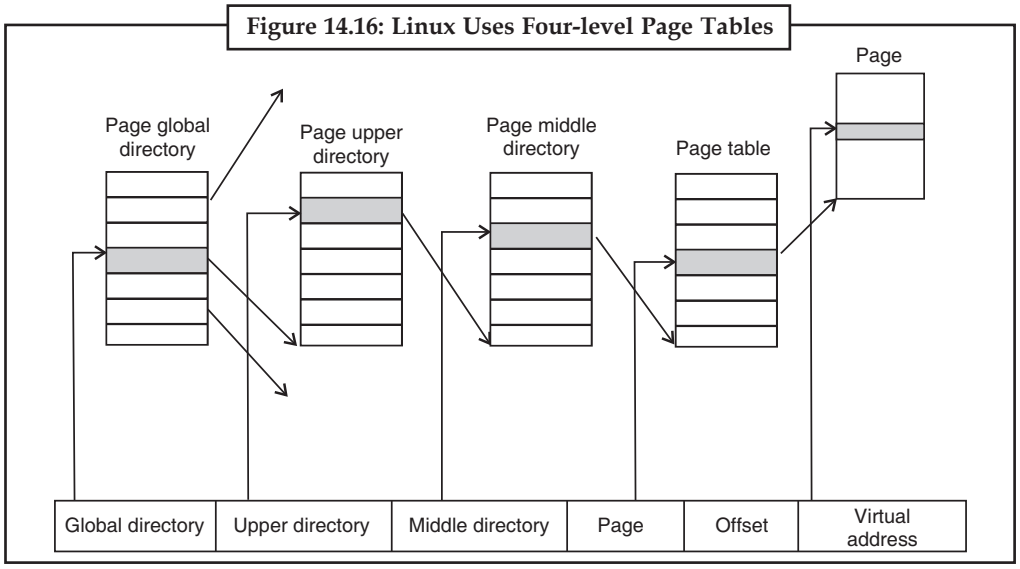


Notes

The size of the page descriptor is 32 bytes, therefore the entire memory map can consume less than 1% of the physical memory (for a page frame of 4 KB). Since the physical memory is divided into zones, for each zone Linux maintains a zone descriptor. The zone descriptor contains information about the memory utilization within each zone, such as number of active or inactive pages, low and high watermarks to be used by the page replacement algorithm described later in this chapter, as well as many other fields.

In addition, a zone descriptor contains an array of free areas. The ith element in this array, identifies the first page descriptor of the first block of 2i free pages. Since there may be multiple blocks of 2i free pages, Linux uses the pair of page descriptor pointers in each page element, to link these together. This information is used in the memory allocation operations supported in Linux. In Figure 14.15 free area[0], which identifies all free areas of main memory consisting of only one page frame (since 20 is one), points to page 70, the first one of the three free areas. The other free blocks of size one can be reached through the links in each.

Figure 14.16: Linux Uses Four-level Page Tables



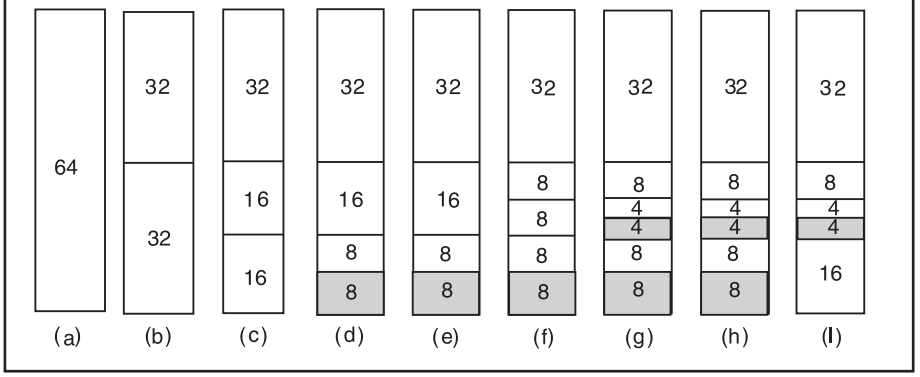
Physical memory is used for various purposes. The kernel itself is fully hardwired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly. In addition, Linux supports dynamically loaded modules, generally device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

14.4.5 Memory Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the so called **buddy algorithm**. The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Figure 14.17: When a request for memory comes in, it is first rounded up to a power of two, say 8 pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

Notes

Figure 14.17: Operation of the Buddy Algorithm



Now suppose that a second request comes in for 8 pages. This can be satisfied hardwired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

Buddy algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk. To alleviate this problem, Linux has a second memory allocation, the **slab allocator**, that takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately. Since the kernel frequently creates and destroy objects of certain type (e.g., task_ struct), it relies on so called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new task_struct it looks in the object cache for task structures, and first tries to find a partially full slab, and allocate a new task_ struct object there. If no such slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task structure object cache. The kmalloc kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here. A third memory allocator, vmalloc, is also available and is used when the requested memory need only be contiguous in virtual space, but not in physical memory. In practice, this is true for most of the requested memory. One exception is devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of vmalloc results in some performance degradation, and is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Notes

14.4.6 Virtual Address Space Representation

The virtual address space is divided into homogeneous, contiguous, page aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas. There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, which supports page frames of 4 MB, Linux can support jumbo page frames of 4 MB each. In addition, in a **PAE (Physical Address Extension)** mode, which is used on certain 32 bit architecture to increase the process address space beyond 4 GB, page sizes of 2 MB are supported. Each area is described in the kernel by a vm_ area_ struct entry. All the vm_ area_ structs for a process are linked together in a list sorted on virtual address so all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching. The vm_ area_ struct entry lists the area's properties. These include the protection mode (e.g., read only or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The vm_ area_ struct also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.

The vm_ area_ struct also records whether the area has backing storage on disk assigned, and if so, where. Text segments use the executable binary as backing storage and memory-mapped files use the disk file as backing storage. Other areas, such as the stack, do not have backing storage assigned until they have to be paged out.

A top-level memory descriptor, mm_ struct, gathers information about all virtual memory areas belonging to an address space, information about the different segments — text, data, stack, about users sharing this address space, etc. All vm_ area_ struct elements of an address space can be accesses through its memory descriptor in two ways. First, they are organized in a linked lists, ordered by virtual memory addresses. This way is useful when all virtual memory areas need to be accessed, or when the kernel is searching to allocated a virtual memory region of a specific size. In addition, the vm_ area_ struct entries are organized in a binary ''red-black'' tree, a data structure optimized for fast lookups. This method is used when a specific virtual memory needs to be accessed. By enabling access to elements of the process address space via these two methods, Linux uses more state per process but allows different kernel operations to use the access method which is more efficient for the task at hand.

14.4.7 Paging in Linux

Early UNIX systems relied on a **swapper** process to move entire processes between memory and disk, whenever not all active processes could fit in the physical memory. Linux, as well as other modern UNIX versions, no longer move entire processes. The main memory management unit is a **page** and almost all memory management components, operate on a page granularity. The swapping subsystem also operates on page granularity and is tightly coupled with the **Page Frame Reclaiming Algorithm**, described later in this section.

The basic idea behind paging in Linux is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are

Notes

swapped in, the process is deemed ''in memory'' and can be scheduled to run. The pages of the text, data, and stack segments are brought in dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in. Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the idle process traditionally called the swapper and process 1 is in it, as shown in Figure 14.12).

Like all daemons, the page daemon is started up periodically so it can look around to see if there is any work for it to do. If it discovers that the number of pages on the list of free memory pages is too low, it initiates action to free up more pages. Linux is a demand-paged system with no prepaging and no working set concept (although there is a system call in which a user can give a hint that a certain page may be needed soon, in the hopes it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition (if present) or one of the fixed-length paging files, called the **swap area**. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The page table is updated to reflect that the page is no longer present in memory (e.g., the page-not-present bit is set) and the disk location is written into the page table entry.

14.4.8 Page Replacement Algorithm

Page replacement works as follows. Linux tries to keep some pages free so they can be claimed as needed. Of course, this pool must be continually replenished, so the **PFRA (Page Frame Reclaiming Algorithm)** algorithm is how this happens.

First of all, Linux distinguishes between four different types of pages: unreclaimable, swappable, syncable, and discardable. Unreclaimable pages, which include reserved or locked pages, kernel mode stacks, etc., may not be paged out. Swappable pages must be written back to the swap area or the paging disk partition before the page can be reclaimed. Syncable pages must be written back to disk if they have been marked as dirty. Finally, discardable pages can be reclaimed immediately.

At boot time, *init* starts up a page daemon, *kswapd*, one per each memory node, and configures them to run periodically. Each time *kswapd* awakens, it checks to see if there are enough free pages available, by comparing the low and high watermarks with the current memory usage for each memory zone. If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed. If the available memory for any of the zones falls below a threshold, *kswapd* initiates the page frame reclaiming algorithm. During each run, only a certain target number of pages is reclaimed, typically 32. This number is limited to control the I/O pressure (the number of disk writes, created during the PFRA operations). Both, the number of reclaimed pages and the total number of scanned pages are configurable parameters.

Each time PFRA executes, it first tries to reclaim easy pages, then proceeds with the harder ones. Discardable and unreferenced pages can be reclaimed immediately by moving them onto the zone's freelist. Next, it looks for pages with backing store which have not been referenced recently, using a clock-like algorithm. Following are shared pages that none of the users seems

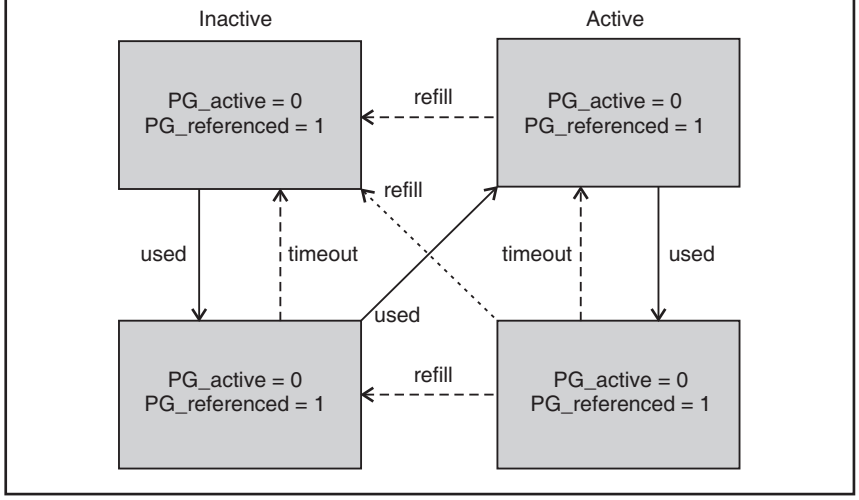
Notes

to be using much. The challenge with shared pages is that, if a page entry is reclaimed, the page tables of all address spaces originally sharing that page must be updated in a synchronous manner. Linux maintains efficient tree like data structures to easily find all users of a shared page. Ordinary user pages are searched next, and if chosen to be evicted, they must be scheduled for write in the swap area. The **swappiness** of the system, that is, the ratio of pages with backing store versus pages which need to be swapped out selected during PFRA, is a tunable parameter of the algorithm.

Finally, if a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped. PFRA uses a clock-like algorithm to select old pages for eviction within a certain category. At the core of this algorithm is a loop which scans through each zone's active and inactive lists, trying to reclaim different kinds of pages, with different urgency. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up.

During PFRA, pages are moved between the active and inactive list in a manner described in Figure 14.18 maintain try to find pages which have not been referenced and are unlikely to be needed in the near future, PFRA maintains two flags per page—active/inactive, and referenced or not. These two flags encode four states, as shown in Figure 14.18 during first set of pages, PFRA first clears their reference bits. If during the second run over the page it is determined that it has been referenced, it is advanced to another state, from which it is less likely to be reclaimed. Otherwise, the page is moved to a state from where it will more likely to be evicted. Pages on the inactive list, which have not been referenced since the last time they were inspected, are best candidates for eviction. These correspond to pages with both PG active and PG referenced equal to zero in Figure 14.18 ever necessary, pages may be reclaimed even if they are in some of the other states. The refill arrows in Figure 14.18 considered in the page frame replacement algorithm.

Figure 14.18: Refill Arrows in Page Frame Replacement Algorithm



The reason PRFA maintains pages in the inactive list although they might have referenced is to prevent following situations. Consider a process which makes periodic accesses to different pages, with a 1-hour period. A page accessed since the last loop will have its reference flag set. However, since it will not be needed again for the next hour, there is no reason not to consider it as a candidate for reclamation.

One other aspect of the memory management system that we have not yet mentioned is a second daemon, pdflush, actually a set of background daemon threads. The pdflush threads either

Notes

(1) wake up periodically, typically each 500 msec, to write back to disk very old dirty pages, or (2) are explicitly awakened by the kernel when available memory levels fall below a certain threshold, to write back dirty pages from the page cache to disk. In **laptop mode**, in order to conserve battery life, dirty pages are written to disk whenever pdflush threads wakeup. Dirty pages may also be written out to disk on explicit requests for synchronization, via systems calls such as sync, orfsync, fdatasync. Older Linux versions used two separate daemons: kupdate, for old page write back, and bdflush, for page write back under low memory conditions. In the 2.4 kernel this functionality was integrated in the pdflush threads. The choice of multiple threads was made in order to hide long disk latencies.

14.5 Input/Output in Linux

The I/O system in Linux is fairly straightforward. Basically, all I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

14.5.1 Fundamental Concepts of Input/Output in Linux

Like all computers, those running Linux have I/O devices such as disks, printers, and networks connected to them. Some way is needed to allow programs to access these devices. Although various solutions are possible, the Linux one is to integrate the devices into the file system as what are called **special files**. Each I/O device is assigned a path name, usually in /dev. For example, a disk might be /dev/hd1, a printer might be /dev/lp, and the network might be /dev/net. These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual open, read, and write system calls will do just fine. For example, the command cp file */dev/lp* copies the *file* to printer, causing it to be printed (assuming that the user has permission to access */dev/lp*). Programs can open, read, and write special files the same way as they do regular files. In fact, *cp* in the above example is not even aware that it is printing. In this way, no special mechanism is needed for doing Special files are divided into two categories—block and character. A **block special file** is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. In other words, a program can open a block special file and read, say, block 124 without first having to read blocks 0 to 123. Block special files are typically used for disks.

Character special files are normally used for devices that input or output a character stream. Keyboards, printers, networks, mice, plotters, and most other I/O devices that accept or produce data for people use character special files. It is not possible (or even meaningful) to seek to block 124 on a mouse. Associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a **major device number** that serves to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk have a **minor device number** that identifies it. Together, the major and minor device numbers uniquely specify every I/O device. In few cases, a single driver handles two closely related devices. For example, the driver corresponding to */dev/tty* controls both the keyboard and the screen, which is often thought of as a single device, the terminal.

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. For example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some key. Some user prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of e-mail (which uses @ within e-mail address), many systems

Notes

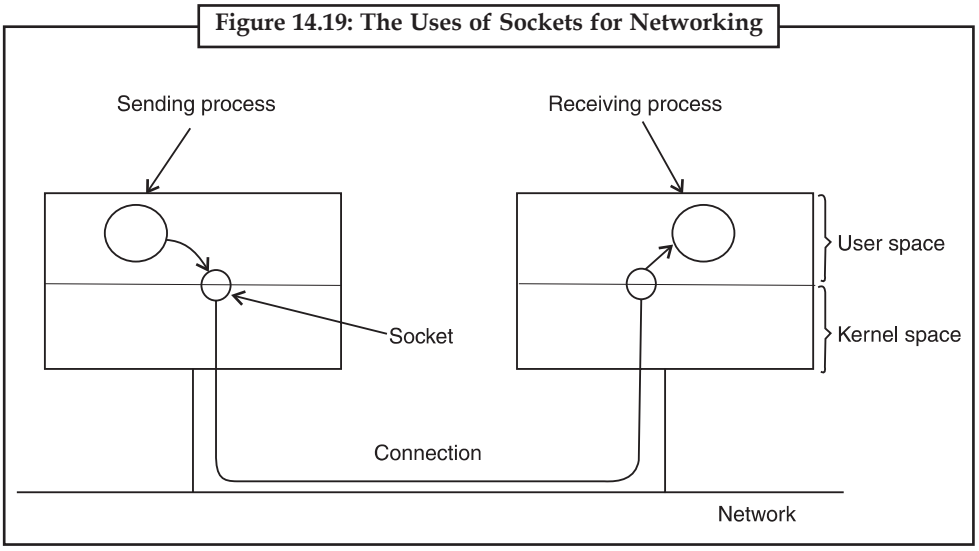
have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here, too, different users have different preferences. CTRL-C is a common choice, but it is not universal.

Rather than making a choice and forcing everyone to use it, Linux allows all these special functions and many others to be customized by the user. A special system call is generally provided for setting these options. This system call also handles tab expansion, enabling and disabling of character echoing, conversion between carriage return and line feed, and similar items. The system call is not permitted on regular files or block special files.

14.5.2 Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX and taken over by Linux more-or-less verbatim. The key concept in the Berkeley design is the **socket**. Sockets are analogous to mailboxes and telephone wall sockets in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and telephone wall sockets allow them to plug in telephones and connect to the telephone system. The sockets' position is shown in Figure 14.19.

Figure 14.19: The Uses of Sockets for Networking



Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing a connection, reading data, writing data, and releasing the connection. Each socket supports a particular type of networking, specified when the socket is created. The most common types are:

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

The first socket type allows two processes on different machines to establish the equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent arrive and in the same order they were sent. The second type is similar to the first one, except that it preserves packet boundaries. If the sender makes five separate calls to write, each for 512 bytes, and the receiver asks for 2560 bytes, with a type 1 socket, all 2560 bytes will be returned at once. With a type 2 socket, only 512 bytes will be returned. Four more calls are needed to get the rest. The third type of socket is used to give the user access to the raw network. This type is especially useful for real-time applications, and for those situations in which the user wants to implement a specialized error handling scheme. Packets may be lost or reordered by the network. There are no guarantees, as in the first two cases. The advantage of this mode is higher performance, which sometimes

Notes

outweighs reliability (e.g., for multimedia delivery, in which being fast counts for more than being right). When a socket is created, one of the parameters specifies the protocol to be used for it. For reliable byte streams, the most popular protocol is **TCP (Transmission Control Protocol)**. For unreliable packet-oriented transmission, **UDP (User Datagram Protocol)** is the usual choice. Both are these are layered on top of **IP (Internet Protocol)**. All of these protocols originated with the U.S. Dept. of Defense's ARPANET, and now form the basis of the Internet. There is no common protocol for reliable packet streams. Before a socket can be used for networking, it must have an address bound to it. This address can be in one of several naming domains. The most common domain is the Internet naming domain, which uses 32-bit integers for naming endpoints in Version 4 and 128-bit integers in Version 6 (Version 5 was an experimental system that never made it to the major leagues).

Once sockets have been created on both the source and destination computers, a connection can be established between them (for connection-oriented communication). One party makes a listen system call on a local socket, which creates a buffer and blocks until data arrive. The other one makes a connect system call, giving as parameters the file descriptor for a local socket and the address of a remote socket. If the remote party accepts the call, the system then establishes a connection between the sockets. Once a connection has been established, it functions analogously to a pipe. A process can read and write from it using the file descriptor for its local socket. When the connection is no longer needed, it can be closed in the usual way, via the close system call.

14.5.3 Input/Output System Calls in Linux

Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file, eliminating the need for special system calls. Nevertheless, sometimes there is a need for something that is device specific. Prior to POSIX most UNIX systems had a system call ioctl that performed a large number of device-specific actions on special files. Over the course of the years, it had gotten to be quite a mess. POSIX cleaned it up by splitting its functions into separate function calls primarily for terminal devices. In Linux, and modern UNIX systems in general, whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four listed in Figure 14.20 are used to set and get the terminal speed. Different calls are provided for input and output because some modems operate at split speed. For example, old videotex systems allowed people to access public databases with short requests from the home to the server at 75 bits/sec with replies coming back at 1200 bits/sec. This standard was adopted at a time when 1200 bits/sec both ways was too expensive for home use. Time has changed in the networking world. This asymmetry still persists, with some telephone companies offering inbound service at 8 Mbps and outbound service at 512 Kbps, often under the name of **ADSL (Asymmetric Digital Subscriber Line)**.

Figure 14.20: The Main POSIX Calls for Managing the Terminal

Function call	Description
s = cfsetospeed(& termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(& termios, speed)	Get the output speed
s = cfgtetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, & termios)	Set the attributes
s = tcgetattr(fd, & termios)	Get the attributes

Notes

The last two calls in the list are for setting and reading back all the special characters used for erasing characters and lines, interrupting processes, and so on. In addition, they enable and disable echoing, handle flow control, and other related functions. Additional I/O function calls also exist, but they are somewhat specialized so we will not discuss them further. In addition, ioctl is still available.

14.5.4 Implementation of Input/Output in Linux

I/O in Linux is implemented by a collection of device drivers, one per device type. The function of the drivers is to isolate the rest of the system from the idiosyncracies of the hardware. By providing standard interfaces between the drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel. When the user accesses a special file, the file system determines the major and minor device numbers belonging to it, whether it is a block special file or a character special file. The major device number is used to index into one of two internal hash tables containing data structures for character or block devices. Thus the located structure contains pointer to the procedures to call to open the device, read the device, write the device, and so on. The minor device number is passed as a parameter. Adding a new device type to Linux means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

Some of the operations which may be associated with different character devices are shown in Figure 14.21. Each row refers to a single I/O device (i.e., a single driver). The columns represent the functions that all character drivers must support. Several other functions also exist. When an operation is performed on a character special file, the system indexes into hash table of character devices to select the proper structure, then calls the corresponding function to have the work performed. Thus each of the file operation contains a pointer to a function contained in the corresponding driver.

Figure 14.21: Some of the File Operations Supported for Typical Character Devices

Device	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	nul	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_iocti	...
Printer	lp_open	lp_close	error	lp_write	lp_iocti	...

Each driver is split into two parts, both of which are part of the Linux kernel that run in kernel mode. The top half runs in the context of the caller and interfaces to the rest of Linux. The bottom half runs in kernel context and interacts with the device. Drivers are allowed to make calls to kernel procedures for memory allocation, timer management, DMA control, and other things. The set of kernel functions that may be called is defined in a document called the **Driver-Kernel Interface**. Writing device drivers for Linux is covered in detail in (Egan and Teixeira, 1992; Rubini and Corbert, 2005).

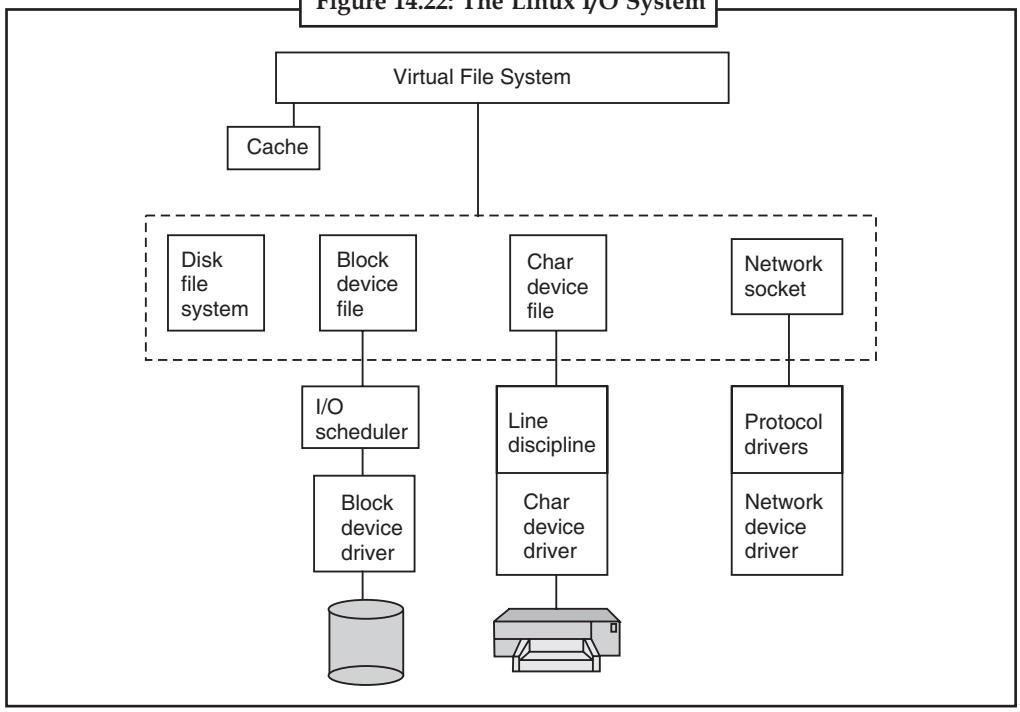
The I/O system is split into two major components: the handling of block special files and the handling of character special files. We will now look at each of these components in turn. The goal of the part of the system that does I/O on block special files (e.g., disks) is to minimize the number of actual transfers that must be done. To accomplish this goal, Linux systems have a **cache** between the disk drivers and the file system, as illustrated in Figure 14.22. Prior to

Notes

the 2.2 kernel, Linux maintained completely separate page and buffer caches, so a file residing in a disk block could be cached in both caches. Newer versions of Linux have a unified cache. A generic block layer holds these components together, and performs the necessary translations between disk sectors, blocks, buffers and pages of data, and enables the operations on them.

The cache is a table in the kernel for holding thousands of the most recently used blocks. When a block is needed from a disk for any purpose (i-node, directory, or data), a check is first made to see that it is in the cache. If so, it is taken from there and a disk access is avoided, thereby resulting in great improvements in system performance. If the block is not in the page cache, it is read from the disk into the cache and from there, copied to where it is needed. Since the page cache has room for only a fixed number of blocks, the page replacement algorithm described in the previous section is invoked. The page cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk.

Figure 14.22: The Linux I/O System



In addition to avoid having blocks stay too long in the cache before being written to the disk, all the dirty blocks are written to the disk every 30 seconds. In order to minimize the latency of repetitive disk head movements, Linux relies on an **I/O scheduler**. The purpose of the I/O scheduler is to reorder or bundle read/write requests to block devices. There are many scheduler variants, optimized for different types of workloads. The basic Linux scheduler is based on the original **Linus Elevator scheduler**. The operations of the elevator scheduler can be summarized as follows—disk operations are sorted in a doubly linked list, ordered by the address of the sector of the disk request. New requests are inserted in this list in a sorted manner. This prevents repeated costly disk head movements.

The request list is than *merged* so that adjacent operations are issued via a single disk request. The basic elevator scheduler can lead to starvation. Therefore, the revised version of the Linux disk scheduler includes two additional lists, maintaining read or write operations ordered by their deadline. The default deadlines are 0.5 sec for read requests and 5 sec for write requests. If a system defined deadline for the oldest write operation is about to expire, that write request will be serviced before any of the requests on the main doubly linked list. The interaction with character devices is much simpler. Since character devices produce or consume streams of

Notes

characters, or bytes of data, support for random access makes little sense. One exception is the use of **line disciplines**. A line discipline can be associated with a terminal device, represented via the structure *tty_ struct*, and it represents an interpreter for the data exchanged with the terminal device. For instance, local line editing can be done (i.e., erased characters and lines can be removed), carriage returns can be mapped onto line feeds, and other special processing can be completed. However, if a process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed. Output works in a similar way, expanding tabs to spaces, converting line feeds to carriage returns + line feeds, adding filler characters following carriage returns on slow mechanical terminals, and so on. Like input, output can go through the line discipline (cooked mode) or bypass it (raw mode). Raw mode is especially useful when sending binary data to other computers over a serial line and for GUIs. Here, no conversions are desired.

The interaction with **network devices** is somewhat different. While network devices also produce/consume streams of characters, their asynchronous nature makes them less suitable for easy integration under the same interface as other character devices. The networking device driver produces packets consisting of multiple bytes of data, along with network headers. These packets are then routed through a series of network protocol drivers, and ultimately are passed to the user space application. A key data structure is the socket buffer structure, skbuff, which is used to represent portions of memory filled with packet data. The data in an skbuff buffer does not always start at the start of buffer. As it is being processed by various protocols in the networking stack, protocol headers may be removed, or added. The user processes interact with networking devices via sockets, which in Linux support the original BSD socket API. The protocol drivers can be bypassed and direct access to the underlying network device is enabled via raw sockets. Only superusers are allowed to create raw sockets.

14.5.5 Modules in Linux

For decades, UNIX device drivers have been statically linked into the kernel so they were all present in memory when the system was booted every time. Given the environment in which UNIX grew up, mostly departmental minicomputers and then high-end workstations, with their small and unchanging sets of I/O devices, this scheme worked well. Basically, a computer center built a kernel containing drivers for the I/O devices and that was it. If next year it bought a new disk, it relinked the kernel. With the arrival of Linux on the PC platform, suddenly all that changed. The number of I/O devices available on the PC is orders of magnitude larger than on any minicomputer. In addition, although all Linux users have (or can easily get) the full source code, probably the vast majority would have considerable difficulty adding a driver, updating the all device-driver related data structures, relinking the kernel, and then installing it as the bootable system (not to mention dealing with the aftermath of building a kernel that does not boot).

Linux solved this problem with the concept of **loadable modules**. These are chunks of code that can be loaded into the kernel while the system is running. Most commonly these are character or block device drivers, but they can also be entire file systems, network protocols, performance monitoring tools, or anything else desired. When a module is loaded, several things have to happen. First, the module has to be relocated on-the-fly, during loading. Second, the system has to check to see if the resources the driver needs are available (e.g., interrupt request levels) and if so, mark them as in use. Third, any interrupt vectors that are needed must be set up. Fourth, the appropriate driver switch table has to be updated to handle the new major device type. Finally, the driver is allowed to run to perform any device-specific initialization it may need. Once all these steps are completed, the driver is fully installed, the same as any statically installed driver. Some modern UNIX systems also support loadable modules now, too.

14.6 Linux File System

The most visible part of any operating system, including Linux, is the file system. In the following sections we will examine the basic ideas behind the Linux file system, the system calls, and how

Notes

the file system is implemented. Some of these ideas derive from MULTICS, and many of them have been copied by MSDOS, Windows, and other systems, but others are unique to UNIX-based systems. With minimal mechanism and a very limited number of system calls, Linux nevertheless provides a powerful and elegant file system.

14.6.1 Fundamental Concepts of Linux File System

The initial Linux file system was the MINIX 1 file system. However, due to the fact that it limited file names to 14 characters (in order to be compatible with UNIX Version 7) and its maximum file size was 64 MB (which was overkill on the 10 MB hard disks of its era), there was interest in better file systems almost from the beginning of the Linux development, which began about 5 years after MINIX 1 was released. The first improvement was the ext file system, which allowed file names of 255 characters and files of 2 GB, but it was slower than the MINIX 1 file system, so the search continued for a while. Eventually, the ext2 file system was invented with long file names, long files, and better performance, and that has become the main file system. However, Linux supports several dozens of file systems using the Virtual File System (VFS) layer (described in the next section).

When Linux is linked, a choice is offered of which file systems should be built into the kernel. Other ones can be dynamically loaded as modules during execution, if need be. A Linux file is a sequence of 0 or more bytes containing arbitrary information. No distinction is made between ASCII files, binary files, or any other kinds of files. The meaning of the bits in a file is entirely up to the file's owner. The system does not care. File names are limited to 255 characters, and all the ASCII characters except NUL are allowed in file names, so a file name consisting of three carriage returns is a legal file name (but not an especially convenient one). By convention, many programs expect file names to consist of a base name and an extension, separated by a dot (which counts as a character). Thus prog.c is typically a C program, prog.f90 is typically a FORTRAN 90 program, and prog.o is usually an object file (compiler output). These conventions are not enforced by the operating system but some compilers and other programs expect them. Extensions may be of any length and files may have multiple extensions, as in prog.java.gz, which is probably a gzip compressed Java program.

Files can be grouped together in directories for convenience. Directories are stored as files, and to a large extent can be treated like files. Directories can contain subdirectories, leading to a hierarchical file system. The root directory is called / and usually contains several subdirectories. The / character is also used to separate directory names, so that the name /usr/ast/x denotes the file x located in the directory ast, which itself is in the */usr* directory. Some of the major directories near the top of the tree are shown in Figure 14.23. There are two ways to specify file names in Linux, both to the shell and when opening a file from within a program. The first way is using an **absolute path**, which means telling how to get to the file starting at the root directory. An example of an absolute path is /usr/ast/books/mos3/chap-10. This tells the system to look in the root directory for a directory called usr, then look there for another directory, ast. In turn, this directory contains a directory books.

Figure 14.23: Some Important Directories Found in Most Linux Systems

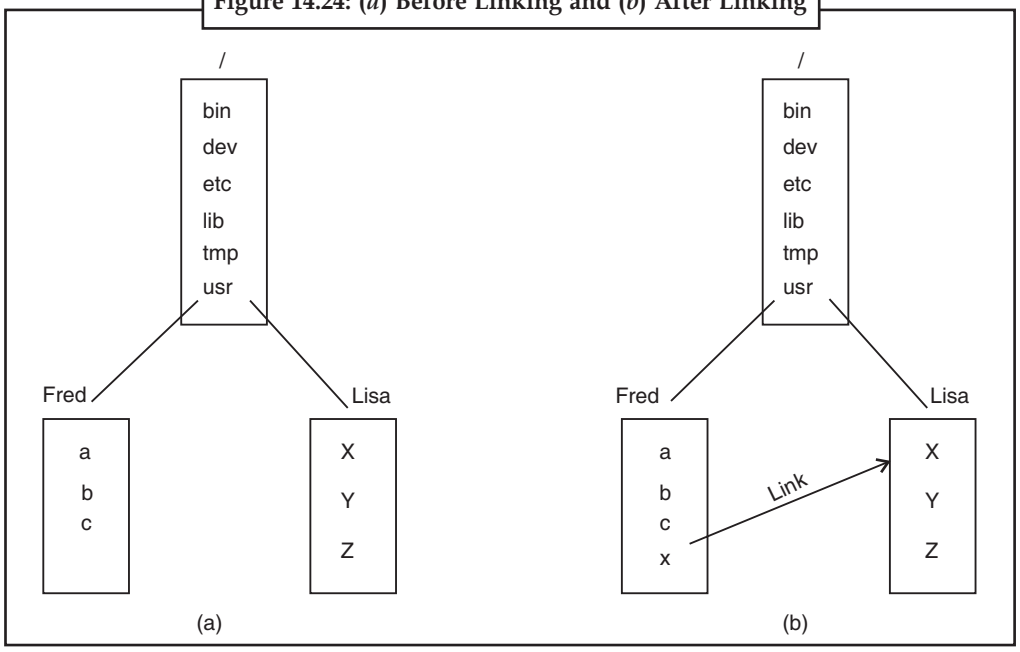
Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Notes

Absolute path names are often long and inconvenient. For this reason, Linux allows users and processes to designate the directory in which they are currently working as the **working directory**. Path names can also be specified relative to the working directory. A path name specified relative to the working directory is a relative path. For example, if */usr/ast/books/mos3* is the working directory, then the shell command cp chap-10 backup-10 has exactly the same effect as the longer command cp /usr/ast/books/mos3/chap-10 / usr/ast/books/mos3/backup-10. It frequently occurs that a user needs to refer to a file that belongs to another user, or at least is located elsewhere in the file tree. For example, if two users are sharing a file, it will be located in a directory belonging to one of them, so the other will have to use an absolute path name to refer to it (or change the working directory). If this is long enough, it may become irritating to have to keep typing it. Linux provides a solution to this problem by allowing users to make a new directory entry that points to an existing file. Such an entry is called a **link**. Consider this situation with an example shown in Figure 14.24. Fred and Lisa are working together on a project, and each one needs frequent access to the other's files. If Fred has */usr/fred* as his working directory, he can refer to the file x in Lisa's directory as */usr/lisa/x*. Alternatively, Fred can create a new entry in his directory as shown in Figure 14.24, after which he can use *x* to mean */usr/lisa/x*.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa's file x was using its absolute path. Actually, this is not really true. When a directory is created, two entries and .., are automatically made in it. The former refers to the working directory itself. The latter refers to the directory's parent, that is, the directory in which it itself is listed. Thus from */usr/fred*, another path to Lisa's file *x* is *../lisa/x*. In addition to regular files, Linux also supports character special files and block special files. Character special files are used to model serial I/O devices such as keyboards and printers. Opening and reading from */dev/tty* reads from the keyboard; opening and writing to */dev/lp* writes to the printer.

Figure 14.24: (*a*) Before Linking and (*b*) After Linking



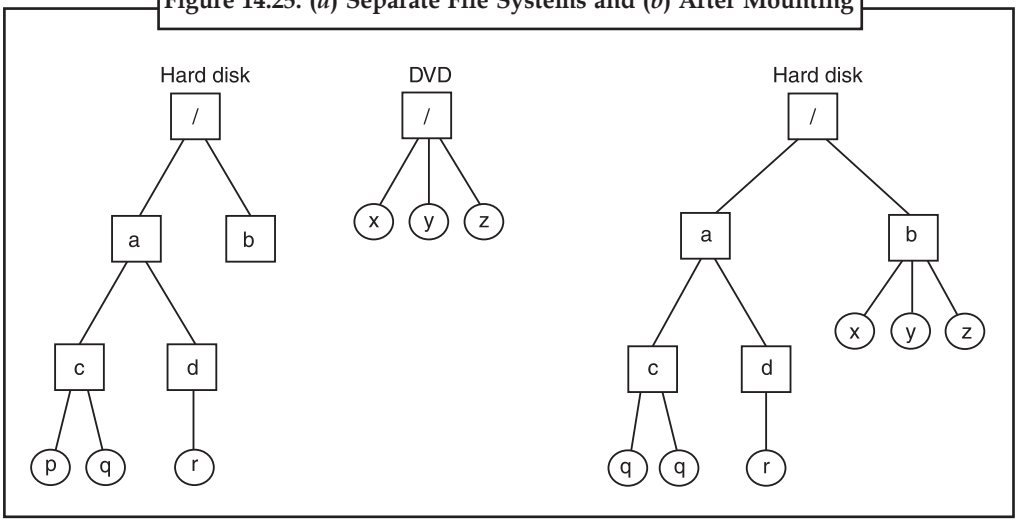
Block special files, often with names like */dev/hd1*, can be used to read and write raw disk partitions without regard to the file system. Thus a seek to byte k followed by a read will begin

Notes

reading from the k-th byte on the corresponding partition, completely ignoring the i-node and file structure. Raw block devices are used for paging and swapping, by programs that lay down file systems (e.g., *mkfs*), and by programs that fix sick file systems (e.g., *fsck*), for example. Many computers have two or more disks. On mainframes at banks, it is frequently necessary to have 100 or more disks on a single machine, in order to hold the huge databases required. Even personal computers normally have at least two disks—a hard disk and an optical (e.g., DVD) drive. When these are multiple disk drives, the question arises of how to handle them. One solution is to put a self-contained file system on each one and just keep them separate. Consider, the situation depicted in Figure 14.25.

Here we have a hard disk, which we will call *C*:, and a DVD, which we will call *D*, each has its own root directory and files. With this solution, the user has to specify both the device and the file when anything other than the default is needed. For example, to copy the file *x* to the directory *d*, (assuming *C*: is the default), one would type cp D:/x /a/d/x. This is the approach taken by systems like MS-DOS, Windows 98, and VMS. The Linux solution is to allow one disk to be mounted in another disk's file tree. In this example, we could mount the DVD on the directory */b*, yielding the file system of Figure 14.25. The user now sees a single file tree, and no longer has to be aware of which file resides on which device.

Figure 14.25: (*a*) Separate File Systems and (*b*) After Mounting



Another interesting property of the Linux file system is **locking**. In some applications, two or more processes may be using the same file at the same time, which may lead to race conditions. One solution is to program the application with critical regions. However, if the processes belong to independent users who do not even know each other, this kind of coordination is generally inconvenient. Consider, for example, a database consisting of many files in one or more directories that are accessed by unrelated users. It is certainly possible to associate a semaphore with each directory or file and achieve mutual exclusion by having processes do a down operation on the appropriate semaphore before accessing the data. The disadvantage, however, is that a whole directory or file is then made inaccessible, even though only one record may be needed.

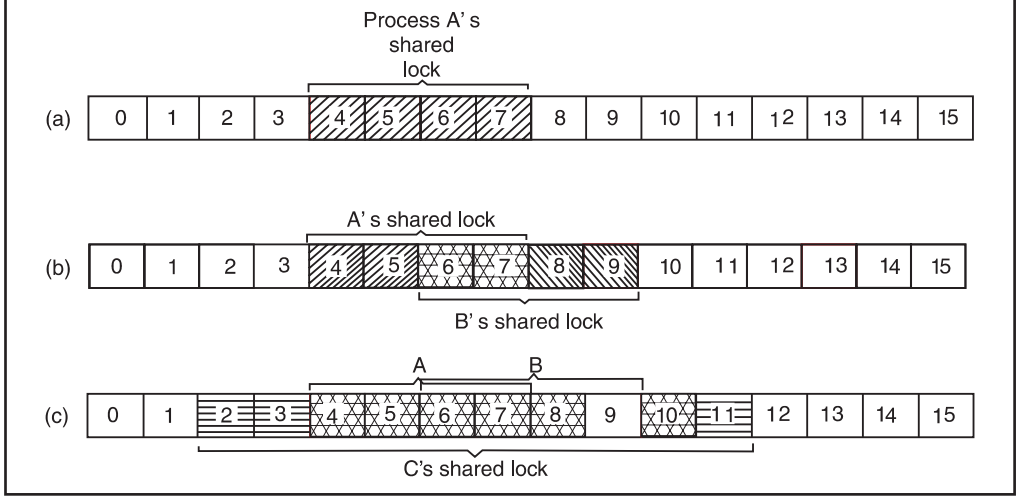
For this reason, POSIX provides a flexible and fine-grained mechanism for processes to lock as little as a single byte and as much as an entire file in one indivisible operation. The locking mechanism requires the caller to specify the file to be locked, the starting byte, and the number of bytes. If the operation succeeds, the system makes a table entry noting that the bytes in question (e.g., a database record) are locked. Two kinds of locks are provided, **shared locks** and **exclusive locks**. If a portion of a file already contains a shared lock, a second attempt to place a shared lock on it is permitted, but an attempt to put an exclusive lock on it will fail. If a portion of a file contains an exclusive lock, all attempts to lock any part of that portion will fail until the lock has been released. In order to successfully place a lock, every byte in the region to be locked must be available.

Notes

When placing a lock, a process must specify whether it wants to block or not in the event that the lock cannot be placed. If it chooses to block, when the existing lock has been removed, the process is unblocked and the lock is placed. If the process chooses not to block when it cannot place a lock, the system call returns immediately, with the status code telling whether the lock succeeded or not.

Locked regions may overlap. In Figure 14.26, we see that process A has placed a shared lock on bytes 4 through 7 of same file. Later, process B places a shared lock on bytes 6 through 9. Finally, C locks bytes 2 through 11. As long as all these locks are shared, they can co-exist.

Figure 14.26: (*a*) A File with One Lock, (*b*) Addition of a Second Lock and (*c*) A Third Lock



Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Figure 14.26(c), with a request to block if the lock fails. Since two previous locks cover this block, the caller will block and will remain blocked until both B and C release their locks.

14.6.2 File System Calls in Linux

Many system calls relate to files and the file system. First we will look at the system calls that operate on individual files. Later we will examine those that involve directories or the file system as a whole. To create a new file, the create call can be used. The parameters provide the name of the file and the protection mode. Thus

fd = creat("abc", mode);

creates a file called abc with the protection bits taken from mode. These bits determine which users may access the file and how. That will be described later. The create call not only creates a new file, but also opens it for writing. To allow subsequent system calls to access the file, a successful create returns as its result a small non-negative integer called a **file descriptor**, fd in the example above. If a create is done on an existing file, that file is truncated to length 0 and its contents are discarded. Files can also be created using the open call with appropriate arguments.

Now let us continue looking at the principal file system calls, which are listed in Figure 14.27. To read or write an existing file, the file must first be opened using open. This call specifies the file name to be opened and how it is to be opened: for reading, writing, or both. Various options can be specified as well. Like create, the call to open returns a file descriptor that can be used for reading or writing.

Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent create or open. Both the create and open calls always return the lowest numbered file descriptor not currently in use. When a program starts executing in the standard way, file

Notes

descriptors 0, 1, and 2 are already opened for standard input, standard output, and standard error, respectively. In this way, a filter, such as the sort program, can just read its input from file descriptor 0 and write its output to file descriptor 1, without having to know what files they are. This mechanism works because the shell arranges for these values to refer to the correct (redirected) files before the program is started. The most heavily used calls are undoubtedly read and write. Each one has three parameters: a file descriptor (telling which open file to read or write), a buffer address (telling where to put the data or get the data from), and a count (telling how many bytes to transfer). A typical call is

n = read(fd, buffer, nbytes);

Although nearly all programs read and write files sequentially, some programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (writing) sequentially, it normally points to the next byte to be read (written). If the pointer is at, say, 4096, before 1024 bytes are read, it will automatically be moved to 5120 after a successful read system call. The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file, or even beyond the end of it. It is called lseek to avoid conflicting with seek, a now-obsolete call that was formerly used on 16 bit computers for seeking.

Figure 14.27: Some System Calls Relating to Files

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcnt(fd, cms, ...)	File locking and other operations

The return code s is ~1 if an error has occurred; fd is a file descriptor, and position is a file offset. The parameters should be self-explanatory. Lseek has three parameters: the first one is the file descriptor for the file; the second one is a file position; the third one tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file after the file pointer was changed. Slightly ironically, lseek is the only file system calls that can never cause an actual disk seek because all it does is update the current file position, which is a number in memory.

For each file, Linux keeps track of the file mode (regular, directory, special file), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter is the file name. The second one is a pointer to a structure where the information requested is to be put. The fields in the structure are shown in Figure 14.28. The fstat call is the same as stat except that it operates on an open file (whose name may not be known) rather than on a path name.

The pipe system call is used to create shell pipelines. It creates a kind of pseudofile, which buffers the data between the pipeline components, and returns file descriptors for both reading and writing the buffer. In a pipeline such as sort <in | head –30 file descriptor 1 (standard output) in the process running

Notes

sort would be set (by the shell) to write to the pipe and file descriptor 0 (standard input) in the process running *head* would be set to read from the pipe. In this way, *sort* just reads from file descriptor 0 (set to the file *in*) and writes to file descriptor 1 (the pipe) without even being aware that these have been redirected. If they have not been redirected, *sort* will automatically read from the keyboard and write to the screen.

Figure 14.28: The Fields Returned by the Stat System Call

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identify of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Similarly, when *head* reads from file descriptor 0, it is reading the data *sort* put into the pipe buffer without even knowing that a pipe is in use. This is a clear example where a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) leads to a powerful tool (connecting program in arbitrary ways without having to modify them at all). The last system call in Figure 14.28 is fcntl. It is used to lock and unlock files, apply shared or exclusive locks, and perform a few other file-specific operations. Now let us look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file. Some common ones are listed in Figure 14.29. Directories are created and destroyed using mkdir and rmdir, respectively. A directory can only be removed if it is empty.

Figure 14.29: Some System Calls Relating to Directories

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

Notes

The return code s is ~1 if an error has occurred; dir identifies a directory stream and dirent is a directory entry. The parameters should be self-explanatory. As we saw in Figure 14.29, linking to a file creates a new directory entry that points to an existing file. The link system call creates the link. The parameters specify the original and new names, respectively. Directory entries are removed with unlink. When the last link to a file is removed, the file is automatically deleted. For a file that has never been linked, the first unlink causes it to disappear. The working directory is changed by the chdir system call. Doing so has the effect of changing the interpretation of relative path names.

The last four calls of Figure 14.30 are for reading directories. They can be opened, closed, and read, analogous to ordinary files. Each call to readdir returns exactly one directory entry in a fixed format. There is no way for users to write in a directory (in order to maintain the integrity of the file system). Files can be added to a directory using create or link and removed using unlink. There is no way to seek to a specific file in a directory, but rewinddir allows an open directory to be read again from the beginning.

14.6.3 Implementation of the Linux File System

In this section first we will look at the abstractions supported by the Virtual File System layer. The VFS hides from higher level processes and applications the differences among many types of file systems supported by Linux, whether they are residing on local devices or are stored remotely and need to be accessed over the network. Devices and other special files are also accessed through the VFS layer. Next, we will describe the implementation of the first wide-spread Linux file system, **ext2**, or the second **extended file system**. Afterward, we will discuss the improvements in the ext3 file system. A wide variety of other file systems are also in use. All Linux systems can handle multiple disk partitions, each with a different file system on it.

14.6.4 Linux Virtual File System

In order to enable applications to interact with different file systems, implemented on different types of local or remote devices, Linux adopts an approach used in other UNIX systems: the Virtual File System (VFS). VFS defines a set of basic file system abstractions and the operations which are allowed on these abstractions. Invocations of the system calls described in the previous section, access the VFS data structures, determine the exact file system where the accessed file belongs, and via function pointers stored in the VFS data structures invoke the corresponding operation in the specified file system. Figure 14.30 summarizes the four main file system structures supported by VFS. The **superblock** contains critical information about the layout of the file system. Destruction of the superblock will render the file system unreadable. The **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**) each describe exactly one file. Note that in Linux, directories and devices are also represented as files, thus they will have corresponding i-nodes. Both superblocks and i-nodes have a corresponding structure maintained on the physical disk where the file system resides.

Figure 14.30: File System Abstractions Supported by the VFS

Object	Description	Operation
Superblock	specific filesystem	read_inode, sync_fs
Dentry	directory entry, single component of a path	create, link
I-node	specific file	d_compare, d_delete
File	open file associated with a process	read, write

Notes

In order to facilitate certain directory operations and traversals of paths, such as /usr/ast/bin, VFS supports a **dentry** data structure which represents a directory entry. This data structure is created by the file system on the fly. Directory entries are cached in a dentry_ cache. For instance, the dentry_ cache would contain entries for /, /usr, /usr/ast, etc. If multiple processes access the same file through the same hard link (i.e., same path) their file object will point to the same entry in this cache.

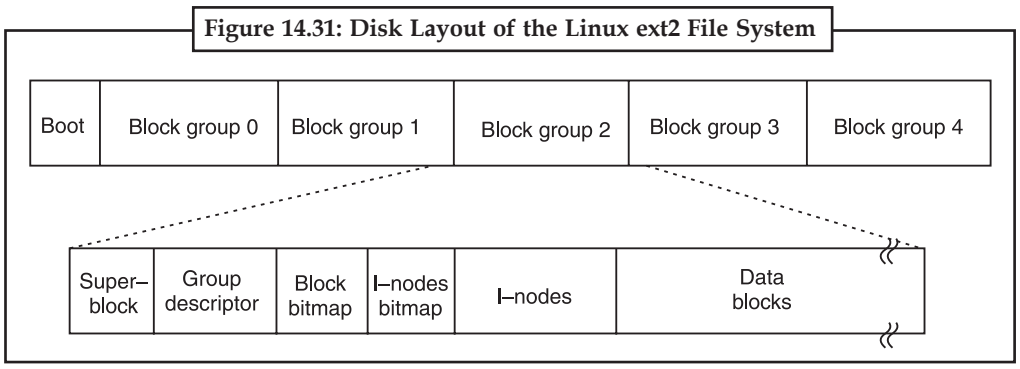
Finally, the **file** data structure is an in-memory representation of an open file, and is created in response to the open system call. It supports operations such as read, write, sendfile, lock, and other system calls described in the previous section. The actual file systems implemented underneath VFS need not use the exact same abstractions and operations internally. They must however implement semantically equivalent file system operations as the ones specified with the VFS objects. The elements of the operations data structures for each of the four VFS objects are pointers to functions in the underlying file system.

14.6.5 Linux Extended File System—Ext2

We next describe the most popular on-disk file system used in Linux — ext2. The first Linux release used the MINIX file system, and was limited by short filenames and 64 MB file sizes. The MINIX file system was replaced with the first extended file system, **ext**, which permitted both longer file names and larger file sizes. Due to its performance inefficiencies, ext is replaced by its successor, **ext2**, which is still in widespread use. An ext2 Linux disk partition contains a file system with the layout, illustrated in Figure 14.31. Block 0 is not used by Linux and often contains code to boot the computer. Following block 0, the disk partition is divided into groups of blocks, without regard to where the disk cylinder boundaries fall each group is organized as follows.

The first block is the **superblock**. It contains information about the layout of the file system, including the number of i-nodes, the number of disk blocks, and the start of the list of free disk blocks (typically a few hundred entries). Next comes the group descriptor, which contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group and the number of directories in the group. This information is important since ext2 attempts to spread directories evenly over the disk.

Figure 14.31: Disk Layout of the Linux ext2 File System



Two bitmaps keep track of the free blocks and free i-nodes, respectively, a choice inherited from the MINIX 1 file system (and in contrast to most UNIX file systems, which use a free list). Each map is one block long. With a 1 KB block, this design limits a block group to 8192 blocks and 8192 i-nodes. The former is a real restriction but the latter is not in practice.

Following the superblock are the i-nodes themselves. They are numbered from 1 up to some maximum. Each i-node is 128 bytes long and describes exactly one file. An i-node contains accounting information (including all the information returned by stat, which simply takes it from the i-node), as well as enough information to locate all the disk blocks that hold the file's data. Following i-nodes are

Notes

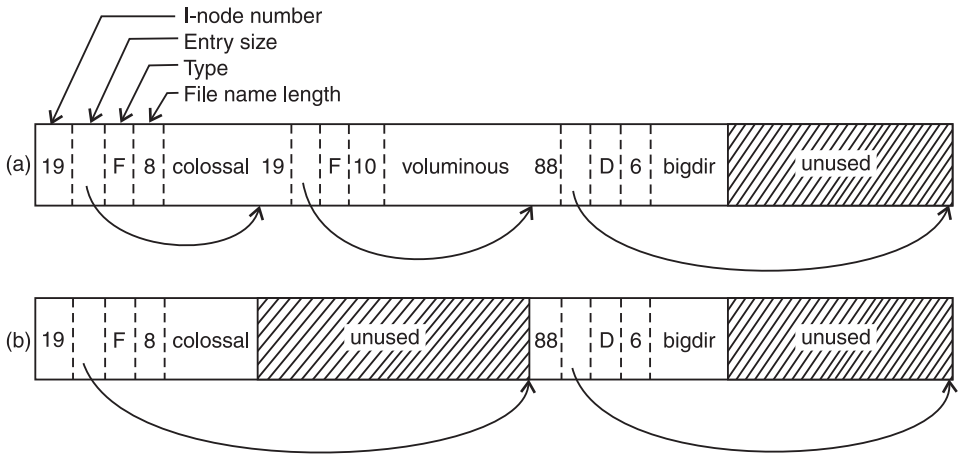
the data blocks. All the files and directories are stored here. If a file or directory consists of more than one block, the blocks need not be contiguous on the disk. In fact, the blocks of a large file are likely to be spread all over the disk.

I-nodes corresponding to directories are dispersed throughout the disk block groups. Ext2 attempts to collocate ordinary files in the same block group as the parent directory, and data files in the same block as the original file i-node, provided that there is sufficient space. This idea was taken from the Berkeley Fast File System (McKusick et al., 1984). The bitmaps are used to make quick decisions regarding where to allocate new file system data. When new file blocks are allocated, ext2 also *preallocates* a number (eight) of additional blocks for that file, so as to minimize the file fragmentation due to future write operations. This scheme balances the file system load across the entire disk. It also performs well due to its tendencies for collocation and reduced fragmentation. To access a file, it must first use one of the Linux system calls, such as open, which requires the file's pathname. The pathname is parsed to extract individual.

If a relative path is specified, the lookup starts from the process' current directory, otherwise it starts from the root directory. In either case, the i-node for the first directory can easily be located: there is a pointer to it in the process descriptor, or, in the case of a root directory, it is typically stored in a predetermined block on disk.

The directory file allows file names up to 255 characters and is illustrated in Figure 14.32. Each directory consists of some integral number of disk blocks so that directories can be written atomically to the disk. Within a directory, entries for files and directories are in unsorted order, with each entry directly following the one before it. Entries may not span disk blocks, so often there is some number of unused bytes at the end of each disk block.

Figure 14.32: (*a*) A Linux Directory with Three Files and (*b*) The Same Directory After the File Voluminous has been Removed



Each directory entry in Figure 14.32 consists of four fixed-length fields and one variable-length field. The first field is the i-node number, 19 for the file colossal, 42 for the file voluminous, and 88 for the directory bigdir. Next comes a field reclen, telling how big the entry is (in bytes), possibly including some padding after the name. This field is needed to find the next entry for the case

Notes

that the file name is padded by an unknown length. That is the meaning of the arrow in Figure 14.32. Then comes the type field: file, directory, etc. The last fixed field is the length of the actual file name in bytes, 8, 10, and 6 in this example. Finally, comes the file name itself, terminated by a 0 byte and padded out to a 32-bit boundary. Additional padding may follow that. In Figure 14.32, we see the same directory after the entry for voluminous has been removed. All that is done is increase the size of the total entry field for colossal, turning the former field for voluminous into padding for the first entry. This padding can be used for a subsequent entry, of course.

Since directories are searched linearly, it can take a long time to find an entry at the end of a large directory. Therefore, the system maintains a cache of recently accessed directories. This cache is searched using the name of the file, and if a hit occurs, the costly linear search is avoided. A dentry object is entered in the dentry cache for each of the path components, and, through its i-node, the directory is searched for the subsequent path element entry, until the actual file i-node is reached.

For instance, to look up a file specified with an absolute path name such as /usr/ast/file the following steps are required. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad block handling. It places an entry in the dentry cache for future lookups of the root directory. Then it looks up the string ''usr'' in the root directory, to get the i-node number of the /usr directory, which is also entered in the dentry cache. This i-node is then fetched, and the disk blocks are extracted from it, so the /usr directory can be read and searched for the string ''ast''. Once this entry is found, the i-node number for the /usr/ast directory can be taken from it. Armed with the i-node number of the /usr/ast directory, this i-node can be read and the directory blocks located. Finally, ''file'' is looked up and its i-node number found. Thus the use of a relative path name is not only more convenient for the user, but it also saves a substantial amount of work for the system.

If the file is present, the system extracts the i-node number, and uses this as an index into the i-node table (on disk) to locate the corresponding i-node and bring it into memory. The i-node is put in the **i-node table**, a kernel data structure that holds all the i-nodes for currently open files and directories. The format of the i-node entries, as a bare minimum, must contain all the fields returned by the stat system call so as to make stat work (see Figure 14.29). In Figure 14.33, we show the some of the fields included in the i-node structure supported by the Linux file system layer. The actual i-node structure contains many more fields, since the same structure is also used to represent directories, devices, and other special files. The i-node structure also contains fields reserved for future use. History has shown that unused bits do not remain that way for long.

Let us now see how the system reads a file. Remember that a typical call tothe library procedure for invoking the read system call looks like this: n = read(fd, buffer, nbytes);When the kernel gets control, all it has to start with are these three parameters,and the information in its internal tables relating to the user. One of the items in the internal tables is the file descriptor array. It is indexed by a file descriptor and contains one entry for each open file (up to the maximum number, usually defaults to 32). The idea is to start with this file descriptor and end up with the corresponding i-node.

Notes

Figure 14.33: Some Fields in the I-node Structure in Linux

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Ninks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Let us consider one possible design—just put a pointer to the i-node in the file descriptor table. Although simple, unfortunately, this method does not work. The problem is as follows. Associated with every file descriptor is a file position that tells at which byte the next read (or write) will start. Where should it go? One possibility is to put it in the i-node table. However, this approach fails if two or more unrelated processes happen to open the same file at the same time because each one has its own file position.

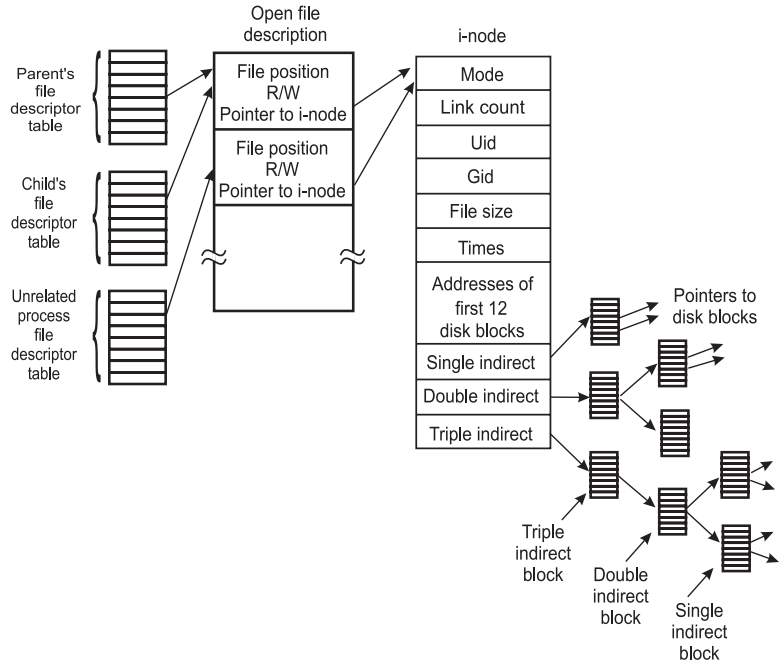
A second possibility is to put the file position in the file descriptor table. In this way, every process that opens a file gets its own private file position. Unfortunately, this scheme fails too, but the reasoning is more subtle and has to do with the nature of file sharing in Linux. Consider a shell script, s, consisting of two commands, p1 and p2, to be run in order. If the shell script is called by the command line

s >x

it is expected that p1 will write its output to x, and then p2 will write its output to x also, starting at the place where p1 stopped. When the shell forks off p1, x is initially empty, so p1 just starts writing at file position 0. However, when p1 finishes, some mechanism is needed to make sure that the initial file position that p2 sees is not 0 (which it would be if the file position were kept in the file descriptor table), but the value p1 ended with. The way this is achieved is shown in Figure 14.34. The trick is to introduce a new table, the **open file description table** between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first p1 and later p2. When the shell forks off p1, its user structure (including the file descriptor table) is an exact copy of the shell's, so both of them point to the same open file description table entry. When p1 finishes, the shell's file descriptor is still pointing to the open file description containing p1's file position. When the shell now forks off p2, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

Notes

Figure 14.34: The Relation between the File Descriptor Table, the Open File Description Table, and the I-node Table



However, if an unrelated process opens the file, it gets its own open file description entry, with its own file position, which is precisely what is needed. Thus the whole point of the open file description table is to allow a parent and child to share a file position, but to provide unrelated processes with their own values.

Getting back to the problem of doing the read, we have now shown how the file position and i-node are located. The i-node contains the disk addresses of the first 12 blocks of the file. If the file position falls in the first 12 blocks, the block is read and the data are copied to the user. For files longer than 12 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Figure 14.34. This block contains the disk addresses of more disk blocks. For example, if a block is 1 KB and a disk address is 4 bytes, the single indirect block can hold 256 disk addresses. Thus this scheme works for files of up to 268 KB in total. Beyond that, a **double indirect block** is used. It contains the addresses of 256 single indirect blocks, each of which holds the addresses of 256 data blocks. This mechanism is sufficient to handle files up to 10 ??216 blocks (67,119,104 bytes). If even this is not enough, the i-node has space for a **triple indirect block**. Its pointers point to many double indirect blocks. This addressing scheme can handle file sizes of 224 1 KB blocks (16 GB). For 8 KB block sizes, the addressing scheme can support file sizes up to 64 TB.

14.6.6 Linux Ext3 File System

In order to prevent data loss after system crashes and power failures, the ext2 file system would have to write out each data block to disk as soon as it was created. The latency incurred during the required disk head seek operation would be so high that the performance would be intolerable. Therefore, writes are delayed, and changes may not be committed to disk for up to 30 sec, which is a very long time interval in the context of modern computer hardware. To improve the robustness of the file system, Linux relies on **journaling file systems**. Ext3, a follow-on of the ext2 file system, is an example of a journaling file system.

Notes

The basic idea behind this type of file systems is to maintain a *journal*, which describes all file system operations in a sequential order. By sequentially writing out and changes to the file system data or metadata (i-nodes, superblock, etc.), the operations do not suffer from the overheads of disk head movement during random disk accesses. Eventually, the changes will be written out, committed, to the appropriate disk location, and the corresponding journal entries can be discarded.

If a system crash or power failure occurs before the changes are committed, during restart, the system will detect that the file system was not unmounted properly, will traverse the journal, and apply the file system changes described in the journal log.

Ext3 is designed to be highly compatible with ext2, and in fact, all core data structures and disk layout are the same in both systems. Furthermore, a file system which has been unmounted as an ext2 system, can be subsequently mounted as an ext3 system and offer the journaling capability.

The journal is a file used as a circular buffer. It may be stored on the same or separate device from the main file system. Since the journal operations are not "journalled" themselves, these are not handled by the same ext3 file system. Instead, a separate **JBD (Journaling Block Device)** is used to perform the journal read/write operations. JBD supports three main data structures: log record, atomic operation handle, and transaction. A log record describes a low level file system operation, typically resulting in changes within a block. Since a system call such as write includes changes at multiple places — i-nodes, existing file blocks, new file blocks, list of free blocks, etc., related log records are grouped in atomic operations. Ext3 notifies JBD of the start and end of a system call processing, so that JBD can ensure that either all log records in an atomic operation are applied, or none of them. Finally, primarily for efficiency reasons, JBD treats collections of atomic operations as transactions. Log records are stored consecutively within a transaction. JBD will allow portions of the journal file to be discarded only after all log records belonging to a transaction are safely committed to disk.

Since writing out a jog entry for each disk change may be costly, ext3 may be configured to keep a journal of all disk changes, or only of changes related to the file system metadata (the i-nodes, superblocks, bitmaps, and so on). Journaling metadata only introduces fewer system overheads and results in better performance, however does not make any guarantees against corruption of file data. Several other journaling file systems maintain logs of only metadata operations:

(e.g., SGI's XFS).

14.6.7 /proc File System

Another Linux file system is the /**proc** (process) file system, an idea originally devised in the 8th edition of UNIX from Bell Labs and later copied in 4.4BSD and System V. However, Linux extends the idea in several ways. The basic concept is that for every process in the system, a directory is created in /*proc*. The name of the directory is the process PID expressed as a decimal number. for example, /proc/619 is the directory corresponding to the process with PID 619. In this directory files that appear to contain information about the process, such as its command line, environment strings, and signal masks. In fact, these files does not exist on the disk. When they are read, the system retrieves the information from the actual process as needed and returns it in a standard format. Many of the Linux extensions relate to other files and directories located in /proc. They contain a wide variety of information about the CPU, disk partitions, devices, interrupt vectors, kernel counters, file systems, loaded modules, and much more. Unprivileged user programs may read much of this information to learn about system behavior in a safe way. Some of these files may be written to in order to change system parameters.

Notes

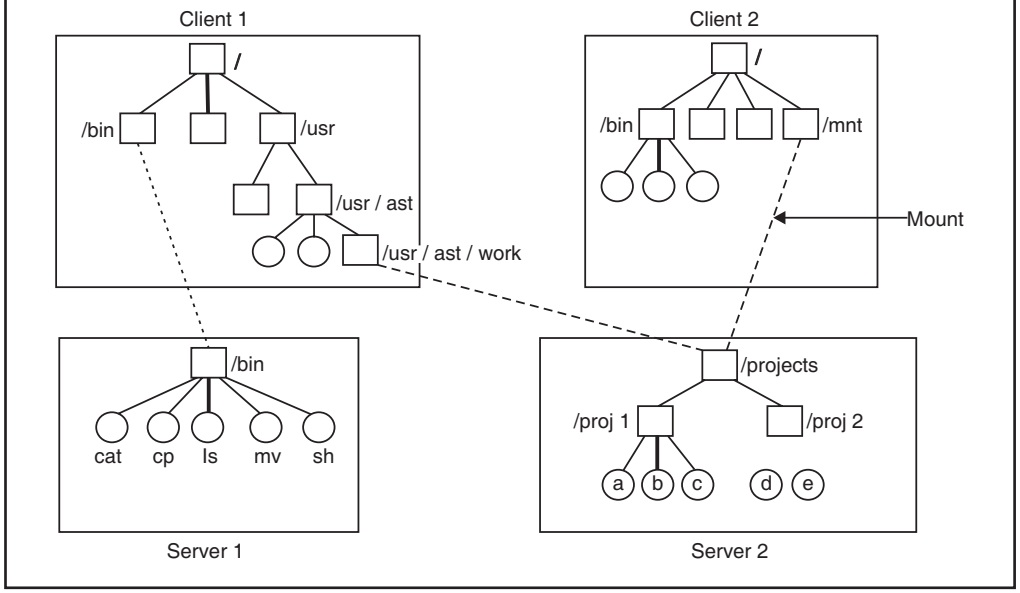
14.6.8 NFS—The Network File System

Networking has played a major role in Linux, and UNIX in general, right from the beginning (the first UNIX network was built to move new kernels from the PDP-11/70 to the Interdata 8/32 during the port to the later). In this section, we will examine Sun Microsystem's **NFS (Network File System)**, which is used on all modern Linux systems to join the file systems on separate computers into one logical whole. Currently, the most dominant NSF implementation is version 3, introduced in 1994 (Pawloski et al, 1994). NSFv4 was introduced in 2000 and provides several enhancements over the previous NFS architecture. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn, first in the context of the simpler NFS version 3, then we will briefly discuss the enhancements included in v4.

14.6.8.1 NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In many cases, all the clients and servers are on the same LAN, but this is not required. It is also possible to run NFS over a wide area network if the server is far from the client. For simplicity we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time. Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in a file, often /etc/exports, so these directories can be exported automatically whenever the server is booted. Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Figure 14.35.

Figure 14.35: Examples of Remote Mounted File Systems. Directories are Shown as Squares and Files are shown as Circles



Notes

In this example, client 1 has mounted the bin directory of server 1 on its own bin directory, so it can now refer to the shell as /bin/sh and get the shell on server 1. Diskless workstations often have only a skeleton file system (in RAM) and get all their files from remote servers like this. Similarly, client 1 has mounted server 2's directory /projects on its directory /usr/ast/work so it can now access file as a /usr/ast/work/proj1/a. Finally, client 2 has also mounted the projects directory and can also access file a, only as /mnt/proj1/a. As seen here, the same file can have different names on different clients due to its being mounted in a different place in the respective trees. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients.

14.6.8.2 NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is it possible for anyone to be able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

NFS accomplishes this goal by defining two client-server protocols. A **protocol** is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients. The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle** to the client. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory or any of its subdirectories use the file handle.

When Linux boots, it runs the /etc/rc shell script before going multiuser. Commands to mount remote file systems can be placed in this script, thus automatically mounting the necessary remote file systems before allowing any logins. Alternatively, most versions of Linux also support **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the /etc/rc file. First, if one of the NFS servers named in /etc/rc happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them needs to be up), and the performance can be improved (by choosing the first one to reply—presumably the least heavily loaded). On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change. The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and read and write files. Also, they can also access file attributes, such as file mode, size, and time of last modification. Most Linux system calls are supported by NFS, with the perhaps surprising exception of open and close.

The omission of open and close is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a

Notes

lookup message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file (i.e., contains a file system identifier and i-node number, among other data). Unlike an open call, this lookup operation does not copy any information into internal system tables. The read call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

Unfortunately, the NFS method makes it difficult to achieve the exact Linux file semantics. For example, in Linux a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the standard UNIX protection mechanism, with the rwx bits for the owner, group, and others (mentioned in Unit 1 and discussed in detail below). Originally, each request message simply contained the user and group IDs of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was—how shall we put it?—naive. Currently, public key cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is enabled, a malicious client cannot impersonate another client because it does not know that client's secret key.

14.6.8.3 NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, most Linux systems use a three-layer implementation. The top layer is the system call layer. This handles calls like open, read, and close. After parsing the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer.

The task of the VFS layer is to maintain a table with one entry for each open file. The VFS layer has an entry, a **virtual i-node**, or **v-node**, for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the file system and i-node are recorded because modern Linux systems can support multiple file systems (e.g., ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern Linux systems now support it as an integral part of the operating system, even if NFS is not used. To see how v-nodes are used, let us trace a sequence of mount, open, and read system calls. To mount a remote file system, the system administrator (or /etc/rc) calls the mount program specifying the remote directory, the local directory on which it is to be mounted, and other information. The mount program parses the name of the remote directory to be mounted and discovers the name of the NFS server on which the remote directory is located. It then contacts that machine asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a mount system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code to create an **r-node** (**remote i-node**) in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in one of the local file systems. Thus from the v-node it is possible to see if a file or directory is local or remote. If it is local, the correct file system and i-node can be located. If it is remote, the remote host and file handle can be located.

Notes

When a remote file is opened on the client, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, use it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled. When the file descriptor is used in a subsequent system call, for example, read, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it. It then sends a message to the server containing the handle, the file offset (which is maintained on the client side, not the server side), and the byte count. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested. When the request message arrives at the server, it is passed to the VFS layer there, which determines which local file system holds the requested file. The VFS layer then makes a call to that local file system to read and return the bytes. These data are then passed back to the client After the client's VFS layer has gotten the 8 KB chunk it asked for, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as **read ahead**, improves performance considerably. For writes an analogous path is followed from client to server. Also, transfers are done in 8 KB chunks here too. If a write system call supplies fewer than 8 KB bytes of data, the data are just accumulated locally. Only when the entire 8 KB chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided. While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and that one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally, the timer is 3 sec for ata blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server. While not perfect, these patches make the system highly usable in most practical circumstances.

Notes

14.6.8.4 NFS Version 4

Version 4 of the Network File System was designed to simplify certain operations from its predecessor. In contrast to NSFv3 which is described above, NFSv4 is a **stateful** file system. This permits open operations to be invoked on remote files, since the remote NFS server will maintain all file system related structures, including the file pointer. Read operations then need not include absolute read ranges, but can be incrementally applied from the previous file pointer position. This results in both, use of shorter messages, and also in the ability to bundle multiple NFSv3 operations in one network transaction. The stateful nature of NFSv4 makes it easy to also integrate the variety of NFSv3 protocols described earlier in this section into one coherent protocol. There is no need to support separate protocols for mounting, caching, locking, or secure operations. NFSv4 also works better with both Linux (and UNIX in general) and Windows file system semantics.

14.7 Security in Linux

Linux, as a clone of MINIX and UNIX, has been a multiuser system almost from the beginning. This history means that security and control of information was built in very early on. In the following sections, we will look at some of the security aspects of Linux.

Self Assessment

Fill in the blanks:

6. ..... command is used to create Linux file system.
7. In the Linux process, the ..... contains the machine instruction.
8. The kernel and memory map parts of main memory in Linux are ..... in memory.
9. A ..... file is one consisting of a sequence of numbered blocks.
10. ..... contains information about the layout of the file system.

14.8 Summary

- The first version of Linux 0.01 was released in 1991. It was cross-developed on a MINIX machine and borrowed numerous ideas from MINIX ranging from the structure of the source tree to the layout of the file system.
- Linux is a multiprogramming system, so multiple independent processes may be running at the same time.
- When the computer is start, the BIOS performs powers-ON self (POST) and initial device process.
- The Linux memory model is straightforward to make programs portable and to make it possible to implement Linux on machine with widely differing memory management units.
- The I/O system in Linux is fairly straightforward.
- The most visible part of any operating system, including Linux is the file system.
- Security of Linux. The Linux as a clone of MINIX and UNIX has been a multiuser system almost from the beginning.

Notes

14.9 Keywords

Kernel: The Linux kernel is an operating system where kernel used by the Linux family of Unix-like operating systems. It is one of the most prominent examples of free and open source software.

Memory: The term memory identifies data storage that comes in the form of chips, and the word storage is used for memory that exists on tapes or disks. Moreover, the term memory is usually used as shorthand for physical memory, which refers to the actual chips capable of holding data. Some computers also use virtual memory, which expands physical memory onto a hard disk.

MINIX : MINIX is a new open-source operating system designed to be highly reliable, flexible, and secure. It is loosely based somewhat on previous versions of MINIX, but is fundamentally different in many key ways.

Page Allocator: An allocation scheme which combines a normal power of two allocator with free buffer coalescing and the basic concept behind it is quite simple. Memory is broken up into a large blocks of pages where each block is a power of two numbers of pages.

Paging: The operating system copies a certain number of pages from your storage device to main memory. When a program needs a page that is not in main memory, the operating system copies the required page into memory and copies another page back to the disk. One says that the operating system pages the data. Each time a page is needed that is not currently in memory, a page fault occurs. An invalid page fault occurs when the address of the page being requested is invalid.

System Call: Operating systems contain sets of routines for performing various low-level operations. For example, all operating systems have a routine for creating a directory. If you want to execute an operating system routine from a program, you must make a system call.

Threads: A thread is placeholder information associated with a single use of a program that can handle multiple concurrent users.

UNIX: Unix is a multitasking, multiuser computer operating system, is originally developed in 1969 by a group of AT&T employees at Bell Labs.

14.10 Review Questions

1. Can a page fault ever lead to the faulting process being terminated? If so, give an example. If not, why not?
2. Why are open file description tables necessary in Linux?
3. A non-real time Linux process has priority levels from 100 to 139. What is the default static priority and how is the nice values used to change this?
4. Why do you think the designers of Linux made it impossible for a process to send a signal to a another process that is not in its process group?
5. In every process' entry is the task structure, the PID of the process' parent is stored. Why?
6. When a new process is forked off, it must be assigned a unique integer as its PID. Is it sufficient to have a counter in the kernel that is incremented on each process creation, with the counter used as the new PID? Discuss your answer.

Answers to Self Assessment

1. (*a*) 2. (*b*) 3. control 4. window manager
5. Magic characters 6. fsck 7. text segement
8. pinned 9. block special file 10. Superblock

Notes

14.11 Further Readings



Books

Operating Systems, by Harvey M. Deitel, Paul J. Deitel, David R. Choffnes. *Introduction to Operating Design and Implementation,* by Michael Kifer, Scoott A. Smolka.



Online link

wiley.com/coolege.silberschatz