

OPERATING SYSTEM

Edited By
Dr. Pooja Gupta

ISBN: 978-93-87034-83-9

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:	Introduction to Operating System *Sarabjit Kumar, Lovely Professional University*	1
Unit 2:	Operation and Function of Operating System *Pooja Gupta, Lovely Professional University*	14
Unit 3:	Operating System Structure *Deepak Mehta, Lovely Professional University*	29
Unit 4:	Process Management *Mithilesh Kumar Dubey, Lovely Professional University*	48
Unit 5:	Scheduling *Sahil Rampal, Lovely Professional University*	70
Unit 6:	Process Synchronization *Pooja Gupta, Lovely Professional University*	96
Unit 7:	Memory Management *Pawan Kumar, Lovely Professional University*	119
Unit 8:	File Management *Mithilesh Kumar Dubey, Lovely Professional University*	139
Unit 9:	I/O & Secondary Storage Structure *Ramandeep Kaur, Lovely Professional University*	159
Unit 10:	System Protection *Sarabjit Kumar, Lovely Professional University*	182
Unit 11:	System Security *Mandeep Kaur, Lovely Professional University*	200
Unit 12:	Security Solution *Ramandeep Kaur, Lovely Professional University*	225
Unit 13:	Case Study: Linux *Yadwinder Singh, Lovely Professional University*	241
Unit 14:	Windows 2000 *Pooja Gupta, Lovely Professional University*	300

SYLLABUS

Operating System

Objectives: In order to meet the ever increasing need of computers, study of operating system is compulsory. This is core technology subject and the knowledge of which is absolutely essential for Computer Engineers. It familiarizes the students with the concepts and functions of operating system. This subject provides knowledge to develop systems using advanced operating system concepts.

- To learn the evolution of Operating systems.
- To study the operations performed by Operating System as a resource manager.
- To study computer security issues and Operating System tools.

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: Introduction to Operating System

Notes

CONTENTS

Objectives

Introduction

1.1 Operating System: Meaning

1.2 History of Computer Operating Systems

1.3 Supervisor and User Mode

1.4 Goals of an Operating System

1.5 Generations of Operating Systems

1.5.1 0th Generation

1.5.2 First Generation (1951-1956)

1.5.3 Second Generation (1956-1964)

1.5.4 Third Generation (1964-1979)

1.5.5 Fourth Generation (1979 – Present)

1.6 Summary

1.7 Keywords

1.8 Self Assessment

1.9 Review Questions

1.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Define operating system
- Know supervisor and user mode
- Explain various goals of an operating system
- Describe generation of operating systems

Introduction

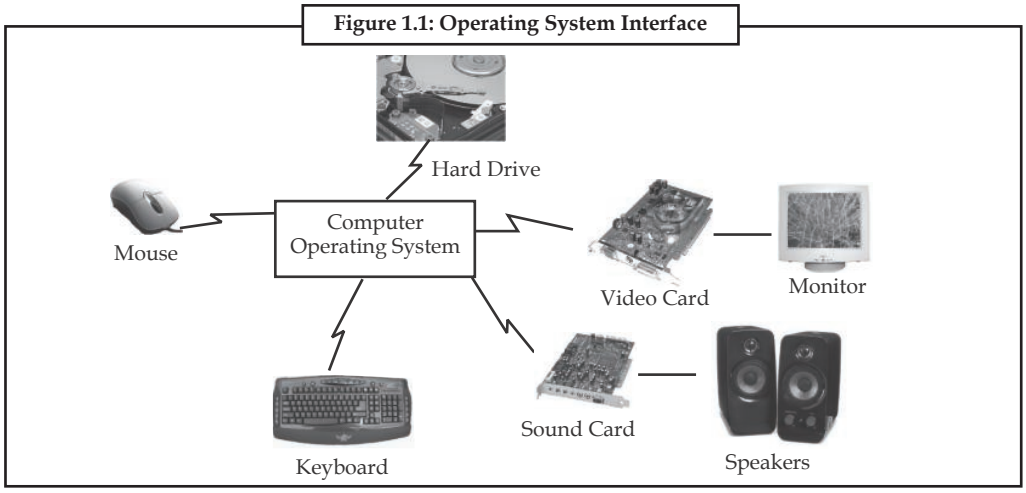
An Operating System (OS) is a collection of programs that acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute the programs. Operating Systems are viewed as resource managers. The main resource is the computer hardware in the form of processors, storage, input/output devices, communication devices, and data. Some of the operating system functions are: implementing the user interface, sharing hardware among users, allowing users to share data among themselves, preventing users from interfering with one another, scheduling resources among users, facilitating input/output, recovering from errors, accounting for resource usage, facilitating parallel operations, organising data for secure and rapid access, and handling network communications.

Notes

1.1 Operating System: Meaning

An operating system (sometimes abbreviated as "OS") is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. The other programs are called applications or application programs. The application programs make use of the operating system by making requests for services through a defined Application Program Interface (API). In addition, users can interact directly with the operating system through a user interface such as a command language or a Graphical User Interface (GUI).

Figure 1.1: Operating System Interface



In a computer system, you find four main components: the hardware, the operating system, the application software and the users. In a computer system, the hardware provides the basic computing resources. The applications programs define the way 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 systems programs and application programs for the various users.

You can view an operating system as a resource allocator. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, files storage space, input/output devices etc. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently.

An operating system is a control program. This program controls the execution of user programs to prevent errors and improper use of the computer. Operating systems exist because: they are a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of a computer system is to execute user programs and solve user problems.

While there is no universally agreed upon definition of the concept of an operating system, the following is a reasonable starting point:

A computer's operating system is a group of programs designed to serve two basic purposes:

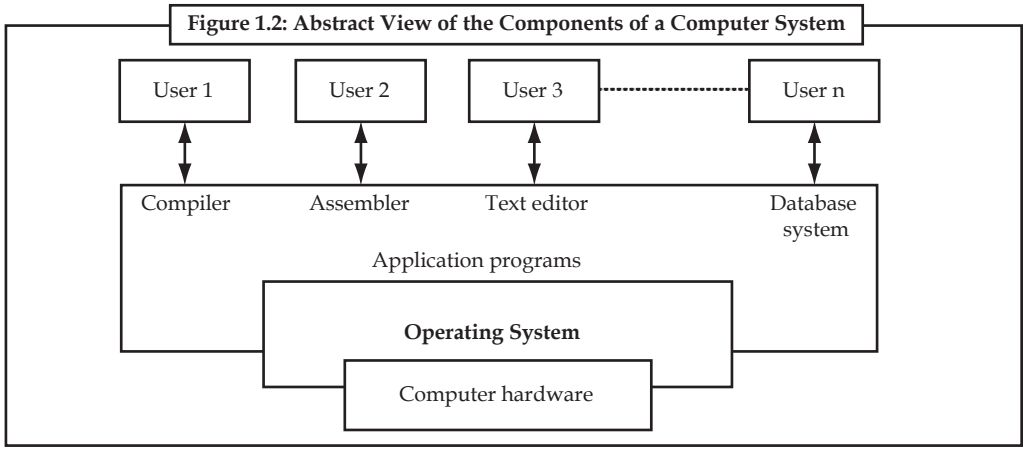
1. To control the allocation and use of the computing system's resources among the various users and tasks, and
2. To provide an interface between the computer hardware and the programmer that simplifies and makes feasible the creation, coding, debugging, and maintenance of application programs.

Notes

An effective operating system should accomplish the following functions:

1. Should act as a command interpreter by providing a user friendly environment.
2. Should facilitate communication with other users.
3. Facilitate the directory/file creation along with the security option.
4. Provide routines that handle the intricate details of I/O programming.
5. Provide access to compilers to translate programs from high-level languages to machine language.
6. Provide a loader program to move the compiled program code to the computer's memory for execution.
7. Assure that when there are several active processes in the computer, each will get fair and non-interfering access to the central processing unit for execution.
8. Take care of storage and device allocation.
9. Provide for long term storage of user information in the form of files.
10. Permit system resources to be shared among users when appropriate, and be protected from unauthorised or mischievous intervention as necessary.

Figure 1.2: Abstract View of the Components of a Computer System



Though systems programs such as editors and translators and the various utility programs (such as sort and file transfer program) are not usually considered part of the operating system, the operating system is responsible for providing access to these system resources.

The abstract view of the components of a computer system and the positioning of OS is shown in the Figure 1.2.

Task "Operating system is a hardware or software". Discuss.

1.2 History of Computer Operating Systems

Early computers lacked any form of operating system. The user had sole use of the machine and would arrive armed with program and data, often on punched paper and tape. The program would be loaded into the machine, and the machine would be set to work until the program completed or crashed. Programs could generally be debugged via a front panel using switches and lights. It is said that Alan Turing was a master of this on the early Manchester Mark I machine,

Notes

and he was already deriving the primitive conception of an operating system from the principles of the Universal Turing machine.

Later machines came with libraries of support code, which would be linked to the user's program to assist in operations such as input and output. This was the genesis of the modern-day operating system. However, machines still ran a single job at a time; at Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different colored clothes-pegs to indicate job-priority.

As machines became more powerful, the time needed for a run of a program diminished and the time to hand off the equipment became very large by comparison. Accounting for and paying for machine usage moved on from checking the wall clock to automatic logging by the computer. Run queues evolved from a literal queue of people at the door, to a heap of media on a jobs-waiting table, or batches of punch-cards stacked one on top of the other in the reader, until the machine itself was able to select and sequence which magnetic tape drives were online. Where program developers had originally had access to run their own jobs on the machine, they were supplanted by dedicated machine operators who looked after the well-being and maintenance of the machine and were less and less concerned with implementing tasks manually. When commercially available computer centers were faced with the implications of data lost through tampering or operational errors, equipment vendors were put under pressure to enhance the runtime libraries to prevent misuse of system resources. Automated monitoring was needed not just for CPU usage but for counting pages printed, cards punched, cards read, disk storage used and for signaling when operator intervention was required by jobs such as changing magnetic tapes.

All these features were building up towards the repertoire of a fully capable operating system. Eventually the runtime libraries became an amalgamated program that was started before the first customer job and could read in the customer job, control its execution, clean up after it, record its usage, and immediately go on to process the next job. Significantly, it became possible for programmers to use symbolic program-code instead of having to hand-encode binary images, once task-switching allowed a computer to perform translation of a program into binary form before running it. These resident background programs, capable of managing multistep processes, were often called monitors or monitor-programs before the term operating system established itself.

An underlying program offering basic hardware-management, software-scheduling and resource-monitoring may seem a remote ancestor to the user-oriented operating systems of the personal computing era. But there has been a shift in meaning. With the era of commercial computing, more and more "secondary" software was bundled in the operating system package, leading eventually to the perception of an operating system as a complete user-system with utilities, applications (such as text editors and file managers) and configuration tools, and having an integrated graphical user interface. The true descendant of the early operating systems is what we now call the "kernel". In technical and development circles the old restricted sense of an operating system persists because of the continued active development of embedded operating systems for all kinds of devices with a data-processing component, from hand-held gadgets up to industrial robots and real-time control-systems, which do not run user-applications at the front-end. An embedded operating system in a device today is not so far removed as one might think from its ancestor of the 1950s.

1.3 Supervisor and User Mode

Single user mode is a mode in which a multiuser computer operating system boots into a single superuser. It is mainly used for maintenance of multi-user environments such as network servers. Some tasks may require exclusive access to shared resources, for example running fsck on a network share. This mode may also be used for security purposes – network services are

Notes

not run, eliminating the possibility of outside interference. On some systems a lost superuser password can be changed by switching to single user mode, but not asking for the password in such circumstances is viewed as a security vulnerability.

You are all familiar with the concept of sitting down at a computer system and writing documents or performing some task such as writing a letter. In this instance, there is one keyboard and one monitor that you interact with.

Operating systems such as Windows 95, Windows NT Workstation and Windows 2000 professional are essentially single user operating systems. They provide you the capability to perform tasks on the computer system such as writing programs and documents, printing and accessing files.

Consider a typical home computer. There is a single keyboard and mouse that accept input commands, and a single monitor to display information output. There may also be a printer for the printing of documents and images.

In essence, a single-user operating system provides access to the computer system by a single user at a time. If another user needs access to the computer system, they must wait till the current user finishes what they are doing and leaves.

Students in computer labs at colleges or University often experience this. You might also have experienced this at home, where you want to use the computer but someone else is currently using it. You have to wait for them to finish before you can use the computer system.

1.4 Goals of an Operating System

The primary objective of a computer is to execute an instruction in an efficient manner and to increase the productivity of processing resources attached with the computer system such as hardware resources, software resources and the users. In other words, you can say that maximum CPU utilisation is the main objective, because it is the main device which is to be used for the execution of the programs or instructions. Brief the goals as:

1. The primary goal of an operating system is to make the computer convenient to use.
2. The secondary goal is to use the hardware in an efficient manner.

1.5 Generations of Operating Systems

Operating systems have been evolving over the years. you will briefly look at this development of the operating systems with respect to the evolution of the hardware/architecture of the computer systems in this section. Since operating systems have historically been closely tied with the architecture of the computers on which they run, you will look at successive generations of computers to see what their operating systems were like. You may not exactly map the operating systems generations to the generations of the computer, but roughly it provides the idea behind them.

You can roughly divide them into five distinct generations that are characterized by hardware component technology, software development, and mode of delivery of computer services.

1.5.1 0th Generation

The term 0th generation is used to refer to the period of development of computing, which predated the commercial production and sale of computer equipment. You consider that the period might be way back when Charles Babbage invented the Analytical Engine. Afterwards the computers by John Atanasoff in 1940; the Mark I, built by Howard Aiken and a group of IBM engineers at Harvard in 1944; the ENIAC, designed and constructed at the University of Pencylvania by

Notes

Wallace Eckert and John Mauchly and the EDVAC, developed in 1944-46 by John Von Neumann, Arthur Burks, and Herman Goldstine (which was the first to fully implement the idea of the stored program and serial execution of instructions) were designed. The development of EDVAC set the stage for the evolution of commercial computing and operating system software. The hardware component technology of this period was electronic vacuum tubes.

The actual operation of these early computers took place without the benefit of an operating system. Early programs were written in machine language and each contained code for initiating operation of the computer itself.

The mode of operation was called "open-shop" and this meant that users signed up for computer time and when a user's time arrived, the entire (in those days quite large) computer system was turned over to the user. The individual user (programmer) was responsible for all machine set up and operation, and subsequent clean-up and preparation for the next user. This system was clearly inefficient and dependent on the varying competencies of the individual programmer as operators.

1.5.2 First Generation (1951-1956)

The first generation marked the beginning of commercial computing, including the introduction of Eckert and Mauchly's UNIVAC I in early 1951, and a bit later, the IBM 701 which was also known as the Defence Calculator. The first generation was characterised again by the vacuum tube as the active component technology.

Operation continued without the benefit of an operating system for a time. The mode was called "closed shop" and was characterised by the appearance of hired operators who would select the job to be run, initial program load the system, run the user's program, and then select another job, and so forth. Programs began to be written in higher level, procedure-oriented languages, and thus the operator's routine expanded. The operator now selected a job, ran the translation program to assemble or compile the source program, and combined the translated object program along with any existing library programs that the program might need for input to the linking program, loaded and ran the composite linked program, and then handled the next job in a similar fashion.

Application programs were run one at a time, and were translated with absolute computer addresses that bound them to be loaded and run from these reassigned storage addresses set by the translator, obtaining their data from specific physical I/O device. There was no provision for moving a program to different location in storage for any reason. Similarly, a program bound to specific devices could not be run at all if any of these devices were busy or broken.

The inefficiencies inherent in the above methods of operation led to the development of the mono-programmed operating system, which eliminated some of the human intervention in running job and provided programmers with a number of desirable functions. The OS consisted of a permanently resident kernel in main storage, and a job scheduler and a number of utility programs kept in secondary storage. User application programs were preceded by control or specification cards (in those day, computer program were submitted on data cards) which informed the OS of what system resources (software resources such as compilers and loaders; and hardware resources such as tape drives and printer) were needed to run a particular application. The systems were designed to be operated as batch processing system.

These systems continued to operate under the control of a human operator who initiated operation by mounting a magnetic tape that contained the operating system executable code onto a "boot device", and then pushing the IPL (Initial Program Load) or "boot" button to initiate the bootstrap loading of the operating system. Once the system was loaded, the operator entered the date and time, and then initiated the operation of the job scheduler program which read and interpreted the control statements, secured the needed resources, executed the first user

Notes

program, recorded timing and accounting information, and then went back to begin processing of another user program, and so on, as long as there were programs waiting in the input queue to be executed.

The first generation saw the evolution from hands-on operation to closed shop operation to the development of mono-programmed operating systems. At the same time, the development of programming languages was moving away from the basic machine languages; first to assembly language, and later to procedure oriented languages, the most significant being the development of FORTRAN by John W. Backus in 1956. Several problems remained, however, the most obvious was the inefficient use of system resources, which was most evident when the CPU waited while the relatively slower, mechanical I/O devices were reading or writing program data. In addition, system protection was a problem because the operating system kernel was not protected from being overwritten by an erroneous application program.

Moreover, other user programs in the queue were not protected from destruction by executing programs.

1.5.3 Second Generation (1956-1964)

The second generation of computer hardware was most notably characterised by transistors replacing vacuum tubes as the hardware component technology. In addition, some very important changes in hardware and software architectures occurred during this period. For the most part, computer systems remained card and tape-oriented systems. Significant use of random access devices, that is, disks, did not appear until towards the end of the second generation. Program processing was, for the most part, provided by large centralised computers operated under mono-programmed batch processing operating systems.

The most significant innovations addressed the problem of excessive central processor delay due to waiting for input/output operations. Recall that programs were executed by processing the machine instructions in a strictly sequential order. As a result, the CPU, with its high speed electronic component, was often forced to wait for completion of I/O operations which involved mechanical devices (card readers and tape drives) that were order of magnitude slower. This problem led to the introduction of the data channel, an integral and special-purpose computer with its own instruction set, registers, and control unit designed to process input/output operations separately and asynchronously from the operation of the computer's main CPU near the end of the first generation, and its widespread adoption in the second generation.

The data channel allowed some I/O to be buffered. That is, a program's input data could be read "ahead" from data cards or tape into a special block of memory called a buffer. Then, when the user's program came to an input statement, the data could be transferred from the buffer locations at the faster main memory access speed rather than the slower I/O device speed. Similarly, a program's output could be written another buffer and later moved from the buffer to the printer, tape, or card punch. What made this all work was the data channel's ability to work asynchronously and concurrently with the main processor. Thus, the slower mechanical I/O could be happening concurrently with main program processing. This process was called I/O overlap.

The data channel was controlled by a channel program set up by the operating system I/O control routines and initiated by a special instruction executed by the CPU. Then, the channel independently processed data to or from the buffer. This provided communication from the CPU to the data channel to initiate an I/O operation. It remained for the channel to communicate to the CPU such events as data errors and the completion of a transmission. At first, this communication was handled by polling – the CPU stopped its work periodically and polled the channel to determine if there is any message.

Polling was obviously inefficient (imagine stopping your work periodically to go to the post office to see if an expected letter has arrived) and led to another significant innovation of the

Notes

second generation – the interrupt. The data channel was able to interrupt the CPU with a message – usually "I/O complete." Infact, the interrupt idea was later extended from I/O to allow signalling of number of exceptional conditions such as arithmetic overflow, division by zero and time-run-out. Of course, interval clocks were added in conjunction with the latter, and thus operating system came to have a way of regaining control from an exceptionally long or indefinitely looping program.

These hardware developments led to enhancements of the operating system. I/O and data channel communication and control became functions of the operating system, both to relieve the application programmer from the difficult details of I/O programming and to protect the integrity of the system to provide improved service to users by segmenting jobs and running shorter jobs first (during "prime time") and relegating longer jobs to lower priority or night time runs. System libraries became more widely available and more comprehensive as new utilities and application software components were available to programmers.

In order to further mitigate the I/O wait problem, system were set up to spool the input batch from slower I/O devices such as the card reader to the much higher speed tape drive and similarly, the output from the higher speed tape to the slower printer. In this scenario, the user submitted a job at a window, a batch of jobs was accumulated and spooled from cards to tape "off line," the tape was moved to the main computer, the jobs were run, and their output was collected on another tape that later was taken to a satellite computer for off line tape-to-printer output. User then picked up their output at the submission windows.

Toward the end of this period, as random access devices became available, tape-oriented operating system began to be replaced by disk-oriented systems. With the more sophisticated disk hardware and the operating system supporting a greater portion of the programmer's work, the computer system that users saw was more and more removed from the actual hardware-users saw a virtual machine.

The second generation was a period of intense operating system development. Also it was the period for sequential batch processing. But the sequential processing of one job at a time remained a significant limitation. Thus, there continued to be low CPU utilisation for I/O bound jobs and low I/O device utilisation for CPU bound jobs. This was a major concern, since computers were still very large (room-size) and expensive machines. Researchers began to experiment with multiprogramming and multiprocessing in their computing services called the time-sharing system.



Note A noteworthy example is the Compatible Time Sharing System (CTSS), developed at MIT during the early 1960s.



Task CPU is the heart of computer system what about ALU.

1.5.4 Third Generation (1964-1979)

The third generation officially began in April 1964 with IBM's announcement of its System/360 family of computers. Hardware technology began to use Integrated Circuits (ICs) which yielded significant advantages in both speed and economy.

Operating system development continued with the introduction and widespread adoption of multiprogramming. This marked first by the appearance of more sophisticated I/O buffering

Notes

in the form of spooling operating systems, such as the HASP (Houston Automatic Spooling) system that accompanied the IBM OS/360 system. These systems worked by introducing two new systems programs, a system reader to move input jobs from cards to disk, and a system writer to move job output from disk to printer, tape, or cards. Operation of spooling system was, as before, transparent to the computer user who perceived input as coming directly from the cards and output going directly to the printer.

The idea of taking fuller advantage of the computer's data channel I/O capabilities continued to develop. That is, designers recognised that I/O needed only to be initiated by a CPU instruction – the actual I/O data transmission could take place under control of separate and asynchronously operating channel program. Thus, by switching control of the CPU between the currently executing user program, the system reader program, and the system writer program, it was possible to keep the slower mechanical I/O device running and minimizes the amount of time the CPU spent waiting for I/O completion. The net result was an increase in system throughput and resource utilisation, to the benefit of both user and providers of computer services.

This concurrent operation of three programs (more properly, apparent concurrent operation, since systems had only one CPU, and could, therefore execute just one instruction at a time) required that additional features and complexity be added to the operating system. First, the fact that the input queue was now on disk, a direct access device, freed the system scheduler from the first-come-first-served policy so that it could select the "best" next job to enter the system (looking for either the shortest job or the highest priority job in the queue). Second, since the CPU was to be shared by the user program, the system reader, and the system writer, some processor allocation rule or policy was needed. Since the goal of spooling was to increase resource utilisation by enabling the slower I/O devices to run asynchronously with user program processing, and since I/O processing required the CPU only for short periods to initiate data channel instructions, the CPU was dispatched to the reader, the writer, and the program in that order. Moreover, if the writer or the user program was executing when something became available to read, the reader program would preempt the currently executing program to regain control of the CPU for its initiation instruction, and the writer program would preempt the user program for the same purpose. This rule, called the static priority rule with preemption, was implemented in the operating system as a system dispatcher program.

The spooling operating system in fact had multiprogramming since more than one program was resident in main storage at the same time. Later this basic idea of multiprogramming was extended to include more than one active user program in memory at time. To accommodate this extension, both the scheduler and the dispatcher were enhanced. The scheduler became able to manage the diverse resource needs of the several concurrently active used programs, and the dispatcher included policies for allocating processor resources among the competing user programs. In addition, memory management became more sophisticated in order to assure that the program code for each job or at least that part of the code being executed, was resident in main storage.

The advent of large-scale multiprogramming was made possible by several important hardware innovations such as:

1. The widespread availability of large capacity, high-speed disk units to accommodate the spooled input streams and the memory overflow together with the maintenance of several concurrently active program in execution.
2. Relocation hardware which facilitated the moving of blocks of code within memory without any undue overhead penalty.
3. The availability of storage protection hardware to ensure that user jobs are protected from one another and that the operating system itself is protected from user programs.
4. Some of these hardware innovations involved extensions to the interrupt system in order to handle a variety of external conditions such as program malfunctions, storage protection

Notes

violations, and machine checks in addition to I/O interrupts. In addition, the interrupt system became the technique for the user program to request services from the operating system kernel.

5. The advent of privileged instructions allowed the operating system to maintain coordination and control over the multiple activities now going on with in the system.

Successful implementation of multiprogramming opened the way for the development of a new way of delivering computing services-time-sharing. In this environment, several terminals, sometimes up to 200 of them, were attached (hard wired or via telephone lines) to a central computer. Users at their terminals, "logged in" to the central system, and worked interactively with the system. The system's apparent concurrency was enabled by the multiprogramming operating system. Users shared not only the system hardware but also its software resources and file system disk space.

The third generation was an exciting time, indeed, for the development of both computer hardware and the accompanying operating system. During this period, the topic of operating systems became, in reality, a major element of the discipline of computing.

1.5.5 Fourth Generation (1979 – Present)

The fourth generation is characterised by the appearance of the personal computer and the workstation. Miniaturisation of electronic circuits and components continued and Large Scale Integration (LSI), the component technology of the third generation, was replaced by Very Large Scale Integration (VLSI), which characterizes the fourth generation. VLSI with its capacity for containing thousands of transistors on a small chip, made possible the development of desktop computers with capabilities exceeding those that filled entire rooms and floors of building just twenty years earlier.

The operating systems that control these desktop machines have brought us back in a full circle, to the open shop type of environment where each user occupies an entire computer for the duration of a job's execution. This works better now, not only because the progress made over the years has made the virtual computer resulting from the operating system/hardware combination so much easier to use, or, in the words of the popular press "user-friendly."

However, improvements in hardware miniaturisation and technology have evolved so fast that you now have inexpensive workstation – class computers capable of supporting multiprogramming and time-sharing. Hence the operating systems that supports today's personal computers and workstations look much like those which were available for the minicomputers of the third generation.



Example: Microsoft's DOS for IBM-compatible personal computers and UNIX for workstation.

However, many of these desktop computers are now connected as networked or distributed systems. Computers in a networked system each have their operating systems augmented with communication capabilities that enable users to remotely log into any system on the network and transfer information among machines that are connected to the network. The machines that make up distributed system operate as a virtual single processor system from the user's point of view; a central operating system controls and makes transparent the location in the system of the particular processor or processors and file systems that are handling any given program.

1.6 Summary

- This unit presented the principle operation of an operating system. In this unit you had briefly described about the history, the generations and the types of operating systems.

Notes

- An operating system is a program that acts as an interface between a user of a computer and the computer hardware.
- The purpose of an operating system is to provide an environment in which a user may execute programs.
- The primary goal of an operating system is to make the computer convenient to use. And the secondary goal is to use the hardware in an efficient manner.

1.7 Keywords

An Operating System: It is the most important program in a computer system that runs all the time, as long as the computer is operational and exits only when the computer is shut down.

Desktop System: Modern desktop operating systems usually feature a Graphical user interface (GUI) which uses a pointing device such as a mouse or stylus for input in addition to the keyboard.

Operating System: An operating system is a layer of software which takes care of technical aspects of a computer's operation.

1.8 Self Assessment

Choose the appropriate answers:

1. GUI stands for
   (a) Graphical Used Interface
   (b) Graphical User Interface
   (c) Graphical User Interchange
   (d) Good User Interface
2. CPU stands for
   (a) Central Program Unit
   (b) Central Programming Unit
   (c) Central Processing Unit
   (d) Centralization Processing Unit
3. FORTRAN stands for
   (a) Formula Translation
   (b) Formula Transformation
   (c) Formula Transition
   (d) Forming Translation
4. VLSI stands for
   (a) Very Long Scale Integration
   (b) Very Large Scale Interchange
   (c) Very Large Scale Interface
   (d) Very Large Scale Integration

Notes

5. API stands for
   (a) Application Process Interface
   (b) Application Process Interchange
   (c) Application Program Interface
   (d) Application Process Interfacing

Fill in the blanks:

6. An operating system is a ..... .
7. Programs could generally be debugged via a front panel using ..... and lights.
8. The data channel allowed some ..... to be buffered.
9. The third generation officially began in April ..... .
10. The system's apparent concurrency was enabled by the multiprogramming .....

1.9 Review Questions

1. What is the relation between application software and operating system?
2. What is an operating system? Is it a hardware or software?
3. Mention the primary functions of an operating system.
4. Briefly explain the evolution of the operating system.
5. What are the key elements of an operating system?
6. What do you understand by the term computer generations?
7. Who give the idea of stored program and in which year? Who give the basic structure of computer?
8. Give the disadvantages of first generation computers over second generation computers.
9. On which system, the second generation computers based on? What are the new inventions in the second generation of computers?
10. Describe the term integrated circuit.
11. What is the significance of third generation computers?
12. Give the brief description of fourth generation computers. How the technology is better than previous generation?
13. What is the period of fifth generation computers?
14. What are the differences between hardware and software?
15. What are the differences between system software and application software?

Answers: Self Assessment

1.	(b)	2.	(c)	3.	(a)	4.	(d)
5.	(c)	6.	control program	7.	switches	8.	I/O
9.	1964	10.	operating system				

1.10 Further Readings

Notes



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 2: Operation and Function of Operating System

CONTENTS

Objectives

Introduction

2.1 Operations and Functions of OS

2.2 Types of Operating System

2.3 Operating System: Examples

2.3.1 Disk Operating System (DOS)

2.3.2 UNIX

2.3.3 Windows

2.3.4 Macintosh

2.4 Summary

2.5 Keywords

2.6 Self Assessment

2.7 Review Questions

2.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe operations and functions of operating system
- Explain various types of operating system

Introduction

The primary objective of operating system is to increase productivity of a processing resource, such as computer hardware or computer-system users. User convenience and productivity were secondary considerations. At the other end of the spectrum, an OS may be designed for a personal computer costing a few thousand dollars and serving a single user whose salary is high. In this case, it is the user whose productivity is to be increased as much as possible, with the hardware utilization being of much less concern. In single-user systems, the emphasis is on making the computer system easier to use by providing a graphical and hopefully more intuitively obvious user interface.

2.1 Operations and Functions of OS

The main operations and functions of an operating system are as follows:

1. Process Management
2. Memory Management
3. Secondary Storage Management
4. I/O Management

Notes

5. File Management
6. Protection
7. Networking Management
8. Command Interpretation.

Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

The operating system is responsible for the following activities in connection with processes management:

1. The creation and deletion of both user and system processes
2. The suspension and resumption of processes.
3. The provision of mechanisms for process synchronization
4. The provision of mechanisms for deadlock handling.

Memory Management

Memory is the most expensive part in the computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

1. Keep track of which parts of memory are currently being used and by whom.
2. Decide which processes are to be loaded into memory when memory space becomes available.
3. Allocate and deallocate memory space as needed.

Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system.

Notes

There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management:

1. Free space management
2. Storage allocation
3. Disk scheduling.

I/O Management

One of the purposes of an operating system is to hide the peculiarities or specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The operating system is responsible for the following activities in connection to I/O management:

1. A buffer caching system
2. To activate a general device driver code
3. To run the driver software for specific hardware devices as and when required.

File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms: magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organisation.

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general a files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organised into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection to the file management:

1. The creation and deletion of files.
2. The creation and deletion of directory.
3. The support of primitives for manipulating files and directories.
4. The mapping of files onto disk storage.
5. Backup of files on stable (non volatile) storage.
6. Protection and security of the files.

Notes

Protection

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorisation from the operating system.



Example: Memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user.



Task "Memory is the most expensive part of system." Discuss.

Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies and the problems of connection and security.

A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.

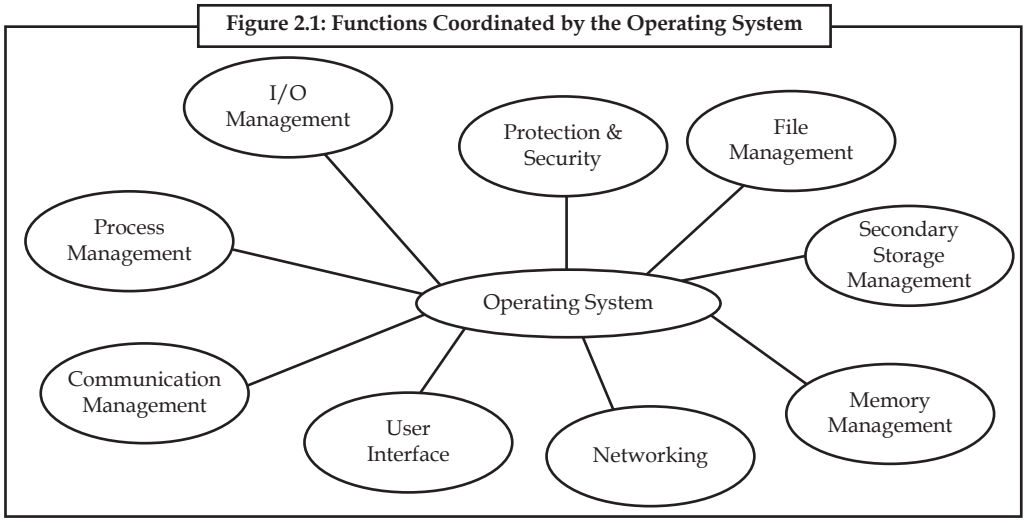
Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed. This program is variously called (1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement, and execute it.

The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

Notes

The Figure 2.1 depicts the role of the operating system in coordinating all the functions.

Figure 2.1: Functions Coordinated by the Operating System



2.2 Types of Operating System

Modern computer operating systems may be classified into three groups, which are distinguished by the nature of interaction that takes place between the computer user and his or her program during its processing. The three groups are called batch, time-sharing and real-time operating systems.

Batch Processing Operating System

In a batch processing operating system environment users submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. In this case, the user has no interaction with the job during its processing, and the computer's response time is the turnaround time the time from submission of the job until execution is complete, and the results are ready for return to the person who submitted the job.

Time Sharing

Another mode for delivering computing services is provided by time sharing operating systems. In this environment a computer provides computing services to several or many users concurrently on-line. Here, the various users are sharing the central processor, the memory, and other resources of the computer system in a manner facilitated, controlled, and monitored by the operating system. The user, in this environment, has nearly full interaction with the program during its execution, and the computer's response time may be expected to be no more than a few second.

Real-time Operating System (RTOS)

The third class is the real time operating systems, which are designed to service those applications where response time is of the essence in order to prevent error, misrepresentation or even disaster. Examples of real time operating systems are those which handle airlines reservations, machine tool control, and monitoring of a nuclear power station. The systems, in this case, are designed to be interrupted by external signals that require the immediate attention of the computer system.

Notes

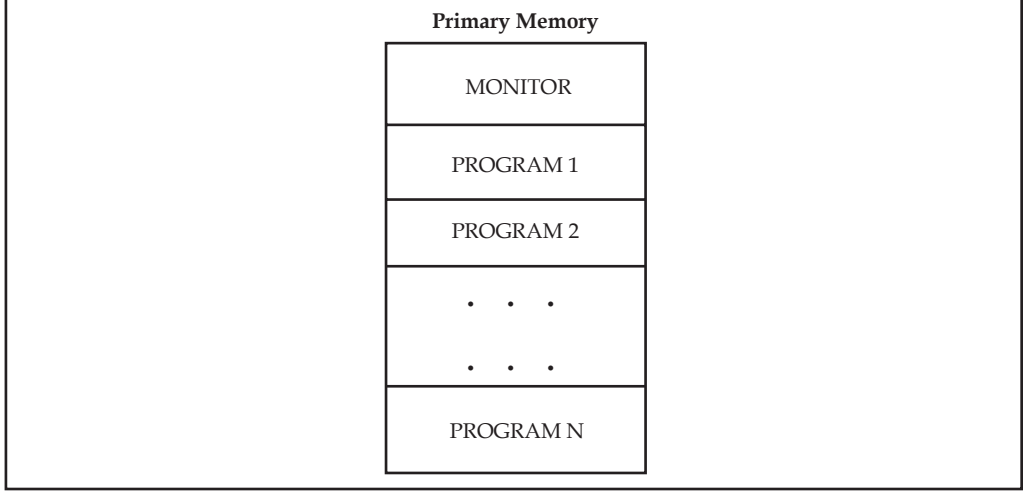
These real time operating systems are used to control machinery, scientific instruments and ndustrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities. A very important part of an RTOS is managing the resources of the computer so that a particular operation 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.

A number of other definitions are important to gain an understanding of operating systems:

Multiprogramming Operating System

A multiprogramming operating system is a system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously. Thus, it is evident that a time-sharing system is a multiprogramming system, but note that a multiprogramming system is not necessarily a time-sharing system. A batch or real time operating system could, and indeed usually does, have more than one active user program simultaneously in main storage. Another important, and all too similar, term is "multiprocessing".

Figure 2.2: Memory Layout in Multiprogramming Environment



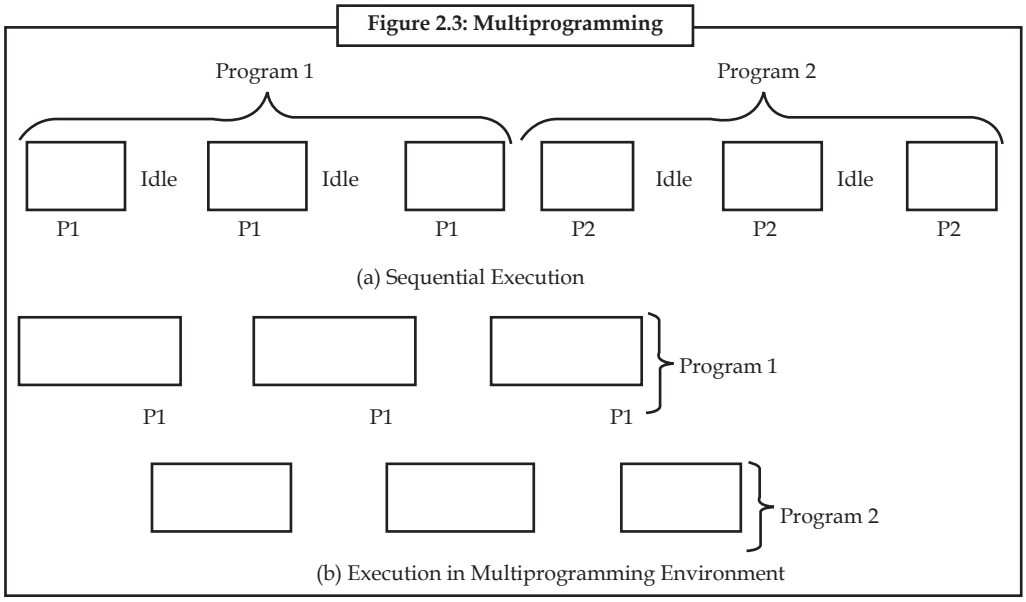
Buffering and Spooling improve system performance by overlapping the input, output and computation of a single job, but both of them have their limitations. A single user cannot always keep CPU or I10 devices busy at all times. Multiprogramming offers a more efficient approach to increase system performance. In order to increase the resource utilisation, systems supporting multiprogramming approach allow more than one job (program) to reside in the memory to utilise CPU time at any moment. More number of programs competing for system resources better will mean better resource utilisation.

The idea is implemented as follows. The main memory of a system contains more than one program (Figure 2.2).

The operating system picks one of the programs and starts executing. During execution of program 1 it needs some I/O operation to complete in a sequential execution environment (Figure 2.3a). The CPU would then sit idle whereas in a multiprogramming system, (Figure 2.3b) the operating system will simply switch over to the next program (program 2).

Notes

Figure 2.3: Multiprogramming



When that program needs to wait for some 110 operation, it switches over to program 3 and so on. If there is no other new program left in the main memory, the CPU will pass its control back to the previous programs.

Multiprogramming has traditionally been employed to increase the resources utilisation of a computer system and to support multiple simultaneously interactive users (terminals).

Multiprocessing System

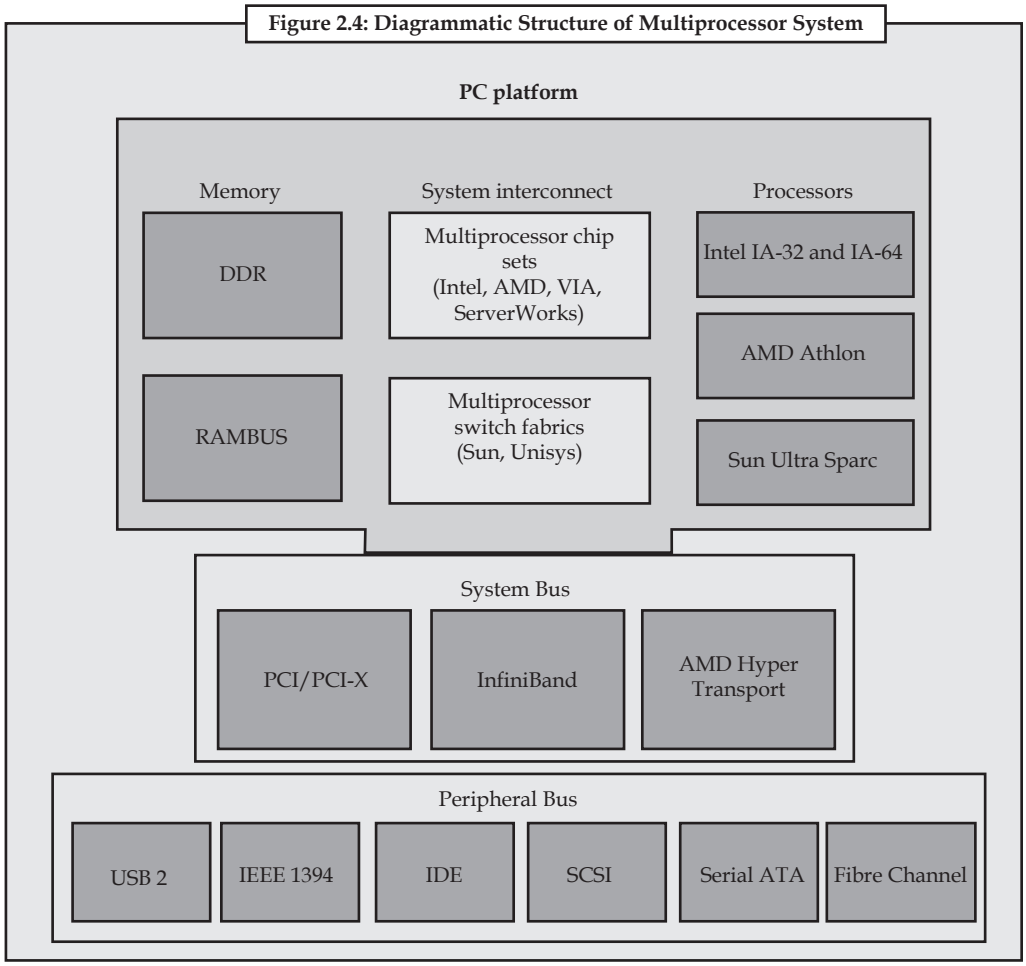
A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications.

A multiprocessor system is simply a computer that has >1 & not <=1 CPU on its motherboard. If the operating system is built to take advantage of this, it can run different processes (or different threads belonging to the same process) on different CPUs.

Today's operating systems strive to make the most efficient use of a computer's resources. Most of this efficiency is gained by sharing the machine's resources among several tasks (multi-processing). Such "large-grain" resource sharing is enabled by operating systems without any additional information from the applications or processes. All these processes can potentially execute concurrently, with the CPU (or CPUs) multiplexed among them. Newer operating systems provide mechanisms that enable applications to control and share machine resources at a finer grain-, that is, at the threads level. Just as multiprocessing operating systems can perform more than one task concurrently by running more than a single process, a process can perform more than one task by running more than a single thread.

Notes

Figure 2.4: Diagrammatic Structure of Multiprocessor System



Task Distinguish between multiprocessing and multitasking OS.

Networking Operating System

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handing communication these additions do not change the essential structure of the operating systems.

Distributed Operating System

A distributed computing system consists of a number of computers that are connected and managed so that they automatically share the job processing load among the constituent computers, or separate the job load as appropriate particularly configured processors. Such a system requires an operating system which, in addition to the typical stand-alone functionality, provides coordination of the operations and information flow among the component computers. The networked and distributed computing environments and their respective operating systems are designed with more complex functional capabilities. In a network operating system, the users are aware of the existence of multiple computers, and can log in to remote machines and copy

Notes

files from one machine to another. Each machine runs its own local operating system and has its own user (or users).

A distributed operating system, in contrast, is one that appears to its users as a traditional uni-processor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uni-processor operating system, because distributed and centralised systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimise the amount of parallelism achieved.

Operating Systems for Embedded Devices

As embedded systems (PDAs, cellphones, point-of-sale devices, VCR's, industrial robot control, or even your toaster) become more complex hardware-wise with every generation, and more features are put into them day-by-day, applications they run require more and more to run on actual operating system code in order to keep the development time reasonable. Some of the popular OS are:

1. ***Nexus's Conix:*** an embedded operating system for ARM processors.
2. ***Sun's Java OS:*** a standalone virtual machine not running on top of any other OS; mainly targeted at embedded systems.
3. ***Palm Computing's Palm OS:*** Currently the leader OS for PDAs, has many applications and supporting companies.
4. Microsoft's Windows CE and Windows NT Embedded OS.

Single Processor System

In theory, every computer system may be programmed in its machine language, with no systems software support. Programming of the "bare-machines" was customary for early computer systems. A slightly more advanced version of this mode of operating is common for the simple evaluation boards that are sometimes used in introductory microprocessor design and interfacing courses.

Programs for the bare machine can be developed by manually translating sequences of instructions into binary or some other code whose base is usually an integer power of 2. Instructions and data are then fed into the computer by means of console switches, or perhaps through a hexadecimal keyboard. Programs are started by loading the program counter with the address of the first instruction. Results of execution are obtained by examining the contents of the relevant registers and memory locations. Input/Output devices, if any, must be controlled by the executing program directly, say, by reading and writing the related I/O ports. Evidently, programming of the bare machine results in low productivity of both users and hardware. The long and tedious process of program and data entry practically precludes execution of all but very short programs in such an environment.

The next significant evolutionary step in computer system usage came about with the advent of input/output devices, such as punched cards and paper tape, and of language translators. Programs, now coded in a programming language, are translated into executable form by a computer program, such as compiler or an interpreter. Another program, called the loader, automates the process of loading executable programs into memory. The user places a program

Notes

and its input data on an input device, and the loader transfers information from that input device into memory. After transferring control to the loaded program by manual or automatic means, execution of the program commences. The executing program reads its input from the designated input device and may produce some output on an output device, such as a printer or display screen. Once in memory, the program may be rerun with different set of input data.

The mechanics of development and preparation of programs in such environments are quite slow and cumbersome due to serial execution of programs and numerous manual operations involved in the process. In a typical sequence, the editor program is loaded to prepare the source code of the user program. The next step is to load and execute the language translator and to provide it with the source code of the user program. When serial input devices, such as card readers, are used, multiple-pass language translators may require the source code to be repositioned for reading during each pass. If syntax errors are detected, the whole process must be repeated from the beginning. Eventually, the object code produced from the syntactically correct source code is loaded and executed. If run-time errors are detected, the state of the machine can be examined and modified by means of console switches, or with the assistance of a program called a debugger. The mode of operation described here was initially used in late fifties, but it was also common in low-end microcomputers of early eighties with cassettes as I/O devices.

In addition to language translators, system software includes the loader and possibly editor and debugger programs. Most of them use input/output devices and thus must contain some code to exercise those devices. Since many user programs also use input/output devices, the logical refinement is to provide a collection of standard I/O routines for the use of all programs.

In the described system, I/O routines and the loader program represent a rudimentary form of an operating system. Although quite crude, it still provides an environment for execution of programs far beyond what is available on the bare machine. Language translators, editors, and debuggers are system programs that rely on the services of, but are not generally regarded as part of, the operating system.

Although a definite improvement over the bare machine approach, this mode of operation is obviously not very efficient. Running of the computer system may require frequent manual loading of programs and data. This results in low utilization of system resources. User productivity, especially in multiuser environments, is low as users await their turn at the machine. Even with such tools as editors and debuggers, program development is very slow and is ridden with manual program and data loading.

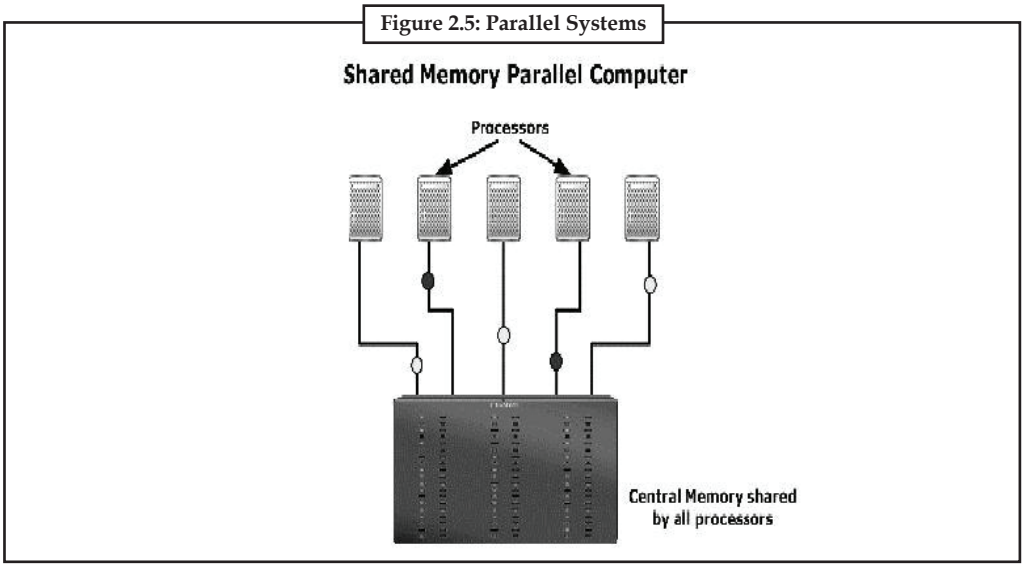
Parallel Processing System

Parallel operating systems are primarily concerned with managing the resources of parallel machines. This task faces many challenges: application programmers demand all the performance possible, many hardware configurations exist and change very rapidly, yet the operating system must increasingly be compatible with the mainstream versions used in personal computers and workstations due both to user pressure and to the limited resources available for developing new versions of these system. There are several components in an operating system that can be parallelized. Most operating systems do not approach all of them and do not support parallel applications directly. Rather, parallelism is frequently exploited by some additional software layer such as a distributed file system, distributed shared memory support or libraries and services that support particular parallel programming languages while the operating system manages concurrent task execution.

The convergence in parallel computer architectures has been accompanied by a reduction in the diversity of operating systems running on them. The current situation is that most commercially available machines run a flavour of the UNIX OS (Digital UNIX, IBM AIX, HP UX, Sun Solaris, Linux).

Notes

Figure 2.5: Parallel Systems



Others run a UNIX based microkernel with reduced functionality to optimize the use of the CPU, such as Cray Research's UNICOS. Finally, a number of shared memory MIMD machines run Microsoft Windows NT (soon to be superseded by the high end variant of Windows 2000).

There are a number of core aspects to the characterization of a parallel computer operating system: general features such as the degrees of coordination, coupling and transparency; and more particular aspects such as the type of process management, inter-process communication, parallelism and synchronization and the programming model.

Multitasking

In computing, multitasking is a method where multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

In the early ages of the computers, they where considered advanced card machines and therefore the jobs they performed where like: "find all females in this bunch of cards (or records)". Therefore, utilisation was high since one delivered a job to the computing department, which prepared and executed the job on the computer, delivering the final result to you. The advances in electroniv engineering increased the processing power serveral times, now leaving input/output devices (card readers, line printers) far behind. This ment that the CPU had to wait for the data it required to perform a given task. Soon, engineers thought: "what if we could both prepare, process and output data at the same time" and multitasking was born. Now one could read data for the next job while executing the current job and outputting the results of a previously job, thereby increasing the utilisation of the very expensive computer.

Cheap terminals allowed the users themselves to input data to the computer and to execute jobs (having the department do it often took days) and see results immediately on the screen, which introduced what was called interactive tasks. They required a console to be updated when a key

Notes

was pressed on the keyboard (again a task with slow input). Same thing happens today, where your computer actually does no work most of the time - it just waits for your input. Therefore using multitasking where serveral tasks run on the same computer improves performance.

Multitasking is the process of letting the operating system perform multiple task at what seems to the user simultaniously. In SMP (Symmetric Multi Processor systems) this is the case, since there are serveral CPU's to execute programs on - in systems with only a single CPU this is done by switching execution very rapidly between each program, thus givin the impression of simultanious execution. This process is also known as task swithcing or timesharing. Practically all modern OS has this ability.

Multitasking is, on single-processor machines, implemented by letting the running process own the CPU for a while (a timeslice) and when required gets replaced with another process, which then owns the CPU. The two most common methods for sharing the CPU time is either cooperative multitasking or preempetive multitasking.

Cooperative Multitasking: The simplest form of multitasking is cooperative multitasking. It lets the programs decide when they wish to let other tasks run. This method is not good since it lets one process monopolise the CPU and never let other processes run. This way a program may be reluctant to give away processing power in the fear of another process hogging all CPU-time. Early versions of the MacOS (up til MacOS 8) and versions of Windows earlier than Win95/WinNT used cooperative multitasking (Win95 when running old apps).

Preempetive Multitasking: Preempetive multitasking moves the control of the CPU to the OS, letting each process run for a given amount of time (a timeslice) and then switching to another task. This method prevents one process from taking complete control of the system and thereby making it seem as if it is crashed. This method is most common today, implemented by among others OS/2, Win95/98, WinNT, Unix, Linux, BeOS, QNX, OS9 and most mainframe OS. The assignment of CPU time is taken care of by the scheduler.

2.3 Operating System: Examples

2.3.1 Disk Operating System (DOS)

DOS (Disk Operating System) was the first widely-installed operating system for personal computers. It is a master control program that is automatically run when you start your personal computer (PC). DOS stays in the computer all the time letting you run a program and manage files. It is a single-user operating system from Microsoft for the PC. It was the first OS for the PC and is the underlying control program for Windows 3.1, 95, 98 and ME. Windows NT, 2000 and XP emulate DOS in order to support existing DOS applications.

2.3.2 UNIX

UNIX operating systems are used in widely-sold workstation products from Sun Microsystems, Silicon Graphics, IBM, and a number of other companies. The UNIX environment and the client/server program model were important elements in the development of the Internet and the reshaping of computing as centered in networks rather than in individual computers. Linux, a UNIX derivative available in both "free software" and commercial versions, is increasing in popularity as an alternative to proprietary operating systems.

UNIX is written in C. Both UNIX and C were developed by AT&T and freely distributed to government and academic institutions, causing it to be ported to a wider variety of machine families than any other operating system. As a result, UNIX became synonymous with "open systems".

Notes

UNIX is made up of the kernel, file system and shell (command line interface). The major shells are the Bourne shell (original), C shell and Korn shell. The UNIX vocabulary is exhaustive with more than 600 commands that manipulate data and text in every way conceivable. Many commands are cryptic, but just as Windows hid the DOS prompt, the Motif GUI presents a friendlier image to UNIX users. Even with its many versions, UNIX is widely used in mission critical applications for client/server and transaction processing systems. The UNIX versions that are widely used are Sun's Solaris, Digital's UNIX, HP's HP-UX, IBM's AIX and SCO's UnixWare. A large number of IBM mainframes also run UNIX applications, because the UNIX interfaces were added to MVS and OS/390, which have obtained UNIX branding. Linux, another variant of UNIX, is also gaining enormous popularity.

2.3.3 Windows

Windows is a personal computer operating system from Microsoft that, together with some commonly used business applications such as Microsoft Word and Excel, has become a de facto "standard" for individual users in most corporations as well as in most homes. Windows contains built-in networking, which allows users to share files and applications with each other if their PC's are connected to a network. In large enterprises, Windows clients are often connected to a network of UNIX and NetWare servers. The server versions of Windows NT and 2000 are gaining market share, providing a Windows-only solution for both the client and server. Windows is supported by Microsoft, the largest software company in the world, as well as the Windows industry at large, which includes tens of thousands of software developers.

This networking support is the reason why Windows became successful in the first place. However, Windows 95, 98, ME, NT, 2000 and XP are complicated operating environments. Certain combinations of hardware and software running together can cause problems, and troubleshooting can be daunting. Each new version of Windows has interface changes that constantly confuse users and keep support people busy, and Installing Windows applications is problematic too. Microsoft has worked hard to make Windows 2000 and Windows XP more resilient to installation of problems and crashes in general.

2.3.4 Macintosh

The Macintosh (often called "the Mac"), introduced in 1984 by Apple Computer, was the first widely-sold personal computer with a Graphical User Interface (GUI). The Mac was designed to provide users with a natural, intuitively understandable, and, in general, "user-friendly" computer interface. This includes the mouse, the use of icons or small visual images to represent objects or actions, the point-and-click and click-and-drag actions, and a number of window operation ideas. Microsoft was successful in adapting user interface concepts first made popular by the Mac in its first Windows operating system. The primary disadvantage of the Mac is that there are fewer Mac applications on the market than for Windows. However, all the fundamental applications are available, and the Macintosh is a perfectly useful machine for almost everybody. Data compatibility between Windows and Mac is an issue, although it is often overblown and readily solved.

The Macintosh has its own operating system, Mac OS which, in its latest version is called Mac OS X. Originally built on Motorola's 68000 series microprocessors, Mac versions today are powered by the PowerPC microprocessor, which was developed jointly by Apple, Motorola, and IBM. While Mac users represent only about 5% of the total numbers of personal computer users, Macs are highly popular and almost a cultural necessity among graphic designers and online visual artists and the companies they work for.



Task DOS is a character based operating system what about Windows operating system.

Notes

2.4 Summary

- Operating systems may be classified by both how many tasks they can perform "simultaneously" and by how many users can be using the system "simultaneously". That is: single-user or multi-user and single-task or multi-tasking.
- A multi-user system must clearly be multi-tasking.
- A possible solution to the external fragmentation problem is to permit the logical address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available.
- Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages.
- Memory protection in a paged environment is accomplished by protection bit that are associated with each frame.
- Segmentation is a memory-management scheme that supports this user view of memory.
- Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment.

2.5 Keywords

Clustered System: A clustered system is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer.

Distributed System: A distributed system is a computer system in which the resources resides in separate units connected by a network, but which presents to the user a uniform computing environment.

Real-time Operating System: A Real-time Operating System (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers, mobile telephones), industrial robots, spacecraft, industrial control and scientific research equipment.

2.6 Self Assessment

Fill in the blanks:

1. A ..... is a program in execution.
2. ..... is a large array of words or bytes, each with its own address.
3. A ..... is a collection of related information defined by its creator.
4. A ..... provides the user with access to the various resources the system maintains.
5. An RTOS typically has very little user-interface capability, and no ..... .
6. A ..... cannot always keep CPU or I10 devices busy at all times.
7. A multiprocessing system is a computer hardware configuration that includes more than ..... independent processing unit.
8. A ..... system is a collection of physical interconnected computers.
9. A system task, such as ....., is also a process.
10. ..... is achieved through a sequence of reads or writes of specific memory address.

Notes

2.7 Review Questions

1. Write short note on Distributed System.
2. Explain the nature of real time system.
3. What is batch system? What are the shortcomings of early batch systems? Explain it.
4. Write the differences between the time sharing system and distributed system.
5. Describe real time operating system. Give an example of it.
6. Explain parallel system with suitable example.
7. Write the differences between the real time system and personal system.
8. "Most modern computer systems use disks as the primary on-line storage of information, of both programs and data". Explain.
9. Write short note on networking.
10. "The operating system picks one of the programs and starts executing". Discuss.

Answers: Self Assessment

1.	process	2.	Memory	3.	file		
4.	distributed system	5.	end-user utilities	6.	single user		
7.	one	8.	networked computing	9.	spooling	10.	Interaction

2.8 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 3: Operating System Structure

Notes

CONTENTS

Objectives

Introduction

3.1 Operating System Services

3.2 System Calls

3.3 System Programs

3.4 Operating System Structure

3.4.1 Monolithic Systems

3.4.2 Client-server Model

3.4.3 Exokernel

3.5 Layered Structure

3.6 Virtual Machine

3.7 Summary

3.8 Keywords

3.9 Self Assessment

3.10 Review Questions

3.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe operating system services
- Define system calls
- Explain system programs
- Know operating system structure
- Describe layered structure

Introduction

Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop – it makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process

Notes

itself is not allowed to do. It is because of the critical nature of operations that the operating system itself does them every time they are needed.



Example: For I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

The fact that improper use of the system can easily cause a system crash, thus the operating system is introduced; it executes at the highest level of order and allows the applications to request for a service – a system call – which is implemented through hooking interrupt(s). A system call is the mechanism used by an application program to request service from the operating system. There are different types of system call.

3.1 Operating System Services

The various operating system services are:

1. Program Execution
2. I/O Operations
3. File System Manipulation
4. Communications
5. Error Detection

Program Execution

The purpose of a computer systems is to allow the user to execute programs. So the operating systems provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a program involves the allocating and deallocating memory, CPU scheduling in case of multiprocess. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So the operating systems by providing I/O makes it convenient for the users to run programs.

For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating systems provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his/her task accomplished. Thus operating systems makes it easier for user programs to accomplished their task.

This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence I think it is

Notes

best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this services left with operating system.

Communications

There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Error Detection

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service cannot allowed to be handled by user programs because it involves monitoring and in cases altering area of memory or deallocation of memory for a faulty process. Or may be relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

3.2 System Calls

System calls provide an interface between a running program and operating system. System calls are generally available as assembly language instructions. Several higher level languages such as C also allow to make system calls directly.

In UNIX operating system the system call interface layer contains entry point in kernel code. All system resources are managed by the kernel. Any request from user or application that involves access to any system resource must be handled by kernel code. The user process must not be given open access to kernel code for security reason. Many opening into kernel code called system calls are provided to user so that the user processes can invoke the execution of kernel code. System calls allow processes and users to manipulate system resources.

There are three general methods that are used to pass information (parameters) between a running program and the operating system.

1. One method is to store parameters in registers.
2. Another is to store parameters in a table in memory and pass the address of table.
3. The third method is to push parameters on stack and allow operating system to pop the parameters off the stack.

System Calls for Process Management

These types of system calls are used to control the processes. Some examples are end, abort, load, execute, create process, terminate process etc.



Example: The exit() system call ends a process and returns a value to it parent.

Notes

In UNIX every process has an alarm clock stored in its system-data segment. When the alarm goes off, signal SIGALRM is sent to the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared. The alarm clock remains set across an exec.

System Calls for Signaling

A signal is a limited form of inter-process communication used in UNIX, UNIX-like, and other POSIX-compliant operating systems. Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred. The number of signals available is system dependent. When a signal is sent to a process, the operating system interrupts the process' normal flow of execution. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed.

Programs can respond to signals three different ways. These are:

1. ***Ignore the signal:*** This means that the program will never be informed of the signal no matter how many times it occurs.
2. A signal can be set to its default state, which means that the process will be ended when it receives that signal.
3. ***Catch the signal:*** When the signal occurs, the system will transfer control to a previously defined subroutine where it can respond to the signal as is appropriate for the program.

System Calls for File Management

The file structure related system calls available in some operating system like UNIX let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. When doing I/O, a process specifies the file descriptor for an I/O channel, a buffer to be filled or emptied, and the maximum size of data to be transferred. An I/O channel may allow input, output, or both. Furthermore, each channel has a read/write pointer. Each I/O operation starts where the last operation finished and advances the pointer by the number of bytes transferred. A process can access a channel's data randomly by changing the read/write pointer.

These types of system calls are used to manage files.



Example: Create file, delete file, open, close, read, write etc.

System Calls for Directory Management

You may need the same sets of operations as for file management for directories also. If you have a directory structure for organizing files in the file system. In addition, for either files or directories, you need to be able to determine the values of various attributes, and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls.

System Calls for Protection

Improper use of the system can easily cause a system crash. Therefore some level of control is required; the design of the microprocessor architecture on basically all modern systems (except

Notes

embedded systems) offers several levels of control - the (low privilege) level of which normal applications execute limits the address space of the program to not be able to access nor modify other running applications nor the operating system itself (called "protected mode" on x86), it also prevents the application from using any system devices (i.e. the frame buffer, network devices - any I/O mapped device). But obviously any normal application needs this ability, thus the operating system is introduced, it executes at the highest level of order and allows the applications to request for a service - a system call - which is implemented through hooking interrupt(s). If allowed the system enters a higher privileged state, executes a specific set of instructions which the interrupting program has no direct control over, then returns control to the former flow of execution. This concept also serves as a way to implement security.

With the development of separate operating modes with varying levels of privilege, a mechanism was needed for transferring control safely from lesser privileged modes to higher privileged modes. Less privileged code could not simply transfer control to more privileged code at any arbitrary point and with any arbitrary processor state. To allow it to do so could allow it to break security. For instance, the less privileged code could cause the higher privileged code to execute in the wrong order, or provide it with a bad stack.

System Calls for Time Management

Many operating systems provide a time profile of a program. It indicates the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

System Calls for Device Management

A program, as it is running, may need additional resources to proceed. Additional resources may be more memory, tape drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user program; otherwise, the program will have to wait until sufficient resources are available.

These types of system calls are used to manage devices.



Example: Request device, release device, read, write, get device attributes etc.



Task System calls are generally available as assembly language instructions. If I use C programming can I receive system call or not.

3.3 System Programs

Another aspect of a modern system is the collection of system programs. In the logical computer hierarchy the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.

They can be divided into these categories:

1. ***File management:*** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Notes

2. ***Status information:*** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted, and is printed to the terminal or other output device or file.

3. ***File modification:*** Several text editors may be available to create and modify the content of files stored on disk or tape.

4. ***Programming-language support:*** Compilers, assemblers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system. Some of these programs are now priced and provided separately.

5. ***Program loading and execution:*** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed also.

6. ***Communications:*** These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

Most operating systems are supplied with programs that solve common problems, or perform common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compiler compilers, plotting and statistical-analysis packages, and games. These programs are known as system utilities or application programs.

Perhaps the most important system program for an operating system is the command interpreter, the main function of which is to get and execute the next user-specified command.

Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. These commands can be implemented in two general ways. In one approach, the command interpreter itself contains the code to execute the command.



Example: A command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

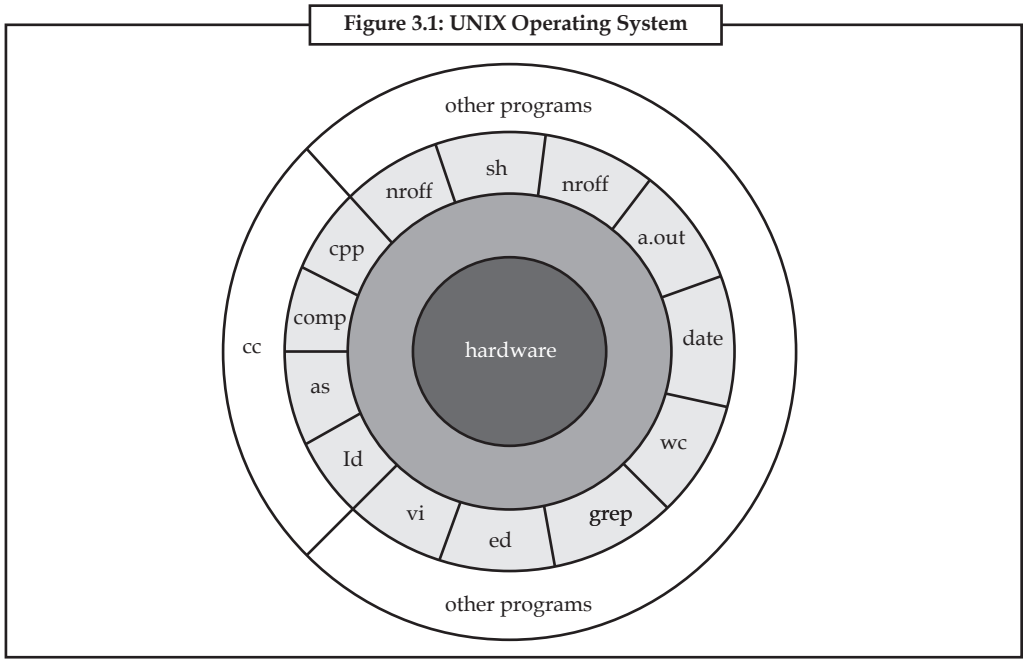
An alternative approach-used by UNIX, among other operating systems implements most commands by system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file.

3.4 Operating System Structure

The operating system structure is a container for a collection of structures for interacting with the operating system's file system, directory paths, processes, and I/O subsystem. The types and functions provided by the operating system substructures are meant to present a model for handling these resources that is largely independent of the operating system. There are different types of structure as described in Figure 3.1.

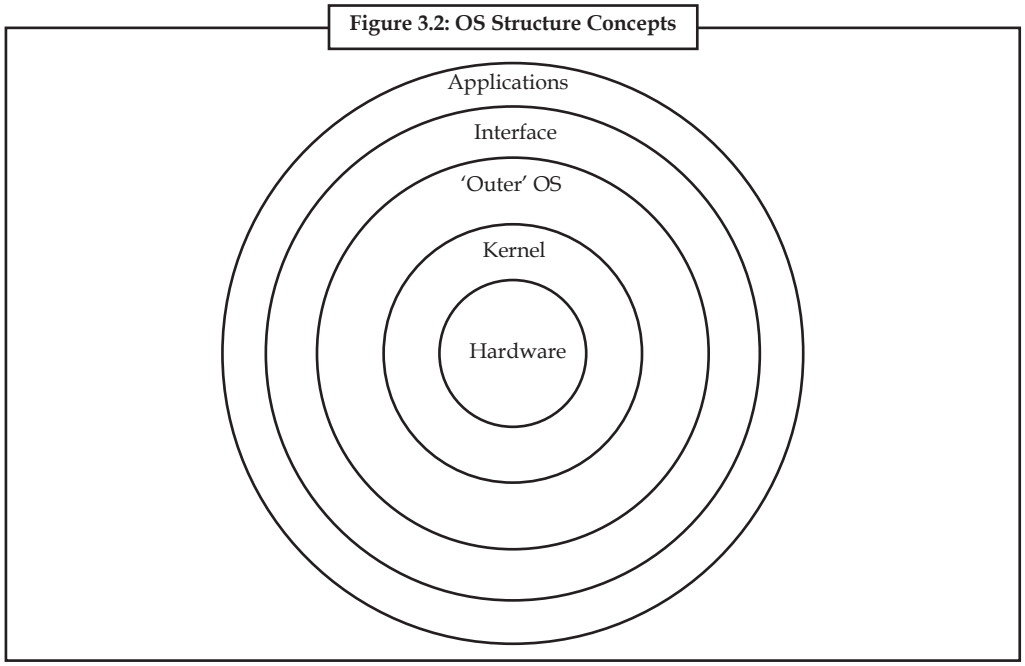
Notes

Figure 3.1: UNIX Operating System



It's very common to find pictures like Figure 3.2 below that describe the basic structure of an operating system.

Figure 3.2: OS Structure Concepts



You might find that some versions of this have different numbers of rings. What does each part represent?

1. ***Hardware:*** The hardware is, obviously, the physical hardware and not particularly interesting to us in this module.
2. ***Kernel:*** The kernel of an operating system is the bottom-most layer of software present on a machine and the only one with direct access to the hardware. The code in the kernel is

Notes

the most 'trusted' in the system - and all requests to do anything significant must go via the kernel. It provides the most key facilities and functions of the system.

3. ***Outer OS:*** Surrounding the kernel are other parts of the operating system. These perform less critical functions - for example, the graphics system which is ultimately responsible for what you see on the screen.
4. ***Interface:*** The interface provides a mechanism for you to interact with the computer.
5. ***Applications:*** There are what do the actual work - they can be complex (for example Office) or simple (for example the is command commonly found on unix and Linux systems that lists files in a directory (or folder).

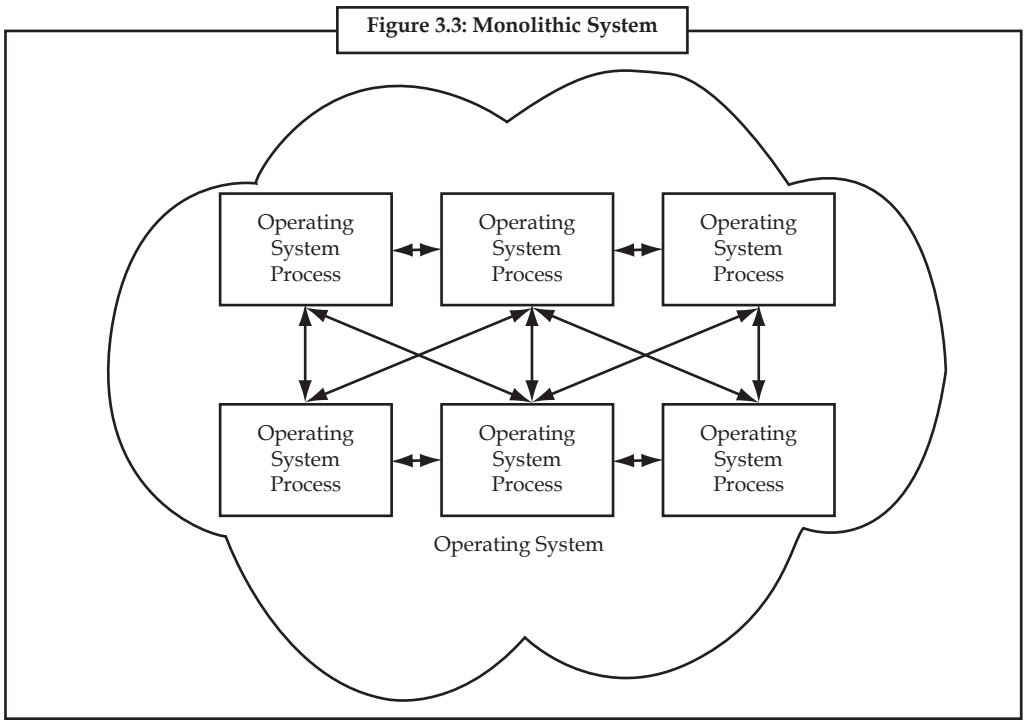
3.4.1 Monolithic Systems

This approach is well known as "The Big Mess". The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

For constructing the actual object program of the operating system when this approach is used, one compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file with the linker. In terms of information hiding, there is essentially none- every procedure is visible to every other one i.e. opposed to a structure containing modules or packages, in which much of the information is local to module, and only officially designated entry points can be called from outside the module.

However, even in Monolithic systems, it is possible to have at least a little structure. The services like system calls provide by the operating system are requested by putting the parameters in well-defined places, such as in registers or on the stack, and then executing a special trap instruction known as a kernel call or supervisor call.

Figure 3.3: Monolithic System



Notes

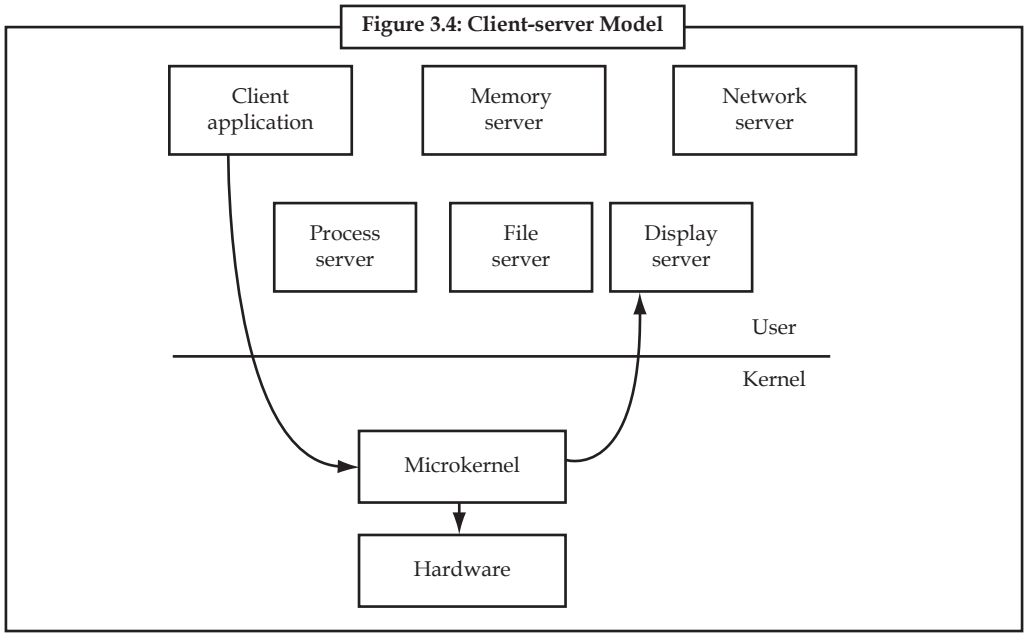
3.4.2 Client-server Model

A trend in modern operating systems is to take this idea of moving code up into higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (presently known as the client process) sends the request to a server process, which then does the work and sends back the answer.

In Client-server Model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one fact of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable; furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed system. If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

Figure 3.4: Client-server Model



3.4.3 Exokernel

Exokernel is an operating system kernel developed by the MIT Parallel and Distributed Operating Systems group, and also a class of similar operating systems.

The idea behind exokernel is to force as few abstractions as possible on developers, enabling them to make as many decisions as possible about hardware abstractions.

Applications may request specific memory addresses, disk blocks, etc. The kernel only ensures that the requested resource is free, and the application is allowed to access it. This low-level hardware access allows the programmer to implement custom abstractions, and omit unnecessary ones,

Notes

most commonly to improve a program's performance. It also allows programmers to choose what level of abstraction they want, high, or low.

Exokernels can be seen as an application of the end-to-end principle to operating systems, in that they do not force an application program to layer its abstractions on top of other abstractions that were designed with different requirements in mind.



Example: In the MIT Exokernel project, the Cheetah web server stores preformatted Internet Protocol packets on the disk, the kernel provides safe access to the disk by preventing unauthorized reading and writing, but how the disk is abstracted is up to the application or the libraries the application uses.

Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications. Traditionally, operating systems hide information about machine resources behind high-level abstractions such as processes, files, address spaces and interprocess communication. These abstractions define a virtual machine on which applications execute; their implementation cannot be replaced or modified by untrusted applications.

Hardcoding the implementations of these abstractions is inappropriate for three main reasons:

1. It denies applications the advantages of domain-specific optimizations,
2. It discourages changes to the implementations of existing abstractions, and
3. It restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

These problems can be solved through application level resource management in which traditional operating system abstractions, such as Virtual Memory (VM) and Interprocess Communication (IPC), are implemented entirely at application level by untrusted software. In this architecture, a minimal kernel-called an exokernel-securely multiplexes available hardware resources. Library operating systems, working above the exokernel interface, implement higher-level abstractions.

Application writers select libraries or implement their own. New implementations of library operating systems are incorporated by simply relinking application executables. Applications can benefit greatly from having more control over how machine resources are used to implement higher-level abstractions. The high cost of general-purpose virtual memory primitives reduces the performance of persistent stores, garbage collectors, and distributed shared memory systems. Application-level control over file caching can reduce application-running time considerably. Application-specific virtual memory policies can increase application performance. The inappropriate file-system implementation decisions can have a dramatic impact on the performance of databases. The exceptions can be made an order of magnitude faster by deferring signal handling to applications.

To provide applications control over machine resources, an exokernel defines a low-level interface. The exokernel architecture is founded on and motivated by a single, simple, and old observation that the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions.

To provide an interface that is as low-level as possible (ideally, just the hardware interface), an exokernel designer has a single overriding goal of separating protection from management. For instance, an exokernel should protect framebuffers without understanding windowing systems and disks without understanding file systems.

One approach is to give each application its own virtual machine. Virtual machines can have severe performance penalties. Therefore, an exokernel uses a different approach - it exports hardware resources rather than emulating them, which allows an efficient and simple implementation.

Notes

An exokernel employs three techniques to export resources securely:

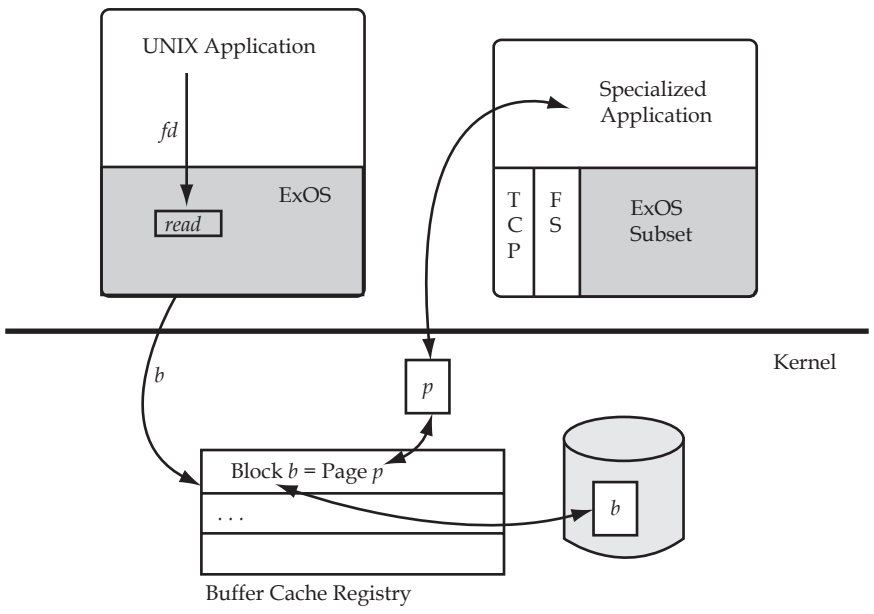
1. By using secure bindings, applications can securely bind to machine resources and handle events.
2. By using visible re-source revocation, applications participate in a resource revocation protocol.
3. By using an abort protocol, an exokernel can break secure bindings of uncooperative applications by force.

The advantages of exokernel systems among others are:

1. Exokernels can be made efficient due to the limited number of simple primitives they must provide
2. Low-level secure multiplexing of hardware resources can be provided with low overhead
3. Traditional abstractions, such as VM and IPC, can be implemented efficiently at application level, where they can be easily extended, specialized, or replaced
4. Applications can create special-purpose implementations of abstractions, tailored to their functionality and performance needs.

Finally, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or tailored to application-specific needs. These heavyweight servers can be viewed as fixed kernel subsystems that run in user-space.

Figure 3.5: Exokernel



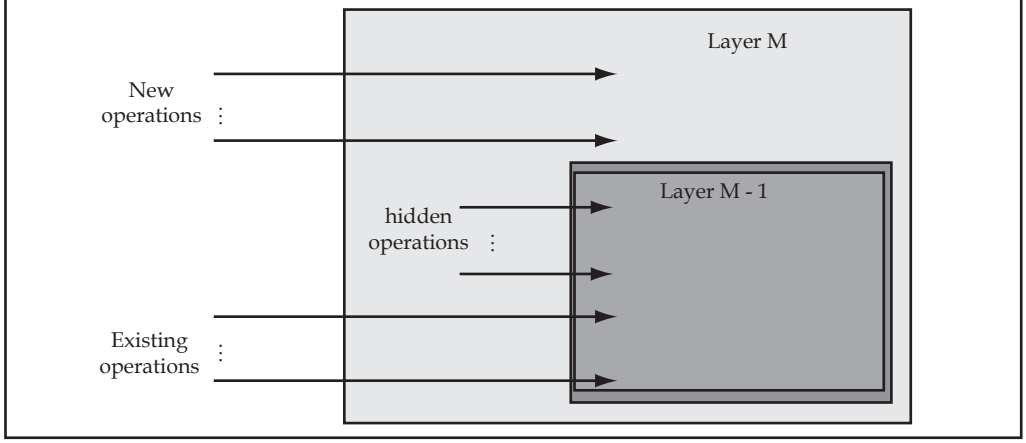
Task How application-specific virtual memory policies increase application performance? Discuss.

Notes

3.5 Layered Structure

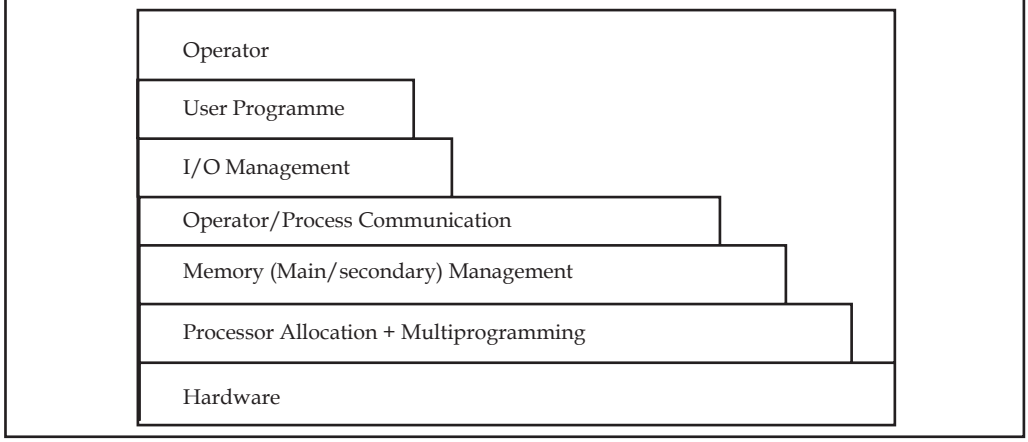
A generalization of the approach as shown in the Figure 3.6 for organizing the operating system as a hierarchy of layers, each one constructed upon the one below it.

Figure 3.6: An Operating System Layer



The system has 6 layers. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Figure 3.7: Layered System



Layer 1 did the memory management. It allocated space for processes in main memory and on a 512k word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console.

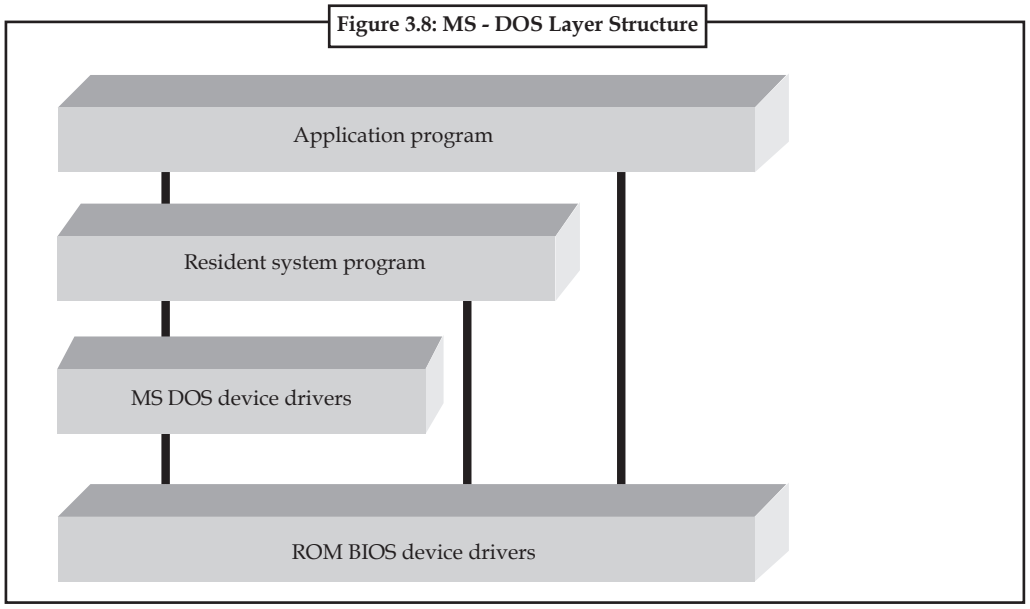
Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.

Notes

Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management.

The system operator process was located in layer 5.

Figure 3.8: MS - DOS Layer Structure



In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

The main advantage of the layered approach is modularity. The layers are selected such that each uses functions (operations) and services of only lower level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is worked on, and so on. If an error is found during the debugging of a particular layer, we know that the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified when the system is broken down into layers.

Each layer is implemented using only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The layer approach to design was first used in the operating system at the Technische Hogeschool Eindhoven. The system was defined in six layers. The bottom layer was the hardware. The next layer implemented CPU scheduling. The next layer implemented memory management; the memory-management scheme was virtual memory. Layer 3 contained device driver for the operator's console. Because it and I/O buffering (level 4) were placed above memory management, the device buffers could be placed in virtual memory. The I/O buffering was also above the operator's console, so that I/O error conditions could be output to the operator's console.

Notes

This approach can be used in many ways. For example, the Venus system was also designed using a layered approach. The lower layers (0 to 4), dealing with CPU scheduling and memory management, were then put into microcode. This decision provided the advantages of additional speed of execution and a clearly defined interface between the microcoded layers and the higher layers.

The major difficulty with the layered approach involves the appropriate definition of the various layers. Because a layer can use only those layers that are at a lower level, careful planning is necessary.



Example: The device driver for the backing store (disk space used by virtual-memory algorithms) must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the backing store.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, for a user program to execute an I/O operation, it executes a system call which is trapped to the I/O layer, which calls the memory-management layer, through to the CPU scheduling layer, and finally to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call and the net result is a system call that takes longer than one does on a non-layered system.

These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction. For instance, OS/2 is a descendant of MS-DOS that adds multitasking and dual-mode operation, as well as other new features.

Because of this added complexity and the more powerful hardware for which OS/2 was designed, the system was implemented in a more layered fashion. Contrast the MS-DOS structure to that of the OS/2. It should be clear that, from both the system-design and implementation standpoints, OS/2 has the advantage. For instance, direct user access to low-level facilities is not allowed, providing the operating system with more control over the hardware and more knowledge of which resources each user program is using.

As a further example, consider the history of Windows NT. The first release had a very layer-oriented organization. However, this version suffered low performance compared to that of Windows 95. Windows NT 4.0 redressed some of these performance issues by moving layers from user space to kernel space and more closely integrating them.

3.6 Virtual Machine

A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization. Virtualization allows the user to see the infrastructure of a network through a process of aggregation. Virtualization may also be used to run multiple operating systems at the same time. Through the help of a virtual machine, the user can operate software located on the computer platform.

There are different types of virtual machines. Most commonly, the term is used to refer to hardware virtual machine software, also known as a hypervisor or virtual machine monitor. This type of virtual machine software makes it possible to perform multiple identical executions on

Notes

one computer. In turn, each of these executions runs an operating system. This allows multiple applications to be run on different operating systems, even those they were not originally intended for.

Virtual machine can also refer to application virtual machine software. With this software, the application is isolated from the computer being used. This software is intended to be used on a number of computer platforms. This makes it unnecessary to create separate versions of the same software for different operating systems and computers. Java Virtual Machine is a very well known example of an application virtual machine.

A virtual machine can also be a virtual environment, which is also known as a virtual private server. A virtual environment is used for running programs at the user level. Therefore, it is used solely for applications and not for drivers or operating system kernels.

A virtual machine may also be a group of computers that work together to create a more powerful machine. In this type of virtual machine, the software makes it possible for one environment to be formed throughout several computers. This makes it appear to the end user as if he or she is using a single computer, when there are actually numerous computers at work.

The heart of the system, known as the virtual machine monitor, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are exact copies of the bare hardware, including kernel/user mod, I/O, interrupts, and everything else the real machine has.

Each virtual machine is identical to the true hardware; therefore, each one can run any operating system that will run directly on the hardware. Different virtual machines can, and usually do, run different operating systems. Some run one of the descendants of OF/360 for batch processing, while other ones run a single-user, interactive system called CMS (Conversational Monitor System) fro timesharing users.

Conceptually, a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system calls for use by outer layers. The systems programs above the kernel are therefore able to use either system calls or hardware instructions, and in some ways these programs do not differentiate between these two. Thus, although they are accessed differently, they both provide functionality that the program can use to create even more advanced functions. System programs, in turn, treat the hardware and the system calls as though they both are at the same level.

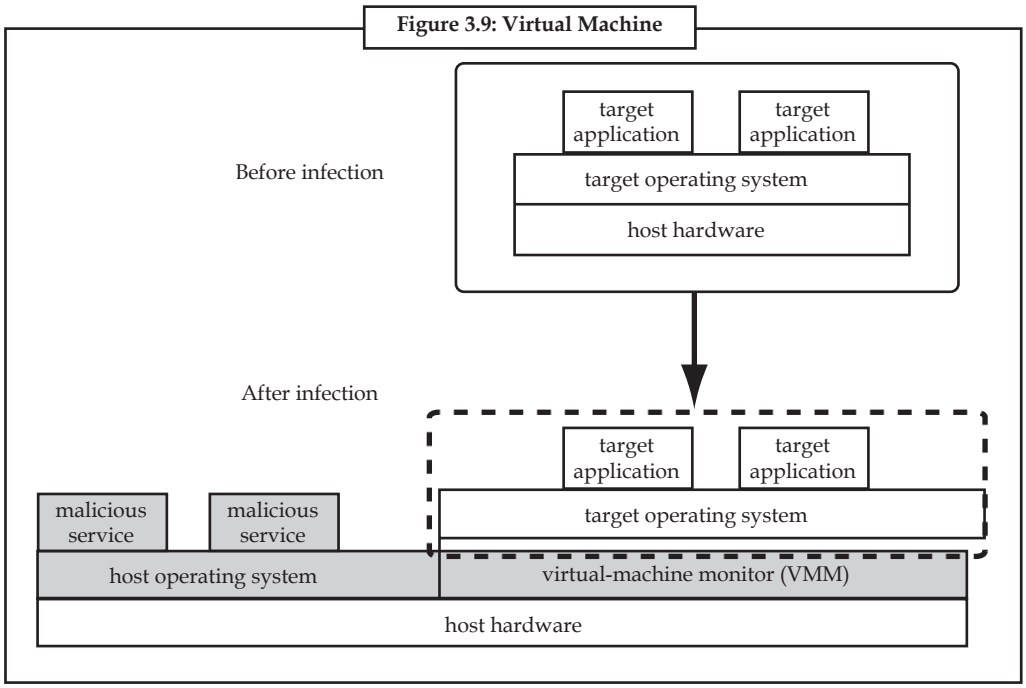
Some systems carry this scheme even a step further by allowing the system programs to be called easily by the application programs. As before, although the system programs are at a level higher than that of the other routines, the application programs may view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a virtual machine. The VM operating system for IBM systems is the best example of the virtual-machine concept, because IBM pioneered the work in this area.

By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion of multiple processes, each executing on its own processor with its own (virtual) memory. Of course, normally, the process has additional features, such as system calls and a file system, which are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional function, but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer.

The resources of the physical computer are shared to create the virtual machines. CPU scheduling can be used to share the CPU and to create the appearance that users have their own processor. Spooling and a file system can provide virtual card readers and virtual line printers. A normal user timesharing terminal provides the function of the virtual machine operator's console.

Notes

Figure 3.9: Virtual Machine



A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine. Remember that the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks, which are identical in all respects except size; these are termed minidisks in IBM's VM operating system. The system implements each minidisk by allocating as many tracks as the minidisk needs on the physical disks. Obviously, the sum of the sizes of all minidisks must be less than the actual amount of physical disk space available.

Users thus are given their own virtual machine. They can then run any of the operating systems or software packages that are available on the underlying machine. For the IBM VM system, a user normally runs CMS, a single-user interactive operating system. The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but does not need to consider any user-support software. This arrangement may provide a useful partitioning of the problem of designing a multiuser interactive system into two smaller pieces.

3.7 Summary

- The operating system provides an environment by hiding the details of underlying hardware where the user can conveniently run programs. All the user sees is that the I/O has been performed without any details.
- The output of a program may need to be written into new files or input taken from some files. It involves secondary storage management.
- The user does not have to worry about secondary storage management. There are instances where processes need to communicate with each other to exchange information.
- It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes.

Notes

- An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors.
- System calls provide an interface between the process and the operating system. These types of system calls are used to control the processes.
- A signal is a limited form of inter-process communication used in UNIX, UNIX-like, and other POSIX-compliant operating systems. The number of signals available is system dependent. File Management System Calls are used to manage files.
- Device Management System Calls are used to manage devices. System programs provide a convenient environment for program development and execution. Communications are the programs that provide the mechanism for creating virtual connections among processes, users, and different computer systems.

3.8 Keywords

Device Management System Calls: These types of system calls are used to manage devices.

Error Detection: This is a process where the operating system constantly monitors the system for detecting the malfunctioning of it.

File Management System Calls: These types of system calls are used to manage files.

File System Manipulation: Creation, deletion, modification or updation of files is known as File System Manipulation.

I/O Operations: It refers to the communication between an information processing system and the outside world - possibly a human, or another information processing system.

File: A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage.

Operating System: An operating system is itself a computer program which must be executed.

Primary-Memory: Primary-Memory or Main-Memory is a large array of words or bytes and it provides storage that can be access directly by the CPU.

Process Communication: A processes need to communicate with other process or with the user to exchange the information, this is known as Process Communication.

Process: A process is only one instant of a program in execution.

Processes Control System Calls: These types of system calls are used to control the processes.

Program Execution: Program execution is a method in which user given commands call up a processes and pass data to them.

Protection: It refers to mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.

Signal: A signal is a limited form of inter-process communication used in some operating systems.

System calls: It provides an interface between the process and the operating system.

Notes

3.9 Self Assessment

Fill in the blanks:

1. Running a program involves the ..... and ..... memory.
2. CPU scheduling is needed in case of ..... .
3. Reading from or writing to a file requires ..... .
4. System calls provide an interface between the ..... and the ..... .
5. A system call is implemented through ..... .
6. System programs provide ..... to users so that they do not need to write their own ..... for program development and ..... .
7. ..... structure is known as "The Big Mess".
8. ..... layers are there in the layered system structure.
9. Exokernel is developed by ..... .
10. In Client-server Model, all the ..... does is handle the communication between clients and servers.

State whether the following statements are true or false:

11. Users programme cannot control I/O service.
12. A process needs to communicate only with OS.
13. OS provides service to manage the primary memory only.

3.10 Review Questions

1. What are the differences between a programme and a process? Explain your answer with example.
2. Explain process management briefly.
3. What are the differences between primary storage and secondary storage?
4. Write a short notes on file management and I/O system management.
5. Do you think a single user system requires process communication? Support your answer with logic.
6. Suppose a user program faced an error during memory access. What will it do then? Will it be informed to the OS? Explain.
7. Define command interpreter. Describe its role in operating system.
8. What is signal? How a program can respond to signals?
9. How information (parameters) is passed between a running program and the operating system?
10. What is protected mode? How is it related to the operating system?

Answers: Self Assessment

Notes

1. allocating, deallocating
2. multiprocess
3. I/O service
4. process, operating system
5. hooking interrupt(s)
6. basic functioning, environment, execution (shells)
7. Monolithic Systems
8. Six
9. MIT
10. kernel
11. True
12. False
13. False

3.11 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 4: Process Management

CONTENTS

Objectives

Introduction

4.1 Process Concepts

4.2 PCB (Process Control Blocks)

4.3 Operation on Processes

4.3.1 Processes Creation

4.3.2 Process State Transitions

4.3.3 Process Termination

4.4 Cooperating Processes

4.5 Inter-process Communication

4.6 Process Communication in Client-Server Environment

4.7 Concept of Thread

4.8 User Level and Kernel Level Threads

4.9 Multi-threading

4.10 Thread Libraries

4.11 Threading Issues

4.12 Processes vs. Threads

4.13 Benefits of Threads

4.14 Summary

4.15 Keywords

4.16 Self Assessment

4.17 Review Questions

4.18 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain process concepts
- Define PCB
- Describe operation on processes
- Explain inter-process communication
- Describe concept of thread

Notes

Introduction

Earlier a computer was used to be fasten the jobs pertaining to computation diligently and incessantly for a single person. Soon it was realized that the computer was far more powerful than just carrying out a single man's single job. Such was the speed of operation that the CPU would sit idle for most of the time awaiting user input. The CPU was certainly capable of carrying out many jobs simultaneously. It could also support many users simultaneously. But, the operating systems then available were not capable of this support. The operating systems facilitating a single-user support at a time was felt inadequate. Then a mechanism was developed which would prevent the wastage of CPU cycles. Hence multi-tasking systems were developed.

In a multi-tasking system a job or task is submitted as what is known as a process. Multi-tasking operating systems could handle multiple processes on a single processor.

Process is a unit of program execution that enables the systems to implement multi-tasking behavior. Most of the operating systems today have multi-processing capabilities. This unit is dedicated to process and process related issues.

In this unit, present and discuss the mechanisms that support or enforce more structured forms of interprocess communications. Subsequent sections are devoted to messages, an extremely versatile and popular mechanism in both centralized and distributed systems, and to facilitate interprocess communication and synchronization.

4.1 Process Concepts

An operating system manages each hardware resource attached with the computer by representing it as an abstraction. An abstraction hides the unwanted details from the users and programmers allowing them to have a view of the resources in the form, which is convenient to them. A process is an abstract model of a sequential program in execution. The operating system can schedule a process as a unit of work.

The term "process" was first used by the designers of the MULTICS in 1960's. Since then, the term "process" is used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions as mentioned below:

1. A program in Execution.
2. An asynchronous activity.
3. The 'animated spirit' of a procedure in execution.
4. The entity to which processors are assigned.
5. The 'dispatchable' unit.

Though there is no universally agreed upon definition, but the definition "Program in Execution" is the one that is most frequently used. And this is a concept you will use in the present study of operating systems.

Now that you agreed upon the definition of process, the question is - what is the relation between process and program. It is same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process. Well, to be very precise. Process is not the same as program.

A process is more than a program code. A process is an 'active' entity as oppose to program which is considered to be a 'passive' entity. As you all know that a program is an algorithm expressed with the help of a programming language. A program is a passive entity sitting on some secondary storage device.

Notes

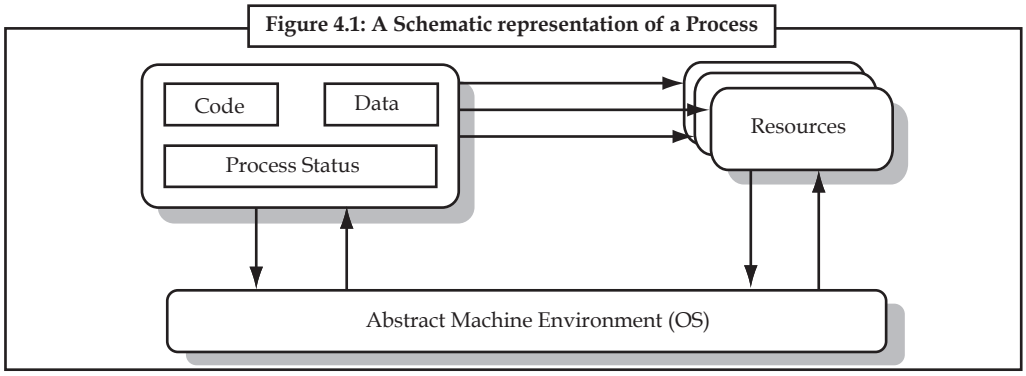
Process, on the other hand, includes:

1. Current value of Program Counter (PC)
2. Contents of the processors registers
3. Value of the variables
4. The process-stack (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
5. A data section that contains global variables.
6. A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multi-programming).

A process includes, besides instructions to be executed, the temporary data such as subroutine parameters, return addresses and variables (stored on the stack), data section having global variables (if any), program counter value, register values and other associated resources. Although two processes may be associated with the same program, yet they are treated as two separate processes having their respective set of resources.

Figure 4.1: A Schematic representation of a Process



Task Process stack contains temporary data what about program counter.

4.2 PCB (Process Control Blocks)

The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or a Process Control Block (PCB). Whenever a process is created (initialized, installed), the operating system creates a corresponding process control block to serve as its run-time description during the lifetime of the process. When the process terminates, its PCB is released to the pool of free cells from which new PCBs are drawn. The dormant state is distinguished from other states because a dormant process has no PCB. A process becomes known to the O.S. and thus eligible to compete for system resources only when it has an active PCB associate with it.

Information stored in a PCB typically includes some or all of the following:

1. Process name (ID)
2. Priority

Notes

3. State (ready, running, suspended)
4. Hardware state.
5. Scheduling information and usage statistics
6. Memory management information (registers, tables)
7. I/O Status (allocated devices, pending operations)
8. File management information
9. Accounting information.

Once constructed for a newly created process, the PCB is filled with the programmer defined attributes found in the process template or specified as the parameters of the CREATE-PROCESS operating system call. Whenever a process is suspended, the contents of the processor registers are usually saved on the stack, and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again.

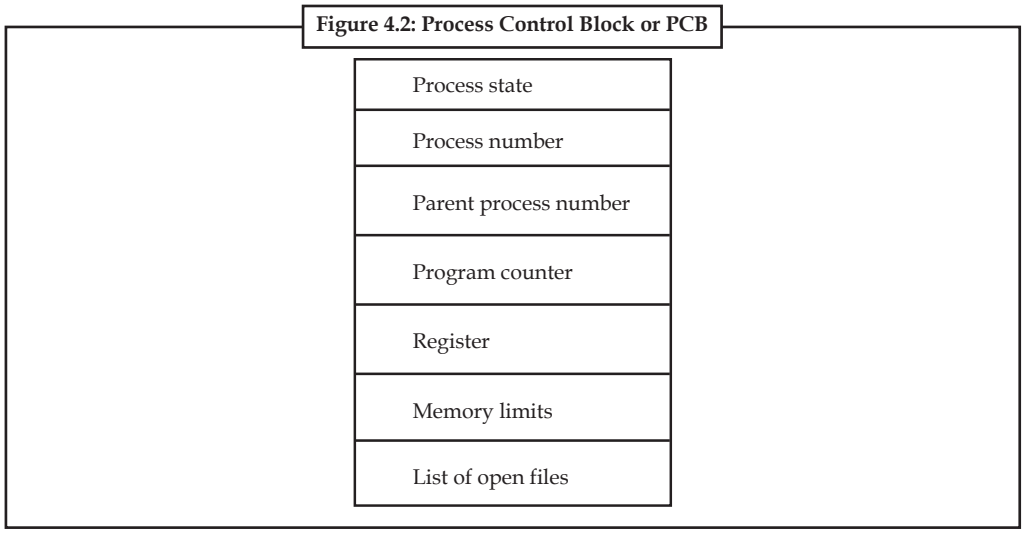
A process control block or PCB is a data structure (a table) that holds information about a process. Every process or program that runs needs a PCB. When a user requests to run a particular program, the operating system constructs a process control block for that program.

Typical information that is stored in a process control block is:

1. The location the process in memory
2. The priority of the process
3. A unique process identification number (called PID)
4. The current process state (ready, running, blocked)
5. Associated data for the process.

The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

Figure 4.2: Process Control Block or PCB



4.3 Operation on Processes

Modern operating systems, such as UNIX, execute processes concurrently. Although there is a single Central Processor (CPU), which execute the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing

Notes

a single process a few hundred microseconds of CPU time before replacing it with another process.)

Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system.

The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU: the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPUs registers, and other data), and removes the process from the use of the CPU.

The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this.

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

4.3.1 Processes Creation

The creation of a process requires the following steps. The order in which they are carried out is not necessarily the same in all cases.

1. ***Name:*** The name of the program which is to run as the new process must be known.
2. ***Process ID and Process Control Block:*** The system creates a new process control block, or locates an unused block in an array. This block is used to follow the execution of the program through its course, keeping track of its resources and priority. Each process control block is labeled by its PID or process identifier.
3. Locate the program to be executed on disk and allocate memory for the code segment in RAM.
4. Load the program into the code segment and initialize the registers of the PCB with the start address of the program and appropriate starting values for resources.
5. ***Priority:*** A priority must be computed for the process, using a default for the type of process and any value which the user specified as a `nice' value.
6. Schedule the process for execution.

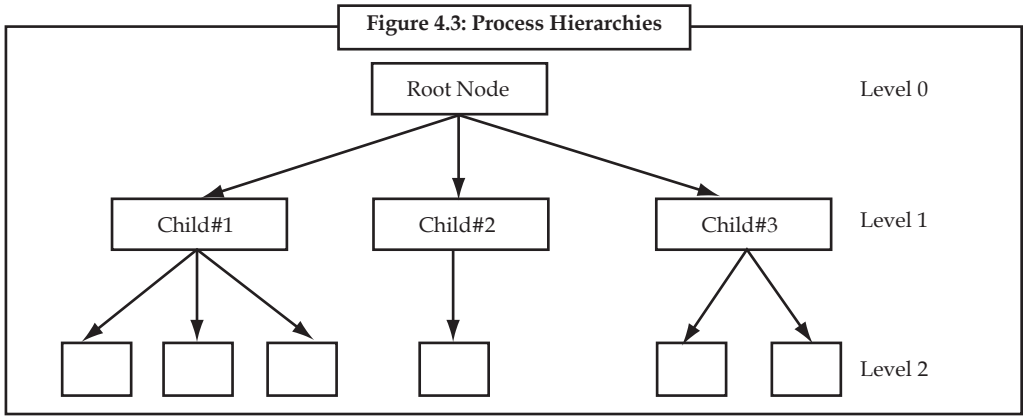
Process Hierarchy: Children and Parent Processes

In a democratic system anyone can choose to start a new process, but it is never users which create processes but other processes! That is because anyone using the system must already be running a shell or command interpreter in order to be able to talk to the system, and the command interpreter is itself a process.

When a user creates a process using the command interpreter, the new process becomes a child of the command interpreter. Similarly the command interpreter process becomes the parent for the child. Processes therefore form a hierarchy.

Notes

Figure 4.3: Process Hierarchies



The processes are linked by a tree structure. If a parent is signaled or killed, usually all its children receive the same signal or are destroyed with the parent. This doesn't have to be the case – it is possible to detach children from their parents – but in many cases it is useful for processes to be linked in this way.

When a child is created it may do one of two things.

1. Duplicate the parent process.
2. Load a completely new program.

Similarly the parent may do one of two things.

1. Continue executing along side its children.
2. Wait for some or all of its children to finish before proceeding.

The specific attributes of the child process that differ from the parent process are:

1. The child process has its own unique process ID.
2. The parent process ID of the child process is the process ID of its parent process.
3. The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes.
4. The elapsed processor times for the child process are set to zero.
5. The child doesn't inherit file locks set by the parent process.
6. The child doesn't inherit alarms set by the parent process.
7. The set of pending signals for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process.)

4.3.2 Process State Transitions

Blocking: It occurs when process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.

This state transition is:

Block: Running? Block.

Time-Run-Out: It occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

Notes

This state transition is:

Time-Run-Out: Running? Ready.

Dispatch: It occurs when all other processes have had their share and it is time for the first process to run again

This state transition is:

Dispatch: Ready? Running.

Wakeup: It occurs when the external event for which a process was waiting (such as arrival of input) happens.

This state transition is:

Wakeup: Blocked? Ready.

Admitted: It occurs when the process is created.

This state transition is:

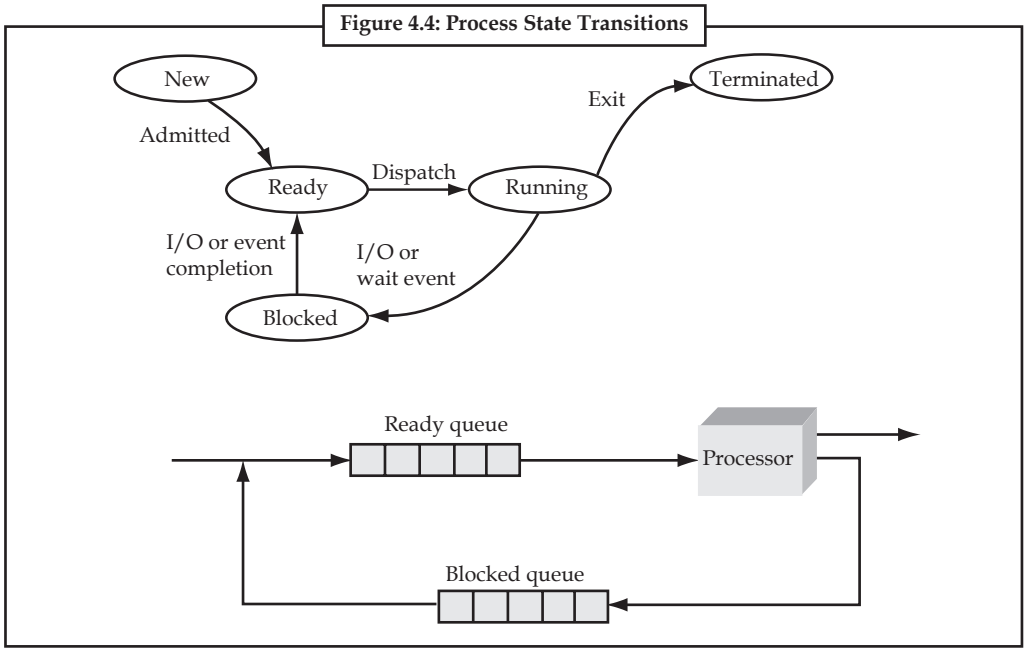
Admitted: New? Ready.

Exit: It occurs when the process has finished execution.

This state transition is:

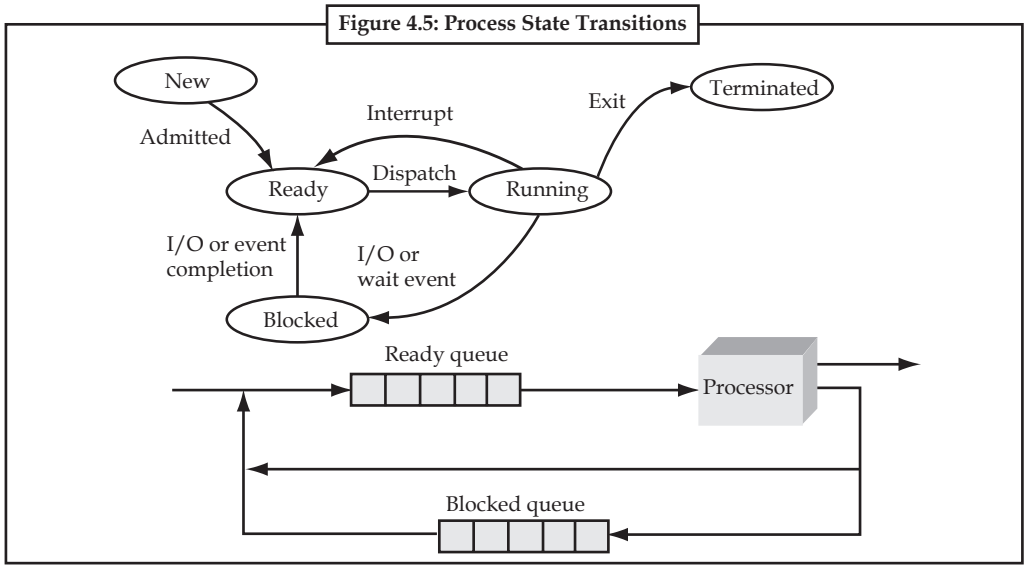
Exit: Running? Terminated.

Figure 4.4: Process State Transitions



Notes

Figure 4.5: Process State Transitions



4.3.3 Process Termination

Processes terminate in one of two ways:

1. Normal Termination occurs by a return from main or when requested by an explicit call to exit.
2. Abnormal Termination occurs as the default action of a signal or when requested by abort.
3. On receiving a signal, a process looks for a signal-handling function. Failure to find a signal-handling function forces the process to call exit, and therefore to terminate.
4. A parent may terminate the execution of one of its children for a variety of reasons, such as these:
   (a) The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
   (b) The task assigned to the child is no longer required.
   (c) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

4.4 Cooperating Processes

Concurrent processes executing in the operating system allows for the processes to cooperate (both mutually or destructively) with other processes. Processes are cooperating if they can affect each other. The simplest example of how this can happen is where two processes are using the same file. One process may be writing to a file, while another process is reading from the file; so, what is being read may be affected by what is being written. Processes cooperate by sharing data. Cooperation is important for several reasons:

Information Sharing

Several processes may need to access the same data (such as stored in a file.)

Notes

Computation Speedup

A task can often be run faster if it is broken into subtasks and distributed among different processes. For example, the matrix multiplication code you saw in class. This depends upon the processes sharing data. (Of course, real speedup also required having multiple CPUs that can be shared as well.) For another example, consider a web server which may be serving many clients. Each client can have their own process or thread helping them. This allows the server to use the operating system to distribute the computer's resources, including CPU time, among the many clients.

Modularity

It may be easier to organize a complex task into separate subtasks, and then have different processes or threads running each subtask.



Example: A single server process dedicated to a single client may have multiple threads running – each performing a different task for the client.

Convenience

An individual user can run several programs at the same time, to perform some task.



Example: A network browser is open, while the user has a remote terminal program running (such as telnet), and a word processing program editing data.

Cooperation between processes requires mechanisms that allow processes to communicate data between each other and synchronize their actions so they do not harmfully interfere with each other. The purpose of this note is to consider ways that processes can communicate data with each other, called Inter-process Communcation (IPC).



Note Another note will discuss process synchronization, and in particular, the most important means of synchronizing activity, the use of semaphores.

4.5 Inter-process Communication

Inter-process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. It involves sending information from one process to another. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and Remote Procedure Calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

Two processes might want to co-operate in performing a particular task. For example a process might want to print to document in response to a user request, so it starts another process to handle the printing and sends a message to it to start printing. Once the process handling the printing request finishes, it sends a message back to the original process, which reads the message and uses this to pop up a dialog box informing the user that the document has been printed.

There are other ways in which processes can communicate with each other, such as using a shared memory space.

Notes

Table 4.1: Inter-process Communication Methods

Method	Provided by (Operating systems or other environments)
File	All operating systems.
Signal	Most operating systems; some systems, such as Windows, only implement signals in the C run-time library and do not actually provide support for their use as an IPC technique.
Socket	Most operating systems.
Pipe	All POSIX systems.
Named pipe	All POSIX systems.
Semaphore	All POSIX systems.
Shared memory	All POSIX systems.
Message passing (shared nothing)	Used in MPI paradigm, Java RMI, CORBA and others.
memory-mapped file	All POSIX systems; may carry race condition risk if a temporary file is used. Windows also supports this technique but the APIs used are platform specific.
Message queue	Most operating systems.
Mailbox	Some operating systems.

4.6 Process Communication in Client-Server Environment

Basically the Client/Server environment is architected to split an application's processing across multiple processor to gain the maximum benefit at the least cost while minimizing the network traffic between machines. The key phase is to split the application processing. In a Client/Server mode each processing works independently but in cooperation with other processors. Each is relying on the other to perform an independent activity to complete the application process.

A good example of this would be the Mid-Range computer, normally called a File Server, which is responsible for holding the customer master file while the Client, normally the Personal Computer, is responsible for requesting an update to a specific customer. Once the Client is authenticated, the File Server is notified that the Client needs Mr. Smith's record for an update. The File Server is responsible for obtaining Mr. Smith's record and passing it to the Client for the actual modification. The Client performs the changes and then passes the changed record back to the File Server which in turn updates the master file. As in this scenario, each processor has a distinct and independent responsibility to complete the update process. The key is to perform this cooperative task while minimizing the dialog or traffic between the machines over the network. Networks have a limited capacity to carry data and if overloaded the application's response time would increase. To accomplish this goal, static processes such as edits, and menus are usually designed to reside on the Client. Update and reporting processes usually are designed to reside on the File Server. In this way, the network traffic to complete the transaction process is minimized. In addition, this design minimizes the processing cost as the Personal Computer usually is the least expensive processor, the File Server being the next expensive, and finally the Main Frame the most expensive.

There are many Client/Server Models. First, one could install all of the application's object programs on the personal computer. Secondly, one could install the static object program routines such as edits and menus on the personal computer and the business logic object programs on the file server. Thirdly, one could install all the object programs on the file server. As another option, one could install all the object programs on the mainframe. Which model you choose depends on your application design.

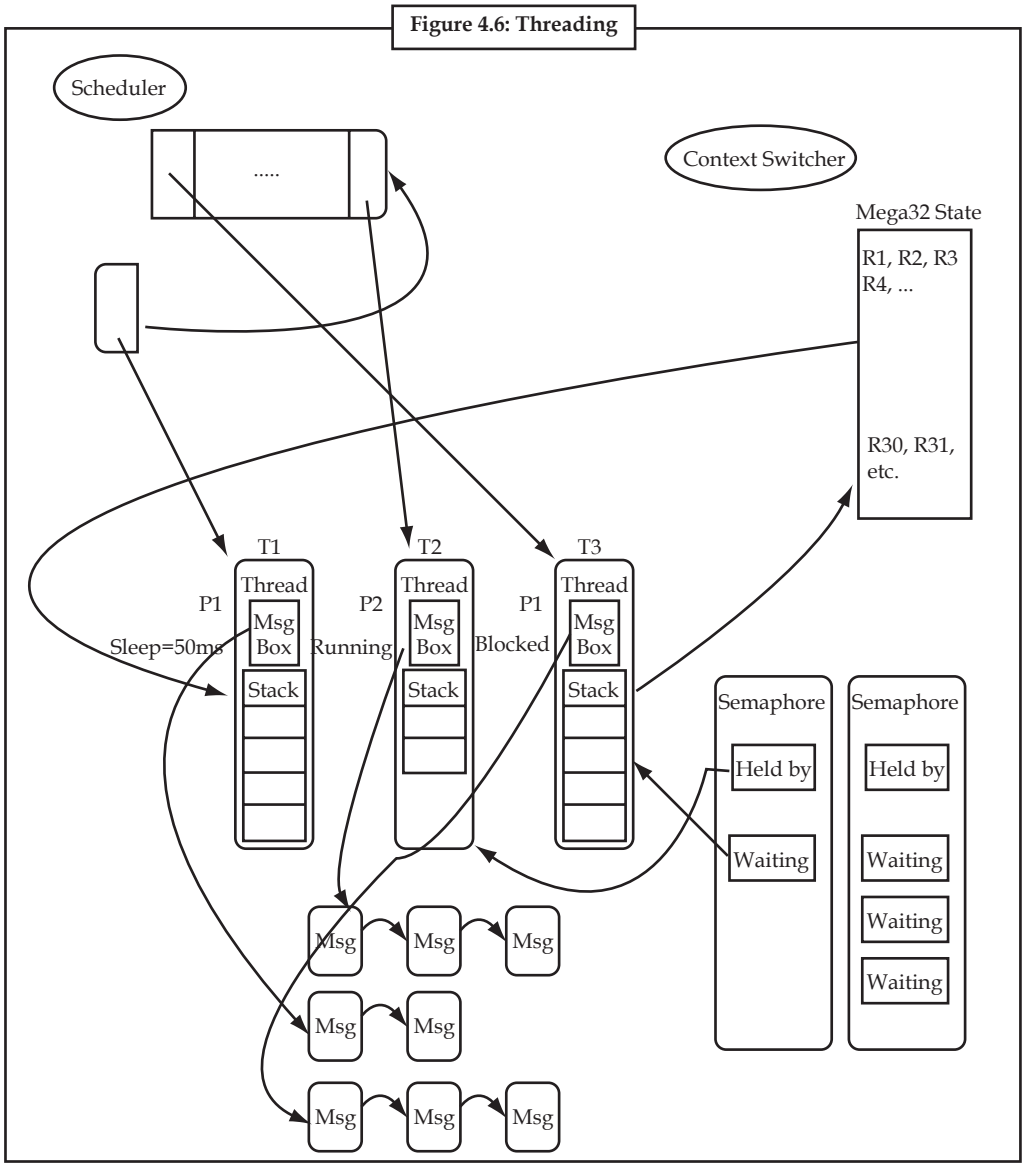
Notes

4.7 Concept of Thread

Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. 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. 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.

Figure 4.6: Threading



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. 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

Multitasking and multiprogramming, the two techniques that intend to use the computing resources optimally have been dealt with in the previous unit at length. In this unit you will learn about yet another technique that has caused remarkable improvement on the utilization of resources - thread.

A thread is a finer abstraction of a process.

Recall that a process is defined by the resources it uses and by the location at which it is executing in the memory. There are many instances, however, in which it would be useful for resources to be shared and accessed concurrently. This concept is so useful that several new operating systems are providing mechanism to support it through a thread facility.

Thread Structure

A thread, sometimes called a lightweight process (LWP), is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack. It shares with peer threads its code section, data section, and operating-system resources such as open files and signals, collectively known as a task.

A traditional or heavyweight process is equal to a task with one thread. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and the creation of threads inexpensive, compared with context switches among heavyweight processes. Although a thread context switch still requires a register set switch, no memory-management-related work need be done. Like any parallel processing environment, multithreading a process may introduce concurrency control problems that require the use of critical sections or locks.

Also, some systems implement user-level threads in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system, and to cause an interrupt to the kernel. Switching between user-level threads can be done independently of the operating system and, therefore, very quickly. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of how a server can handle many requests efficiently. User-level threads do have disadvantages, however. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

You can grasp the functionality of threads by comparing multiple-thread control with multiple-process control. With multiple processes, each process operates independently of the others; each process has its own program counter, stack register, and address space. This type of organization is useful when the jobs performed by the processes are unrelated. Multiple processes can perform the same task as well. For instance, multiple processes can provide data to remote machines in a network file system implementation.

However, it is more efficient to have one process containing multiple threads serve the same purpose. In the multiple process implementation, each process executes the same code but has its own memory and file resources. One multi-threaded process uses fewer resources than multiple redundant processes, including memory, open files and CPU scheduling, for example, as Solaris evolves, network daemons are being rewritten as kernel threads to increase greatly the performance of those network server functions.

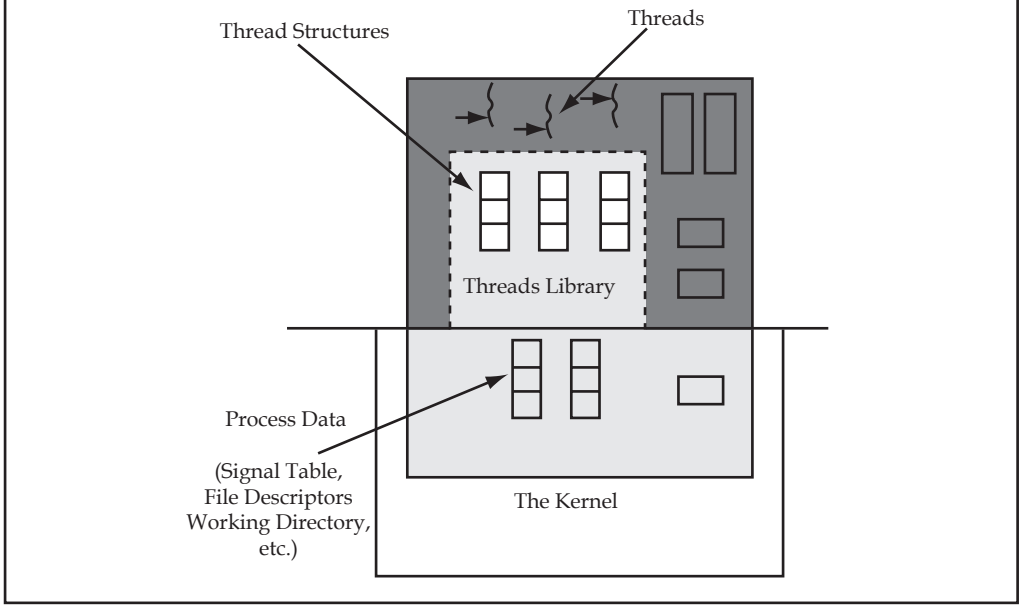
Threads operate, in many respects, in the same manner as processes. Threads can be in one of several states: ready, blocked, running, or terminated.

A thread within a process executes sequentially, and each thread has its own stack and program counter. Threads can create child threads, and can block waiting for system calls to complete; if one thread is blocked, another can run. However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write

Notes

over any other thread's stacks. This structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.

Figure 4.7: Thread Structure



Let us return to our example of the blocked file-server process in the single-process model. In this scenario, no other server process can execute until the first process is unblocked. By contrast, in the case of a task that contains multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run. In this application, the cooperation of multiple threads that are part of the same job confers the advantages of higher throughput and improved performance. Other applications, such as the producer-consumer problem, require sharing a common buffer and so also benefit from this feature of thread utilization. The producer and consumer could be threads in a task. Little overhead is needed to switch between them, and, on a multiprocessor system, they could execute in parallel on two processors for maximum efficiency.

Task Discuss something about structure of a thread.

4.8 User Level and Kernel Level Threads

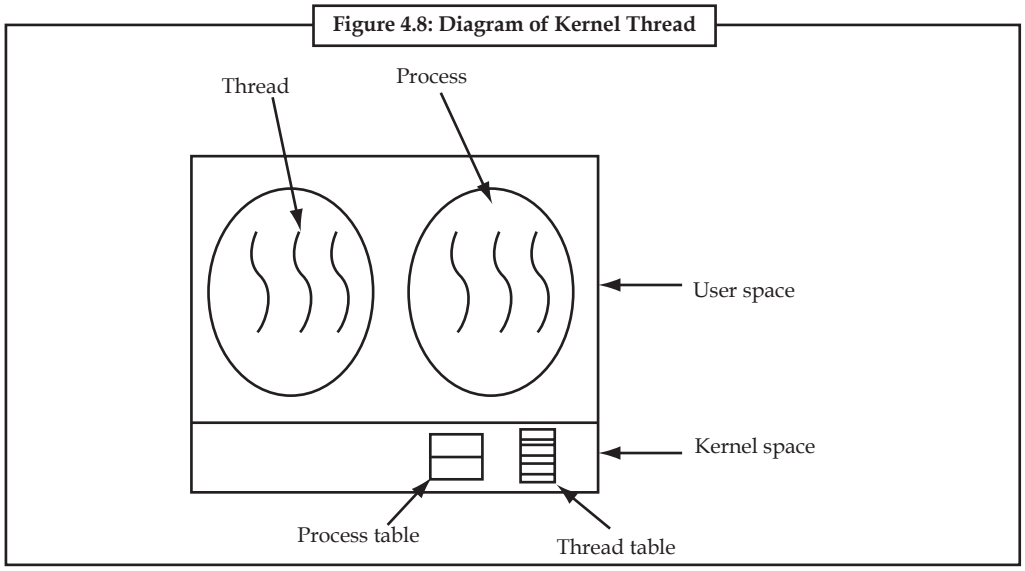
The abstraction presented by a group of lightweight processes is that of multiple threads of control associated with several shared resources. There are many alternatives regarding threads.

Threads can be supported by the kernel (as in the Mach and OS/2 operating systems). In this case, a set of system calls similar to those for processes is provided. Alternatively, they can be supported above the kernel, via a set of library calls at the user level (as is done in Project Andrew from CMU).

Notes

To implement parallel and concurrent mechanisms you need to use specific primitives of our operating system. These must have context switching capabilities, which can be implemented in two ways, using kernel level threads or using user level threads.

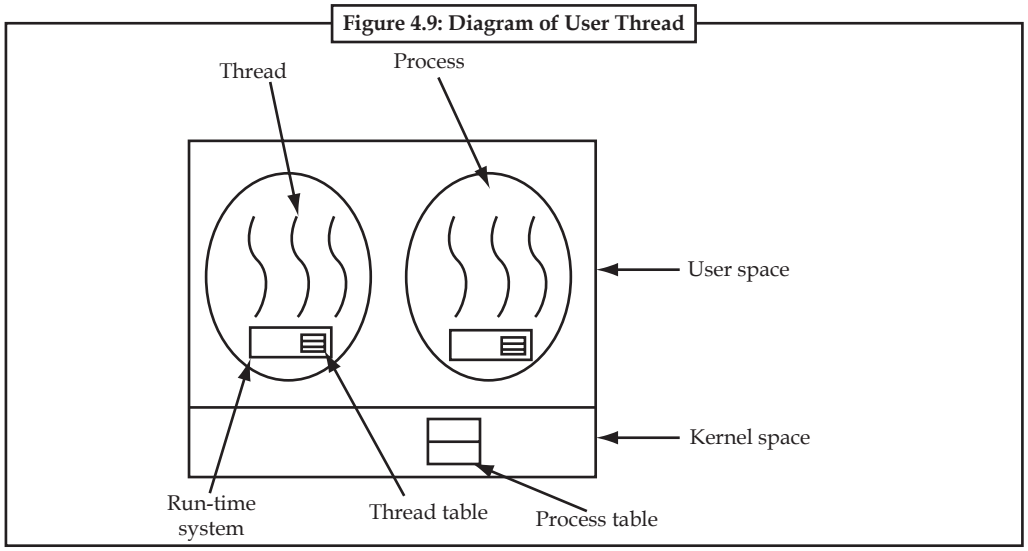
Figure 4.8: Diagram of Kernel Thread



If I use kernel level threads, the operating system will have a descriptor for each thread belonging to a process and it will schedule all the threads. This method is commonly called one to one. Each user thread corresponds to a kernel thread.

There are two major advantages around this kind of thread. The first one concerns switching aspects; when a thread finishes its instruction or is blocked, another thread can be executed. The second one is the ability of the kernel to dispatch threads of one process on several processors. These characteristics are quite interesting for multi-processor architectures. However, thread switching is done by the kernel, which decreases performances.

Figure 4.9: Diagram of User Thread

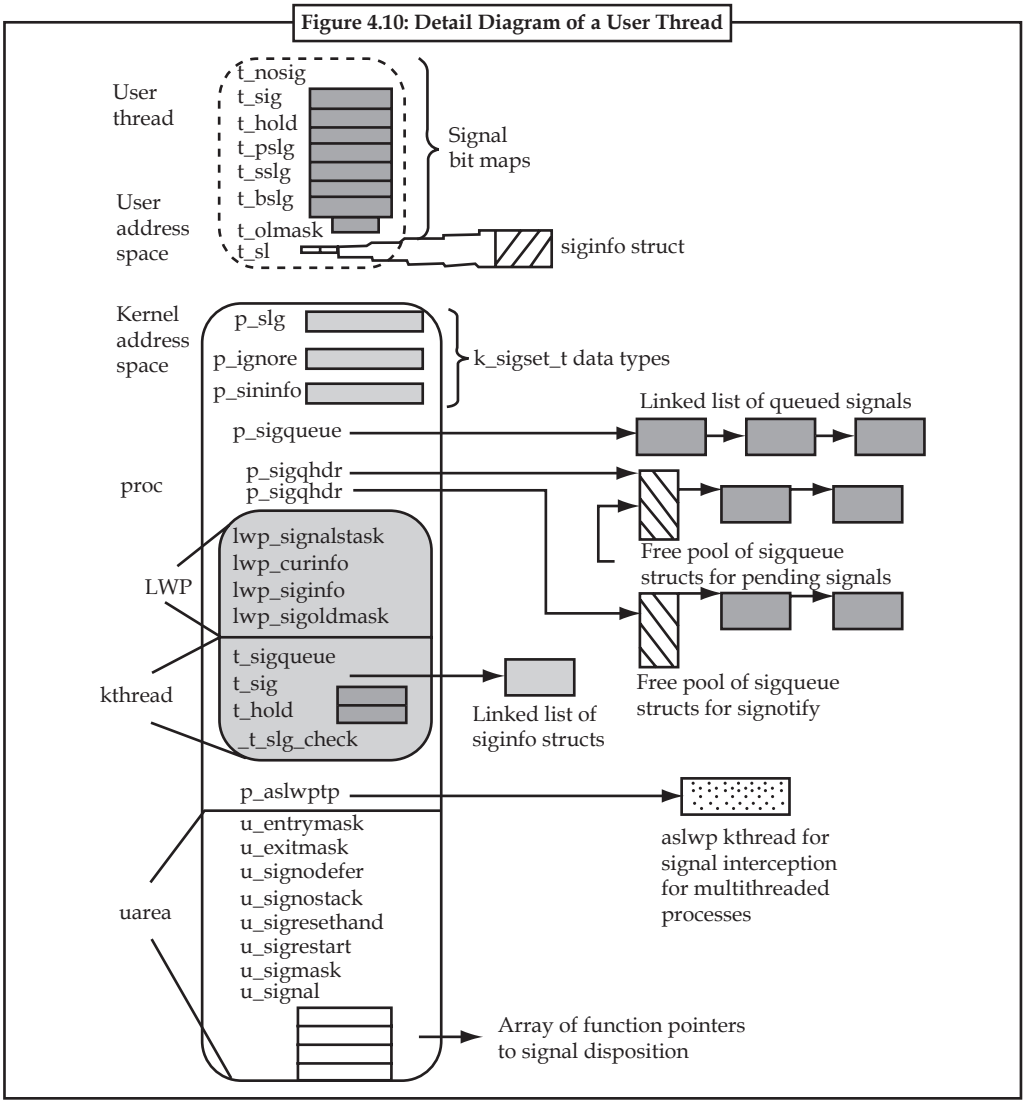


User level threads are implemented inside a specialized library that provides primitives to handle them. All information about threads is stored and managed inside the process address space. This is called many to one, because one kernel thread is associated to several user threads. Its has some advantages: The first is that is independent of the system, thus, it runs faster than context

Notes

switching at kernel level. The second comes from the scheduler that can be chosen by the user in order to manage a better thread execution. Nevertheless, if a thread of a process is jammed, all other threads of the same process are jammed too. Another disadvantage is the impossibility to execute two threads of the same process on two processors. So, user level thread is not interesting in multi-processor architectures.

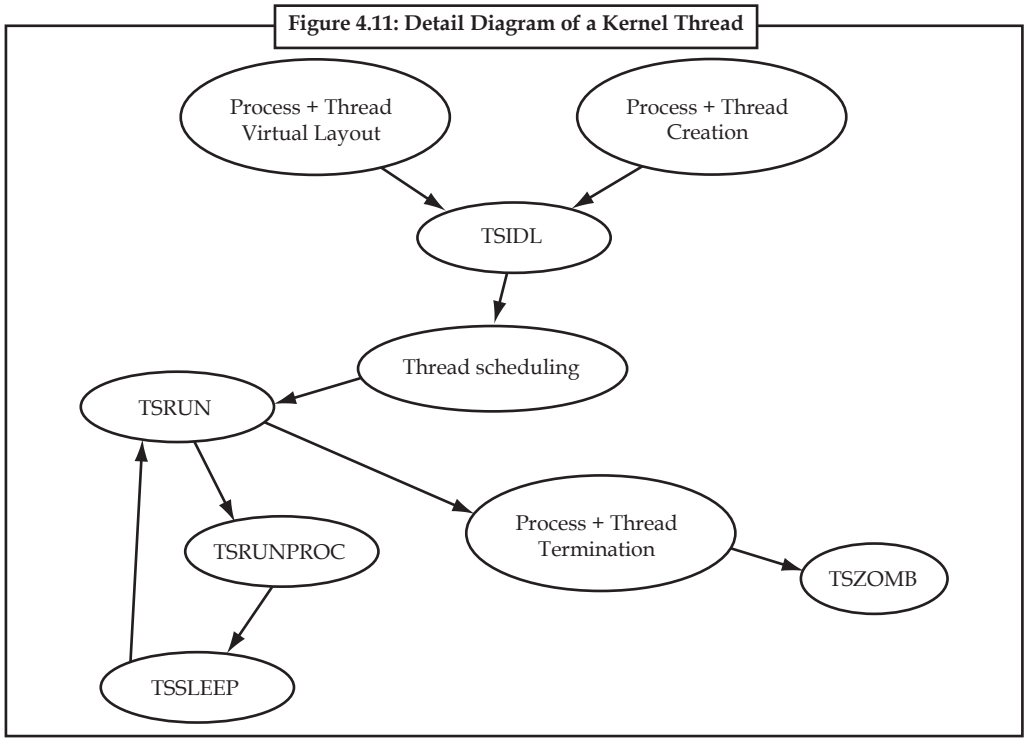
Figure 4.10: Detail Diagram of a User Thread



Why should an operating system support one version or the other? User-level threads do not involve the kernel, and therefore are faster to switch among than kernel-supported threads. However, any calls to the operating system can cause the entire process to wait, because the kernel schedules only processes (having no knowledge of threads), and a process which is waiting gets no CPU time. Scheduling can also be unfair. Consider two processes, one with 1 thread (process a) and the other with 100 threads (process b). Each process generally receives the same number of time slices, so the thread in process a runs 100 times as fast as a thread in process b. On systems with kernel-supported threads, switching among the threads is more time-consuming because the kernel (via an interrupt) must do the switch. Each thread may be scheduled independently, however, so process b could receive 100 times the CPU time that process it receives. Additionally, process b could have 100 system calls in operation concurrently, accomplishing far more than the same process would on a system with only user-level thread support.

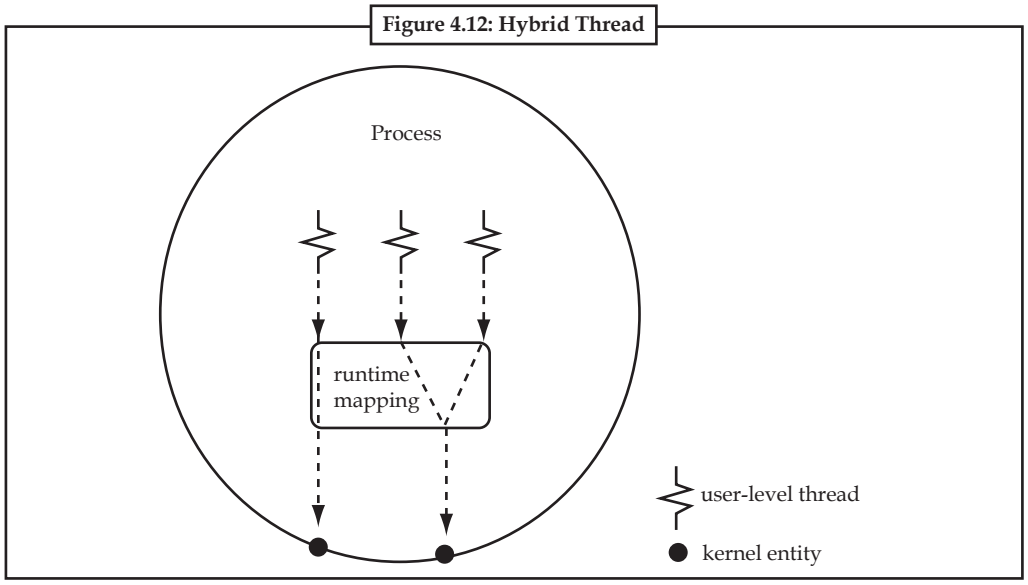
Notes

Figure 4.11: Detail Diagram of a Kernel Thread



Because of the compromises involved in each of these two approaches to threading, some systems use a hybrid approach in which both user-level and kernel-supported threads are implemented. Solaris 2 is such a system. A diagrammatic approach of hybrid thread is mentioned in Figure 4.12.

Figure 4.12: Hybrid Thread



4.9 Multi-threading

When the computers were first invented, they were capable of executing one program at a time. Thus once one program was completely executed, they then picked the second one to execute and so on. With time, the concept of timesharing was developed whereby each program was given

Notes

a specific amount of processor time and when its time got over the second program standing in queue was called upon (this is called Multi-tasking, and you would learn more about it soon). Each running program (called the process) had its own memory space, its own stack, heap and its own set of variables. One process could spawn another process, but once that occurred the two behaved independent of each other. Then the next big thing happened. The programs wanted to do more than one thing at the same time (this is called Multi-threading, and you would learn what it is soon). A browser, for example, might want to download one file in one window, while it is trying to upload another and print some other file. This ability of a program to do multiple things simultaneously is implemented through threads (detailed description on threads follows soon).

Multi-tasking vs. Multi-threading

Multi-tasking is the ability of an operating system to execute more than one program simultaneously. Though I say so but in reality no two programs on a single processor machine can be executed at the same time. The CPU switches from one program to the next so quickly that appears as if all of the programs are executing at the same time. Multi-threading is the ability of an operating system to execute the different parts of the program, called threads, simultaneously. The program has to be designed well so that the different threads do not interfere with each other. This concept helps to create scalable applications because you can add threads as and when needed. Individual programs are all isolated from each other in terms of their memory and data, but individual threads are not as they all share the same memory and data variables. Hence, implementing multi-tasking is relatively easier in an operating system than implementing multi-threading.

4.10 Thread Libraries

The threads library allows concurrent programming in Objective Caml. It provides multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space. Threads communicate by in-place modification of shared data structures, or by sending and receiving data on communication channels.

The threads library is implemented by time-sharing on a single processor. It will not take advantage of multi-processor machines. Using this library will therefore never make programs run faster. However, many programs are easier to write when structured as several communicating processes.

Two implementations of the threads library are available, depending on the capabilities of the operating system:

1. ***System threads:*** This implementation builds on the OS-provided threads facilities: POSIX 1003.1c threads for Unix, and Win32 threads for Windows. When available, system threads support both bytecode and native-code programs.
2. ***VM-level threads:*** This implementation performs time-sharing and context switching at the level of the OCaml virtual machine (bytecode interpreter). It is available on Unix systems, and supports only bytecode programs. It cannot be used with native-code programs.

Programs that use system threads must be linked as follows:

- ocamlc -thread other options unix.cma threads.cma other files
- ocamlopt -thread other options unix.cmxa threads.cmxa other files


Task POSIX 1003.1c threads for Unix, for windows which thread available.

Notes

4.11 Threading Issues

The threading issues are:

1. System calls form and exec is discussed here. In a multithreaded program environment, form and exec system calls is changed. Unix system have two version of form system calls. One call duplicates all threads and another that duplicates only the thread that invoke the form system call whether to use one or two version of form system call totally depends upon the application. Duplicating all threads in unnecessary if exec is called immediately after form system call.
2. Thread cancellation is a process of thread terminate before its completion of task.



Example: In multiple thread environment thread concurrently searching through a database. If any one thread return the result, the remaining thread might be cancelled.

3. Thread cancellation is of two types:
   (a) *Asynchronous cancellation:* One thread immediately terminates the target thread.
   (b) *Deferred cancellation:* The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not.

4.12 Processes vs. Threads

As we mentioned earlier that in many respects threads operate in the same way as that of processes. Let us point out some of the similarities and differences.

Similarities

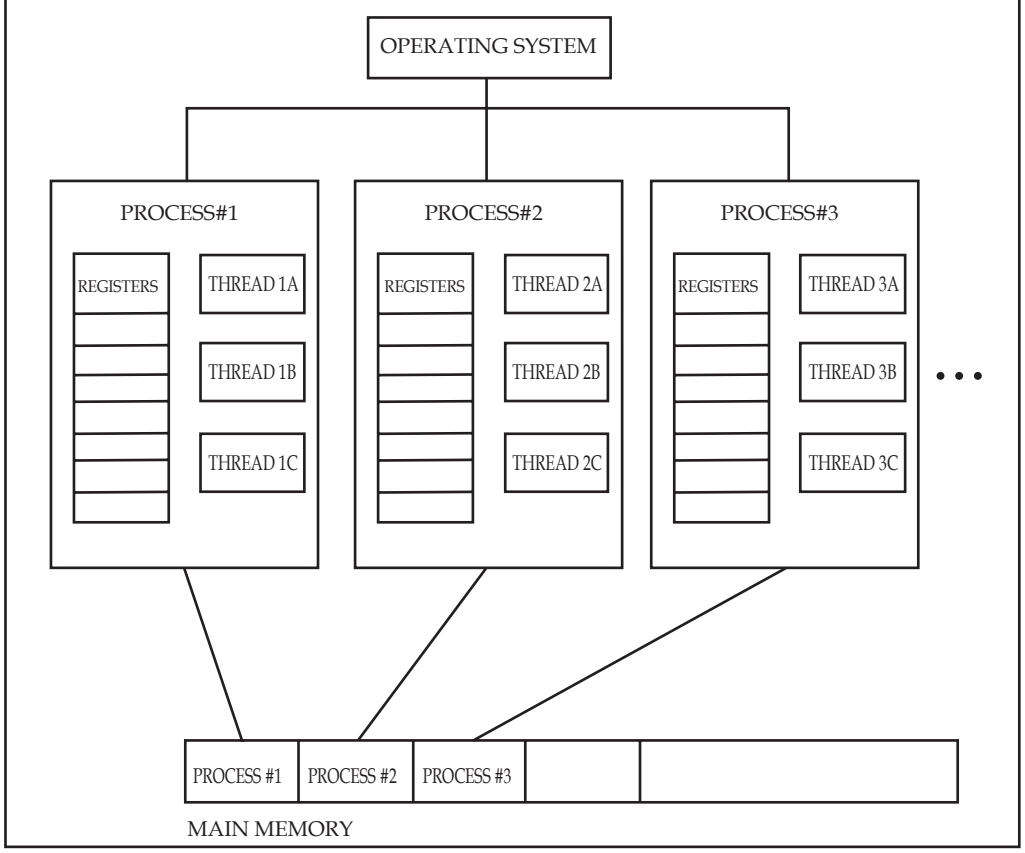
1. Like processes threads share CPU and only one thread active (running) at a time.
2. Like processes, threads within a processes, threads within a processes execute sequentially.
3. Like processes, thread can create children.
4. And like process, if one thread is blocked, another thread can run.

Differences

1. Unlike processes, threads are not independent of one another.
2. Unlike processes, all threads can access every address in the task .
3. Processes might or might not assist one another because processes may originate from different users, but threads are design to assist one other.

Notes

Figure 4.13: Diagram of Process with Multi-thread



4.13 Benefits of Threads

Following are some reasons why threads are used in designing operating systems:

1. A process with multiple threads make a great server for example printer server.
2. Because threads can share common data, they do not need to use inter-process communication.
3. Because of the very nature, threads can take advantage of multi-processors.
4. Threads need a stack and storage for registers therefore, threads are cheap to create.
5. Threads do not need new address space, global data, program code or operating system resources.

4.14 Summary

- Process management is an operating system's way of dealing with running multiple processes at once.
- A multi-tasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multi-threading or other similar technology).

Notes

- Processes are often called tasks in embedded operating systems. Process is the entity to which processors are assigned. The rapid switching back and forth of CPU among processes is called multi-programming.
- A thread is a single sequence stream within in a process. A process can have five states like created, ready, running, blocked and terminated.
- A process control block or PCB is a data structure (a table) that holds information about a process.
- Time-Run-Out occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- Dispatch occurs when all other processes have had their share and it is time for the first process to run again. Wakeup occurs when the external event for which a process was waiting (such as arrival of input) happens. Admitted occurs when the process is created. Exit occurs when the process has finished execution.

4.15 Keywords

Admitted: It is a process state transition which occurs when the process is created.

Blocking: It is a process state transition which occurs when process discovers that it cannot continue.

Dispatch: It is a process state transition which occurs when all other processes have had their share and it is time for the first process to run again.

Exit: It is a process state transition which occurs when the process has finished execution.

Multiprogramming: The rapid switching back and forth of CPU among processes is called multiprogramming.

Process control block (PCB): It is a data structure (a table) that holds information about a process.

Process management: It is an operating system's way of dealing with running multiple processes at once.

Process: It is the entity to which processors are assigned.

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

Time-Run-Out: It is a process state transition which occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

Wakeup: It is a process state transition which occurs when the external event for which a process was waiting (such as arrival of input) happens.

4.16 Self Assessment

Fill in the blanks:

1. Interrupt driven processes will normally run at a very ..... priority.
2. Processes are often called ..... in embedded operating systems.
3. The term "process" was first used by the designers of the ..... in ..... .
4. In new state, the process awaits admission to the ..... state.
5. The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or ..... .

Notes

6. ..... is a set of techniques for the exchange of data among two or more threads in one or more processes.
7. ..... are a way for a program to fork itself into two or more simultaneously running tasks.
8. ..... is the ability of an operating system to execute more than one program simultaneously.
9. The threads library is implemented by time-sharing on a ..... .
10. A process includes PC, registers, and ..... .

4.17 Review Questions

1. Do you think a single user system requires process communication? Support your answer with logic.
2. Suppose a user program faced an error during memory access. What will it do then? Will it be informed to the OS? Explain.
3. What resources are used when a thread created? How do they differ from those when a process is created?
4. What are the different process states? What is the state of the processor, when a process is waiting for some event to occur?
5. Write a brief description on process state transition.
6. What is PCB? What is the function of PCB?
7. How a process is created?
8. What is process hierarchy?
9. How a process terminated?
10. What is cooperating process? Explain it with example.
11. Why inter-process communication required?

Answers: Self Assessment

1. high priority
2. tasks
3. MULTICS, 1960's
4. ready
5. Process Control Block (PCB).
6. Inter-process Communication (IPC)
7. Threads
8. Multitasking
9. single processor
10. variables

4.18 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Published by Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.

Notes



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 5: Scheduling

CONTENTS

Objectives

Introduction

5.1 CPU Scheduling

5.2 CPU Scheduling Basic Criteria

5.3 Scheduling Algorithms

5.3.1 First-Come, First-Served (FCFS)

5.3.2 Shortest-Job-First (SJF)

5.3.3 Shortest Remaining Time (SRT)

5.3.4 Priority Scheduling

5.3.5 Round-Robin (RR)

5.3.6 Multilevel Feedback Queue Scheduling

5.3.7 Real-time Scheduling

5.3.8 Earliest Deadline First

5.3.9 Rate Monotonic

5.4 Operating Systems and Scheduling Types

5.5 Types of Scheduling

5.5.1 Long-term Scheduling

5.5.2 Medium Term Scheduling

5.5.3 Short-term Scheduling

5.6 Multiple Processor Scheduling

5.7 Thread Scheduling

5.7.1 Load Sharing

5.7.2 Gang Scheduling

5.7.3 Dedicated Processor Assignment

5.7.4 Dynamic Scheduling

5.8 Summary

5.9 Keywords

5.10 Self Assessment

5.11 Review Questions

5.12 Further Readings

Notes

Objectives

After studying this unit, you will be able to:

- Describe CPU scheduling
- Explain CPU scheduling basic criteria
- Know scheduling algorithms
- Describe types of scheduling
- Explain multiple processor scheduling
- Define thread scheduling

Introduction

CPU scheduling is the basics of multiprogramming. By switching the CPU among several processes the operating systems can make the computer more productive. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.

5.1 CPU Scheduling

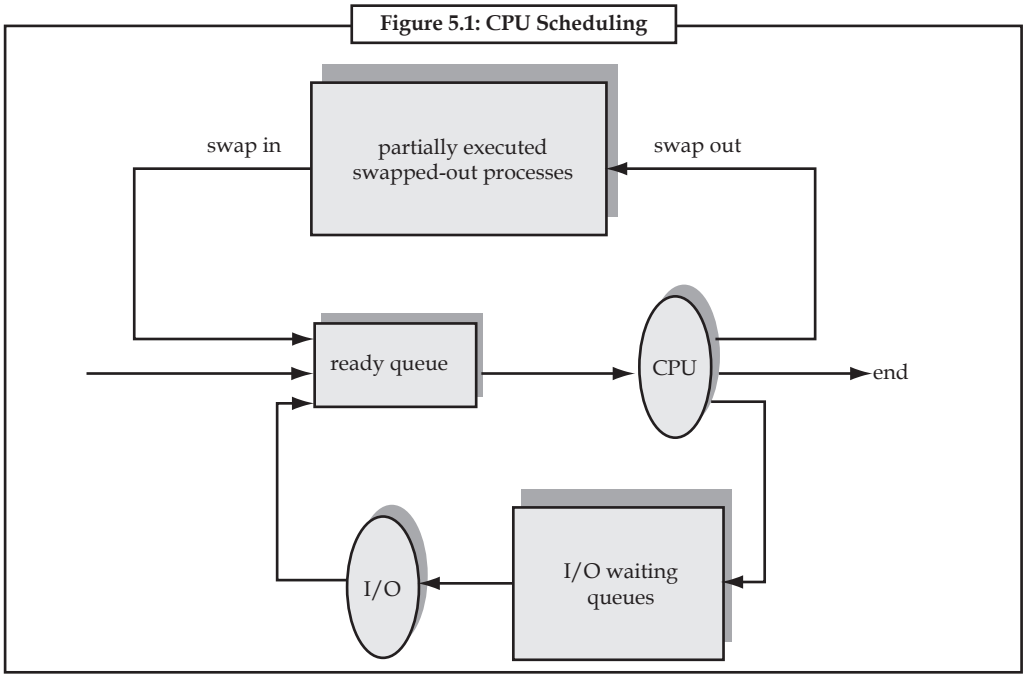
The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. The objective of time-sharing system is to switch the CPU among processes so frequently that users can interact with each program while it is running. For a uni-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

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 will contain pointers to the first and last PCBs in the list. Each PCB has a pointer field that points to the next process in the ready queue.

There are also other queues in the system. 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 an I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since there are many processes in the system, 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.

Notes

Figure 5.1: CPU Scheduling



Task Differentiate between uniprocessor and multi-processor.

Scheduling Mechanisms

A multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time-multiplexing. Part of the reason for using multiprogramming is that the operating system itself is implemented as one or more processes, so there must be a way for the operating system and application processes to share the CPU. Another main reason is the need for processes to perform I/O operations in the normal course of computation. Since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions, multiprograming systems allocate the CPU to another process whenever a process invokes an I/O operation.

Goals for Scheduling

Make sure your scheduling strategy is good enough with the following criteria:

1. ***Utilization/Efficiency:*** keep the CPU busy 100% of the time with useful work
2. ***Throughput:*** maximize the number of jobs processed per hour.
3. ***Turnaround time:*** from the time of submission to the time of completion, minimize the time batch users must wait for output
4. ***Waiting time:*** Sum of times spent in ready queue - Minimize this
5. ***Response Time:*** time from submission till the first response is produced, minimize response time for interactive users
6. ***Fairness:*** make sure each process gets a fair share of the CPU

Notes

Context Switching

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

1. Save the context of the current process
2. Select the next process to run
3. Restore the context of this new process.

Non-preemptive vs. Preemptive Scheduling

Non-preemptive

Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor).

```
context_switch() is called only when the process terminates or blocks.
```

Preemptive

Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the ready list, the process on the processor should be removed and returned to the ready list until it is once again the highest-priority process in the system.

```
context_switch() is called even when the process is running usually done via a
timer interrupt.
```

5.2 CPU Scheduling Basic Criteria

CPU scheduling is the basics of multiprogramming. By switching the CPU among several processes the operating systems can make the computer more productive. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. On systems with 1 processor, only one process may run at a time; any other processes must wait until CPU is free to be rescheduled.

In multiprogramming, a process executes until it must wait (either interrupted, or doing IO), at which point, the CPU is assigned to another process, which again, executes until it must wait, at which point another process gets the CPU, and so on.

Processes generally execute a CPU burst, followed by an IO burst, followed by the CPU burst, followed by the CPU burst, etc. This cycle is central to all processes. Every process must have CPU bursts, and every process must do some IO. The operating system maintains what is known as a ready-queue. Processes on this queue are ready to be executed. Whenever a currently executing

Notes

process needs to wait (does IO, is interrupted, etc.) the operating system picks a process from the ready queue and assigns the CPU to that process. The cycle then continues. There are many scheduling algorithms, ranging in complexity and robustness: First-come, First-serve scheduling, Shortest Job First scheduling, Round-Robin scheduling, etc.

A major task of an operating system is to manage a collection of processes. In some cases, a single process may consist of a set of individual threads.

In both situations, a system with a single CPU or a multi-processor system with fewer CPU's than processes has to divide CPU time among the different processes/threads that are competing to use it. This process is called CPU scheduling.

There are many scheduling algorithms and various criteria to judge their performance. Different algorithms may favor different types of processes. Some criteria are as follows:

1. ***CPU utilization:*** CPU must be as busy as possible in performing different tasks. CPU utilization is more important in real-time system and multi-programmed systems.
2. ***Throughput:*** The number of processes executed in a specified time period is called throughput. The throughput increases .for short processes. It decreases if the size of processes is huge.
3. ***Turnaround Time:*** The amount of time that is needed to execute a process is called turnaround time. It is the actual job time plus the waiting time.
4. ***Waiting Time:*** The amount of time the process has waited is called waiting time. It is the turnaround time minus actual job time.
5. ***Response Time:*** The amount of time between a request is Submitted and the first response is produced is called response time.

5.3 Scheduling Algorithms

Most Operating Systems today use very similar CPU time scheduling algorithms, all based on the same basic ideas, but with Operating System-specific adaptations and extensions. What follows is a description of those rough basic ideas.

What should be remarked is that this algorithm is not the best algorithm that you can imagine, but it is, proven mathematically and by experience in the early days of OS programming (sixties and seventies), the algorithm that is the closest to the 'best' algorithm. Perhaps when computers get more powerful some day then we might implement the ideal CPU time scheduler.

Another remark is that this algorithm is designed for general-purpose computers. Special-purpose Operating Systems or systems, and some real-time systems will use a very different algorithm.

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

A CPU scheduling algorithm should try to maximize the following:

1. CPU utilization
2. Throughput

A CPU scheduling algorithm should try to minimize the following:

1. Turnaround time
2. Waiting time
3. Response time

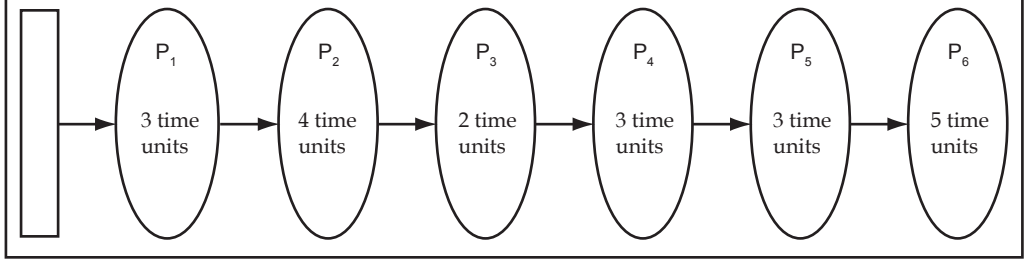
Different algorithms are used for CPU scheduling.

Notes

5.3.1 First-Come, First-Served (FCFS)

This is a Non-Preemptive scheduling algorithm. FCFS strategy assigns priority to processes in the order in which they request the processor. The process that requests the CPU first is allocated the CPU first. When a process comes in, add its PCB to the tail of ready queue. When running process terminates, dequeue the process (PCB) at head of ready queue and run it.

Figure 5.2: FCFS Scheduling



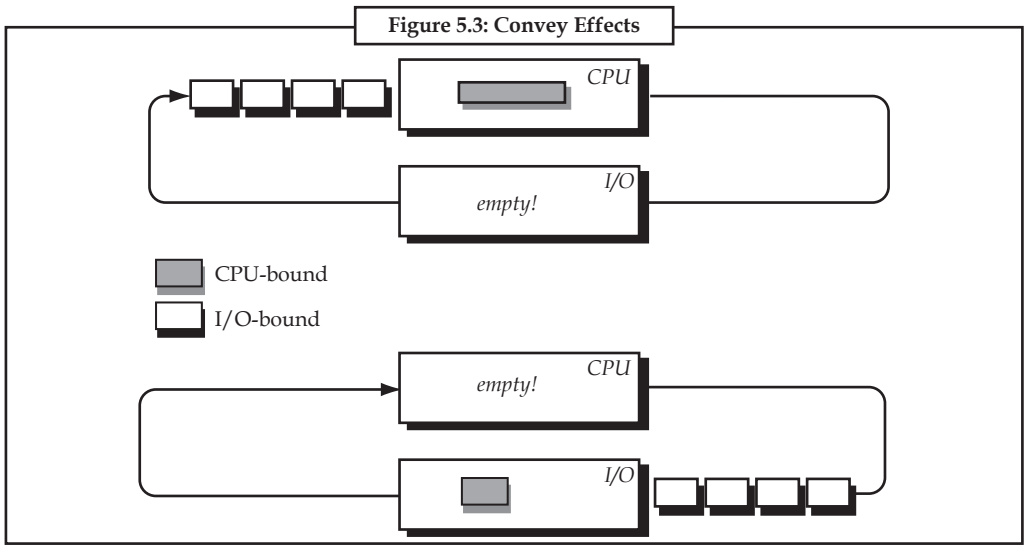
Advantage

Very simple

Disadvantages

1. Long average and worst-case waiting times
2. Poor dynamic behavior (convoy effect - short process behind long process)

Figure 5.3: Convey Effects

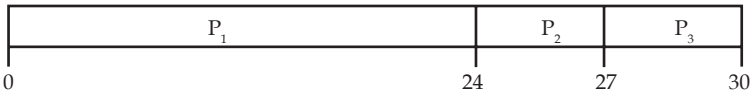


Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Notes

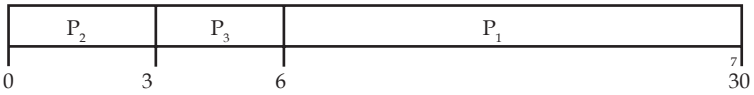
Suppose that the processes arrive in the order: P_1, P_2, P_3. The Gantt chart for the schedule is:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order P_2, P_3, P_1. The Gantt chart for the schedule is:



Waiting time for P_1 = 6; P_2 = 0; P_3 = 3

Average waiting time: (6 + 0 + 3)/3 = 3

5.3.2 Shortest-Job-First (SJF)

The SJF algorithm takes processes that use the shortest CPU time first. Mathematically seen, and corresponding to the experience, this is the ideal scheduling algorithm. I won't give details in here about the performance. It's all to do about overhead and response time, actually: the system will be fast when the scheduler doesn't take much of the CPU time itself, and when interactive processes react quickly (get a fast response). This system would be very good.

The overhead caused by the algorithm itself is huge, however. The scheduler would have top implement some way to time the CPU usage of processes and predict it, or the user should tell the scheduler how long a job (this is really a word that comes from very early computer design, when Batch Job Operating Systems were used would take. So, it is impossible to implement this algorithm without hurting performance very much.

Advantage

Minimizes average waiting times.

Disadvantages

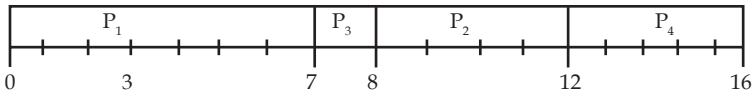
1. How to determine length of next CPU burst?
2. Starvation of jobs with long CPU bursts.

Example:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

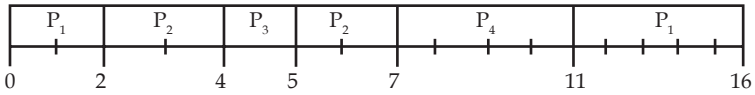
1. SJF (non-preemptive)

Notes



Average waiting time = (0 + 6 + 3 + 7)/4 = 4

2. SRT (preemptive SJB)



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

5.3.3 Shortest Remaining Time (SRT)

Shortest remaining time is a method of CPU scheduling that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added, though this threat can be minimal when process times follow a heavy-tailed distribution.

Like shortest job first scheduling, shortest remaining time scheduling is rarely used outside of specialized environments because it requires accurate estimations of the runtime of all processes that are waiting to execute.

5.3.4 Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities to compute the priority of a process.

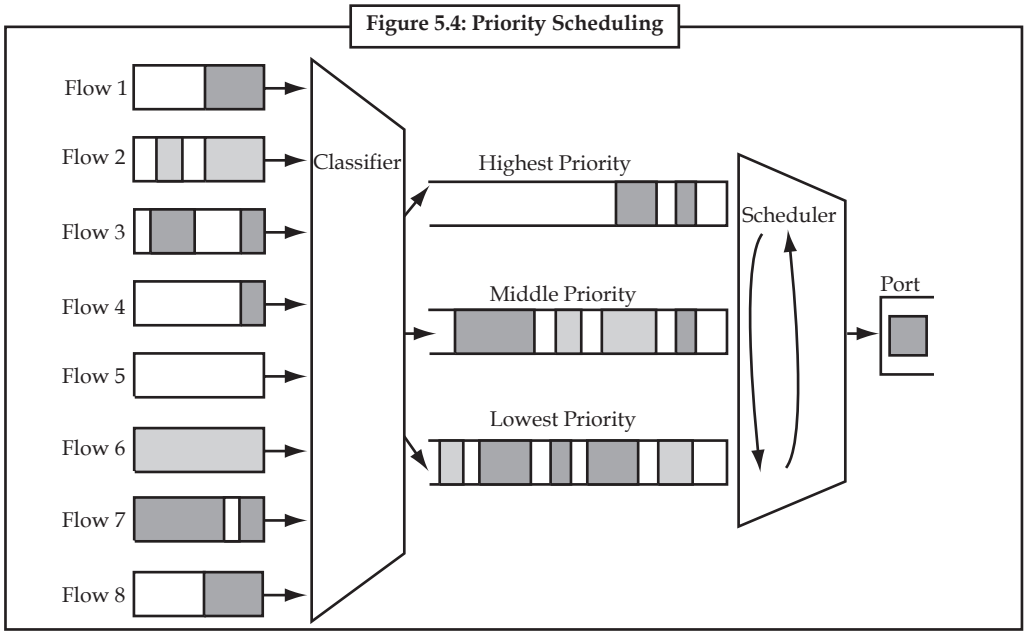
Example: Time limits, memory requirements, no. of open files, ratio of average I/O burst time to average CPU burst time etc. external priorities are set by criteria that are external to the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring work and other often political factors.

Priority scheduling can be preemptive or non preemptive. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking or starvation. This can be solved by a technique called aging wherein I gradually increase the priority of a long waiting process.

Notes

Figure 5.4: Priority Scheduling



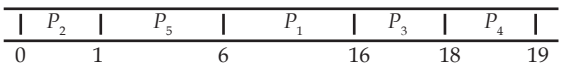
Advantages and Disadvantages

The main advantage is the important jobs can be finished earlier as much as possible. The main disadvantage is the lower priority jobs will starve.

Example:

Process	Burst time	Arrival	Priority
P_1	10	0	3
P_2	1	0	1
P_3	2	0	4
P_4	1	0	5
P_5	5	0	2

Gantt Chart



Average waiting time: (0+1+6+16+18)/5 = 8.2

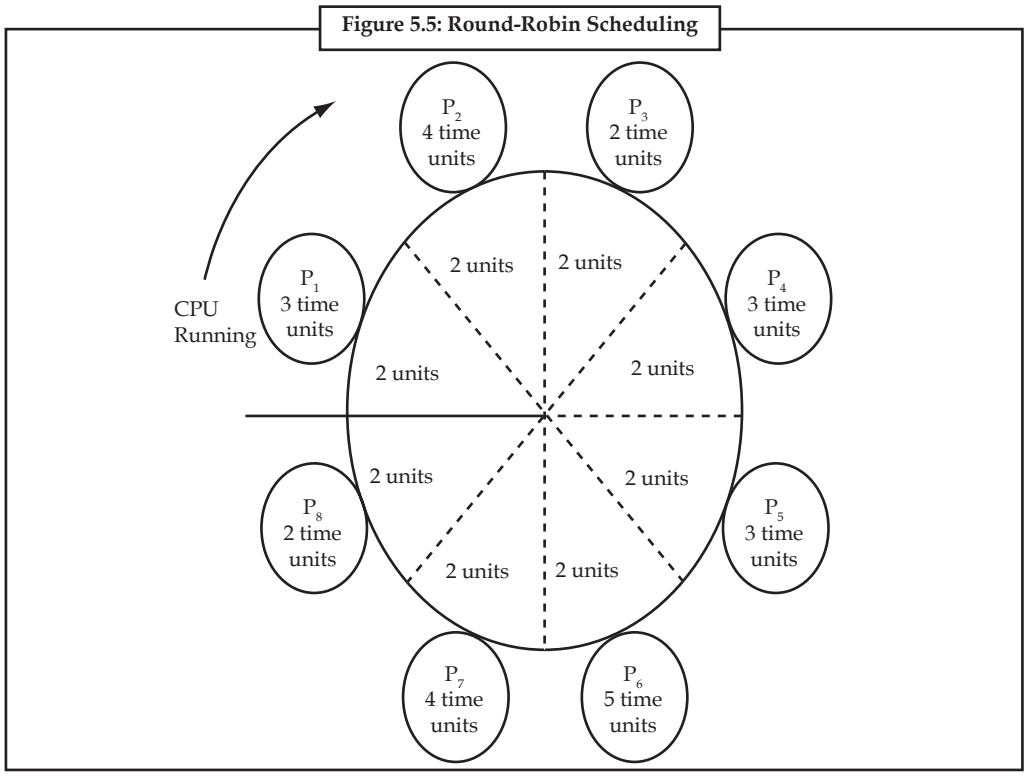
5.3.5 Round-Robin (RR)

Round-robin scheduling is really the easiest way of scheduling. All processes form a circular array and the scheduler gives control to each process at a time. It is off course very easy to implement and causes almost no overhead, when compared to all other algorithms. But response time is very low for the processes that need it. Of course it is not the algorithm I want, but it can be used eventually.

This algorithm is not the best at all for general-purpose Operating Systems, but it is useful for bath-processing Operating Systems, in which all jobs have the same priority, and in which response time is of minor or no importance. And this priority leads us to the next way of scheduling.

Notes

Figure 5.5: Round-Robin Scheduling



Advantages

Simple, low overhead, works for interactive systems

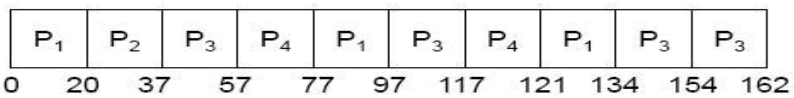
Disadvantages

1. If quantum is too small, too much time wasted in context switching
2. If too large (i.e., longer than mean CPU burst), approaches FCFS

Example: Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

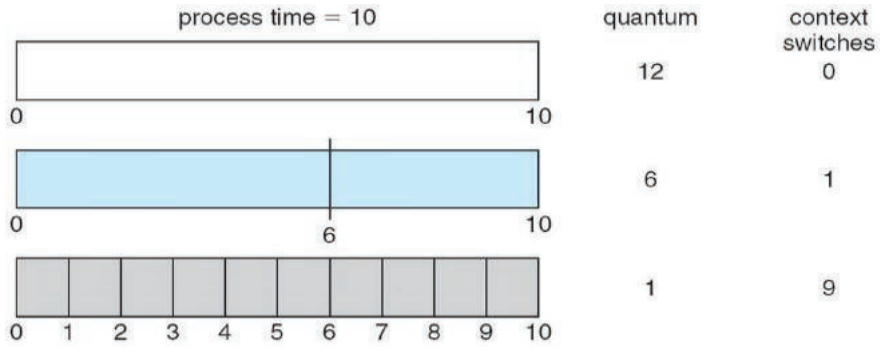
The Gantt chart is:



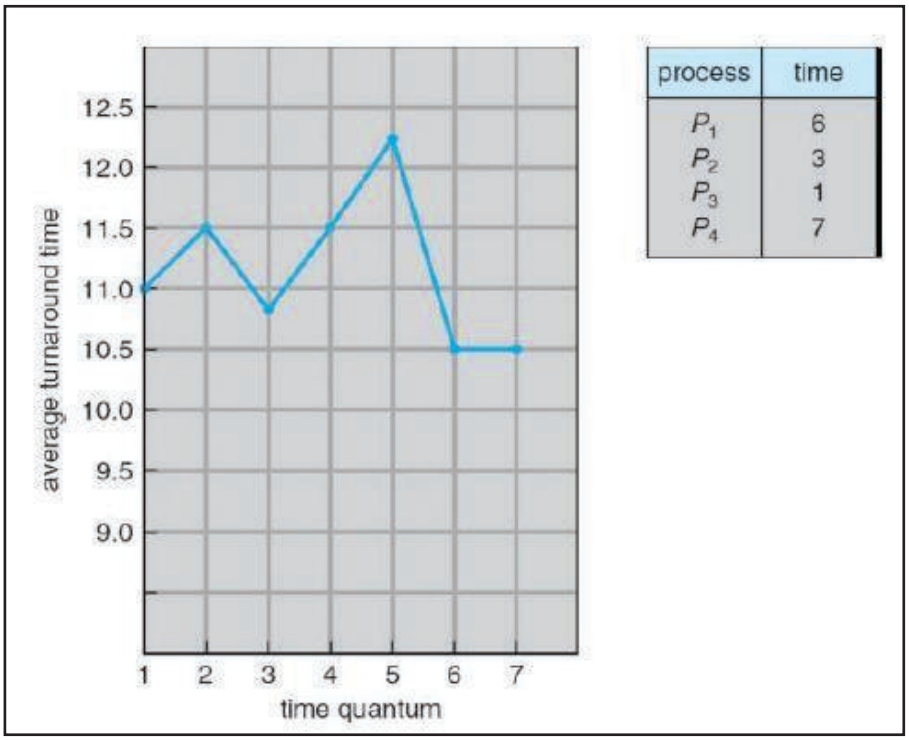
Typically, higher average turnaround than SJF, but better response

Notes

Time Quantum and Context Switch Time



Turnaround Time Varies with the Time Quantum



Example: Assume you have the following jobs to execute with one processor, with the jobs arriving in the order listed here:

i	T(pi)	Arrival Time
0	80	0
1	20	10
2	10	10
3	20	80
4	50	85

1. Suppose a system uses RR scheduling with a quantum of 15. Create a Gantt chart illustrating the execution of these processes?

2. What is the turnaround time for process p_3?

3. What is the average wait time for the processes?

Notes

Solution:

1. As the Round-Robin Scheduling follows a circular queue implementation, the Gantt chart is as follows:

p_0	p_1	p_2	p_0	p_1	p_2	p_0	p_1	p_3	p_4	p_0	p_3	p_4	p_0	p_4

0 15 30 45 60 75 85 100 110 125 140 155 160 175 190 205

2. The turnaround time for process P_3 is =160-80

   = 80 sec.

3. Average waiting time:

   Waiting time for process p_0 = 0 sec.

   Waiting time for process p_1 = 5 sec.

   Waiting time for process p_2 = 20 sec.

   Waiting time for process p_3 = 30 sec.

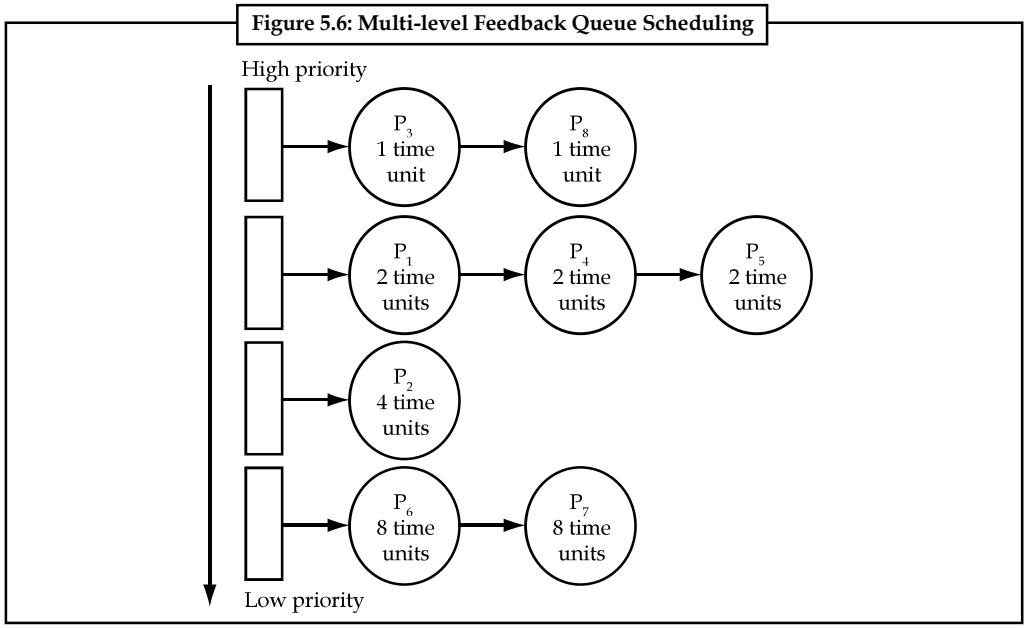
   Waiting time for process p_4 = 40 sec.

Therefore, the average waiting time is (0+5+20+30+40)/5 = 22 sec.

5.3.6 Multilevel Feedback Queue Scheduling

In this CPU schedule a process is allowed to move between queues. If a process uses too much CPU time, it will be moved to a lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queues. Similarly a process that waits too long in a lower priority queue may be moved to a higher priority queue.

Figure 5.6: Multi-level Feedback Queue Scheduling



Notes

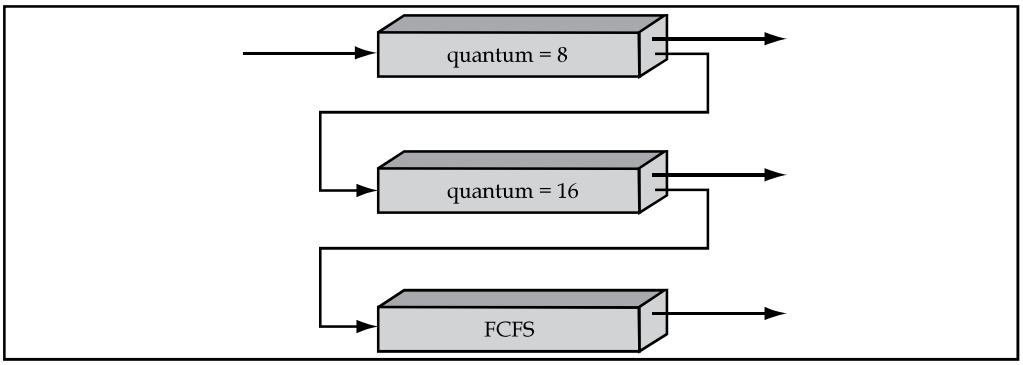
Example: Multilevel Feedback Queue

Three Queues

1. Q_0–RR with time quantum 8 milliseconds
2. Q_1–RR time quantum 16 milliseconds
3. Q_2–FCFS

Scheduling

1. A new job enters queue Q_0 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 Q_1.
2. At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



5.3.7 Real-time Scheduling

Real-time systems design is an increasingly important topic in systems research communities as well as the software industries. Real-time applications and their requirements can be found in almost every area of operating systems and networking research. An incomplete list of such domains includes distributed systems, embedded systems, network protocol processing, aircraft design, spacecraft design..., and the list goes on.

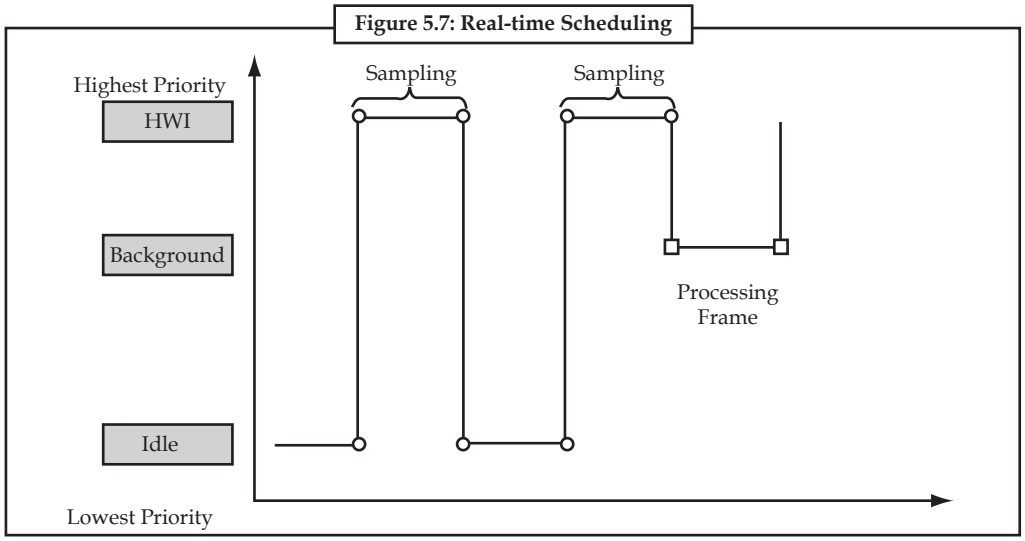
One of the most important responsibilities of a real-time system is to schedule tasks according to their deadlines in order to guarantee that all real-time activities achieve the required service level. Many scheduling algorithms exist for a variety of task models, but fundamental to many of these are the earliest deadline first (EDF) and rate-monotonic (RM) scheduling policies.

A schedule for a set of tasks is said to be feasible if a proof exists that every task instance in the set will complete processing by its associated deadline. Also, a task set is schedulable if there exists a feasible schedule for the set.

The utilization associated with a given task schedule and resource (i.e. CPU) is the fraction of time that the resource is allocated over the time that the scheduler is active.

Notes

Figure 5.7: Real-time Scheduling



5.3.8 Earliest Deadline First

The EDF scheduling algorithm is a preemptive and dynamic priority scheduler that executes tasks in order of the time remaining before their corresponding deadlines. Tasks with the least time remaining before their deadline are executed before tasks with more remaining time. At each invocation of the scheduler, the remaining time is calculated for each waiting task, and the task with the least remaining time is dispatched. If a task set is schedulable, the EDF algorithm results in a schedule that achieves optimal resource utilization. However, EDF is shown to be unpredictable if the required utilization exceeds 100%, known as an overload condition. EDF is useful for scheduling periodic tasks, since the dynamic priorities of tasks do not depend on the determinism of request periods.

5.3.9 Rate Monotonic

Under the static-priority rate monotonic scheduling algorithm, tasks are ordered according to the value of their request period, T. Tasks with shorter request periods are assigned higher priority than those with longer periods. Liu and Layland proved that a feasible schedule is found under rate monotonic scheduling if the total requested utilization is less than or equal to ln, which is approximately 69.3%.

RM scheduling is optimal with respect to maximum utilization over all static-priority schedulers. However, this scheduling policy only supports tasks that fit the periodic task model, since priorities depend upon request periods. Because the request times of a periodic tasks are not always predictable, these tasks are not supported by the RM algorithm, but are instead typically scheduled using a dynamic priority scheduler such as EDF.

Characteristics of Scheduling Algorithms

	FCFS	Round Robin	SJF	SRT	HRRN	Feedback
Selection Function	max[w]	constant	Min[s]	min[s-e]	max[(w+s)/s]	see text
Decision mode	Non-preemptive	preemptive	Non-preemptive	preemptive	**Non-preemptive**	preemptive at time quantum
						Contd...

Notes

Throughput	N/A	low for small quantum	high	high	high	N/A
Response Time	May be high	good for short processes	good for short processes	good	good	N/A
Overhead	minimal	low	Can be high	can be high	can be high	can be high
Effect on Processes						
Starvation	No	No	Possible	Possible	No	Possible

w = time spent in the system so far, waiting and executing

e = time spent in execution so far.

s = total service time required by the process, including e.



Task "Priority scheduling can be preemptive or non-preemptive." Discuss.

5.4 Operating Systems and Scheduling Types

1. ***Solaris 2*** uses priority-based process scheduling.
2. ***Windows 2000*** uses a priority-based preemptive scheduling algorithm.
3. ***Linux*** provides two separate process-scheduling algorithms: one is designed for time-sharing processes for fair preemptive scheduling among multiple processes; the other designed for real-time tasks:

   (a) For processes in the time-sharing class Linux uses a prioritized credit-based algorithm

   (b) Real-time scheduling: Linux implements two real-time scheduling classes namely FCFS (First come first serve) and RR (Round Robin)

5.5 Types of Scheduling

In many multitasking systems the processor scheduling subsystem operates on three levels, differentiated by the time scale at which they perform their operations. In this sense differentiate among:

1. ***Long term scheduling:*** which determines which programs are admitted to the system for execution and when, and which ones should be exited.
2. ***Medium term scheduling:*** which determines when processes are to be suspended and resumed;
3. ***Short term scheduling (or dispatching):*** which determines which of the ready processes can have CPU resources, and for how long.

5.5.1 Long-term Scheduling

Long term scheduling obviously controls the degree of multiprogramming in multitasking systems, following certain policies to decide whether the system can honour a new job submission or, if more than one job is submitted, which of them should be selected. The need for some form of compromise between degree of multiprogramming and throughput seems evident, especially when one considers interactive systems. The higher the number of processes, in fact, the smaller

Notes

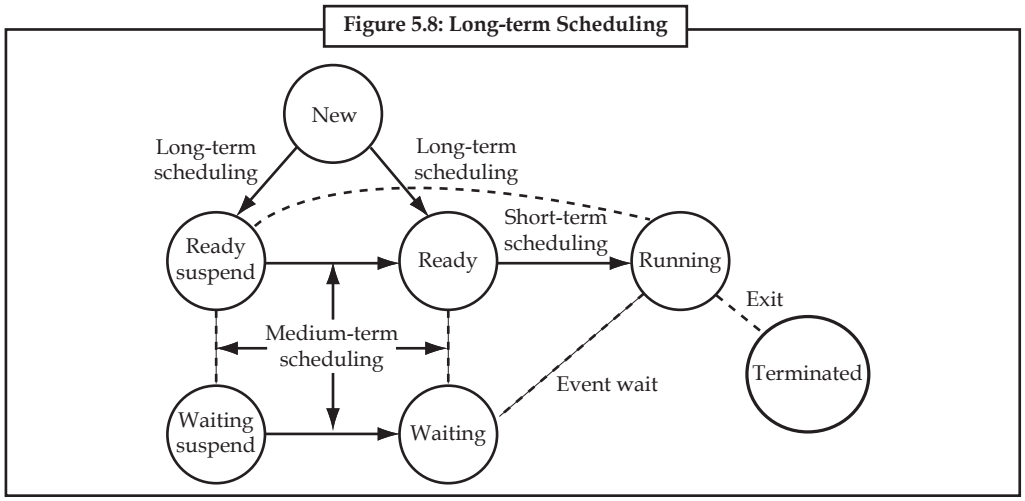
the time each of them may control CPU for, if a fair share of responsiveness is to be given to all processes. Moreover you have already seen that a too high number of processes causes waste of CPU time for system housekeeping chores (trashing in virtual memory systems is a particularly nasty example of this). However, the number of active processes should be high enough to keep the CPU busy servicing the payload (i.e. the user processes) as much as possible, by ensuring that - on average - there always be a sufficient number of processes not waiting for I/O.

Simple policies for long term scheduling are:

1. ***Simple First Come First Served (FCFS):*** It's essentially a FIFO scheme. All job requests (e.g. a submission of a batch program, or an user trying to log in in a time shared system) are honoured up to a fixed system load limit, further requests being refused tout court, or enqueued for later processing.
2. ***Priority schemes:*** Note that in the context of long term scheduling ``priority'' has a different meaning than in dispatching: here it affects the choice of a program to be entered the system as a process, there the choice of which ready process process should be executed.

Long term scheduling is performed when a new process is created. It is shown in the figure below. If the number of ready processes in the ready queue becomes very high, then there is a overhead on the operating system (i.e., processor) for maintaining long lists, context switching and dispatching increases. Therefore, allow only limited number of processes in to the ready queue. The "long-term scheduler" managers this. Long-term scheduler determines which programs are admitted into the system for processing. Once when admit a process or job, it becomes process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler scheduling manage queues to minimize queueing delay and to optimize performance.

Figure 5.8: Long-term Scheduling



The long-term scheduler limits the number of processes to allow for processing by taking the decision to add one or more new jobs, based on FCFS (First-Come, first-serve) basis or priority or execution time or Input/Output requirements. Long-term scheduler executes relatively infrequently.

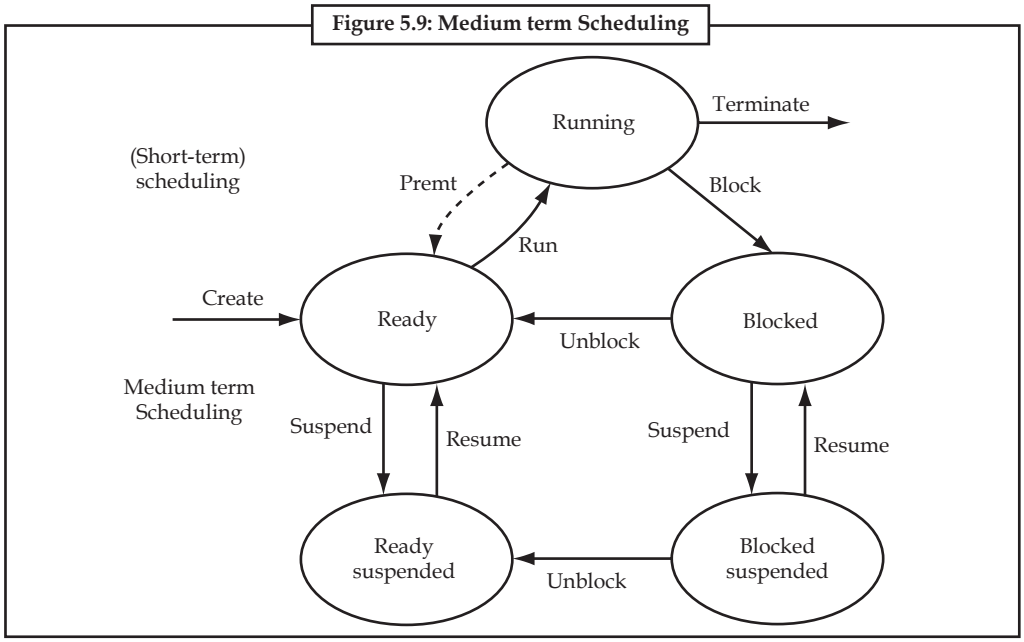
5.5.2 Medium Term Scheduling

Medium term scheduling is essentially concerned with memory management, hence it's very often designed as a part of the memory management subsystem of an OS. Its efficient interaction with the short term scheduler is essential for system performances, especially in virtual memory

Notes

systems. This is the reason why in paged system the pager process is usually run at a very high (dispatching) priority level.

Figure 5.9: Medium term Scheduling



Unblock is done by another task (a.k.a. wakeup, release, V) Block is a.k.a. sleep, request, P)

In addition to the short-term scheduling we have discussed, we add medium-term scheduling in which decisions are made at a coarser time scale.

Recall my favorite diagram, shown again on the right. Medium term scheduling determines the transitions from the top triangle to the b ottom line. We suspend (swap out) some process if memory is over-committed, dropping the (ready or blocked) process down. We also need resume transitions to return a process to the top triangle.

Criteria for choosing a victim to suspend include:

1. How long since previously suspended.
2. How much CPU time used recently.
3. How much memory does it use.
4. External priority (pay more, get swapped out less).

5.5.3 Short-term Scheduling

Short term scheduling concerns with the allocation of CPU time to processes in order to meet some pre-defined system performance objectives. The definition of these objectives (scheduling policy) is an overall system design issue, and determines the ``character'' of the operating system from the user's (i.e. the buyer's) point of view, giving rise to the traditional distinctions among ''multi-purpose, time shared'', ''batch production'', ''real-time'' systems, and so on.

From a user's point of view, the performance criteria base on:

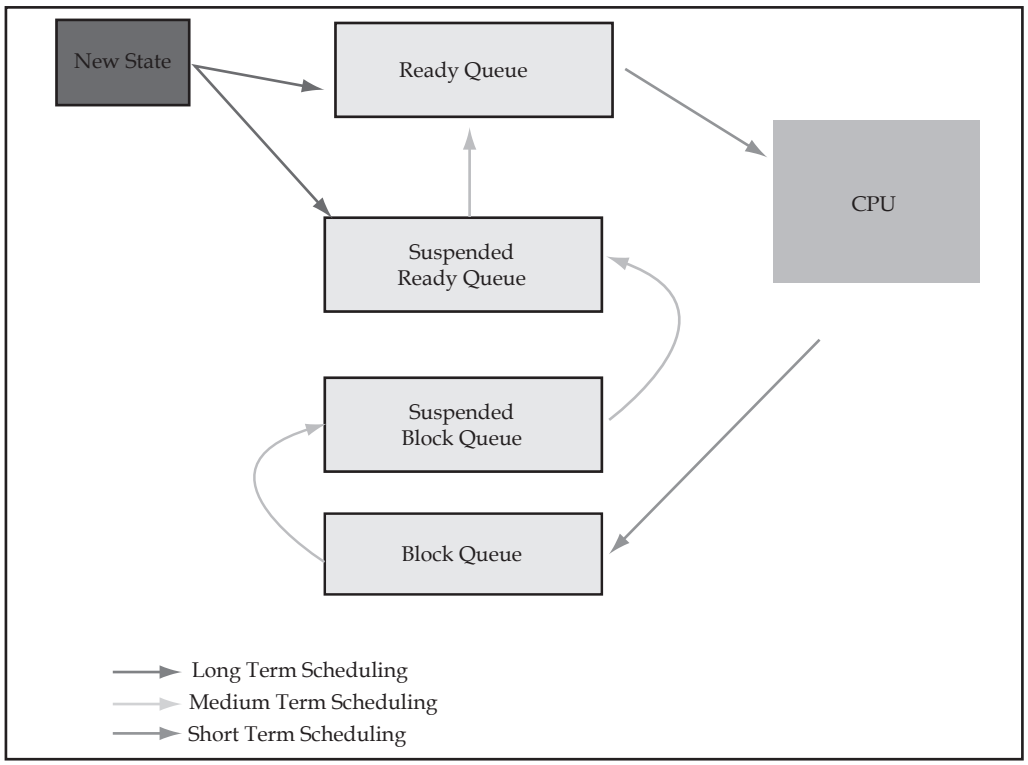
1. ***Response time:*** The interval of time from the moment a service is requested until the response begins to be received. In time-shared, interactive systems this is a better measure of responsiveness from a user's point of view than turnaround time, since processes may begin to produce output early in their execution.

Notes

2. ***Turnaround time:*** The interval between the submission of a process and the completion of its execution, including the actual running time, plus the time spent sleeping before being dispatched or while waiting to access various resources. This is the appropriate responsiveness measure for batch production, as well as for time-shared systems that maintain multiple batch queues, sharing CPU time among them.
3. ***Meeting deadlines:*** The ability of the OS to meet pre-defined deadlines for job completion. It makes sense only when the minimal execution time of an application can be accurately predicted.
4. ***Predictability:*** The ability of the system to ensure that a given task is executed within a certain time interval, and/or to ensure that a certain constant response time is granted within a strict tolerance, no matter what the machine load is.

When the overall system performance is considered, additional scheduling criteria must be taken into account:

1. ***Throughput:*** The rate of completion of processes (processes completed per unit time). This is a ''raw'' measure of how much work is performed, since it depends on the execution length of processes, but it's obviously affected by the scheduling policy.
2. ***User processor utilisation:*** Time (percentage of unit time) during which the CPU is running user processes. This is a measure of how well the system can serve the payload and keep at minimum time spent in housekeeping chores.
3. ***Overall processor utilisation:*** Time percentage during which the CPU is busy. It's a significant criterion for expensive hardware, that must be kept busy as much as possible in order to be justify its cost (e.g. supercomputers for numerical calculus applications).
4. ***Resource utilisation balance:*** It extends the idea of processor utilisation to take into account all system resources. A good scheduler should try to keep all the hardware resources in use at any time.



Notes

The design of the short term scheduler is one of the critical areas in the overall system design, because of the immediate effects on system performance from the user's point of view. It's usually one of the trickiest as well: since most processor architectures support their own task switching facilities, the implementation of the process switch mechanism is generally machine-dependent. The result is that the actual process switch software is usually written in the assembly language of a particular machine, whether the operating system is meant to be portable across different machines or not.

5.6 Multiple Processor Scheduling

The development of appropriate scheduling schemes for multiprocessor systems is problematic. Not only are uni-processor algorithms not directly applicable but some of the apparently correct methods are counter intuitive.

The scheduling problem for multiprocessor systems can be generally stated as "How can you execute a set of tasks T on a set of processors P subject to some set of optimizing criteria C?"

The most common goal of scheduling is to minimize the expected runtime of a task set. Examples of other scheduling criteria include minimizing the cost, minimizing communication delay, giving priority to certain users' processes, or needs for specialized hardware devices.

The scheduling policy for a multiprocessor system usually embodies a mixture of several of these criteria. Issues in Multiprocessor Scheduling Solutions to the scheduling problem come in two general forms: algorithms and scheduling systems.

Algorithms concentrate on policy while scheduling systems provide mechanism to implement the algorithms. Some scheduling systems run outside the operating system kernel, while others are part of a tightly-integrated distributed or parallel operating system.

Distributed systems communicate via message-passing, while parallel systems use shared memory. A task is the unit of computation in computing systems, and a job consists of one or more cooperating tasks. Global scheduling involves assigning a task to a particular processor within the system.

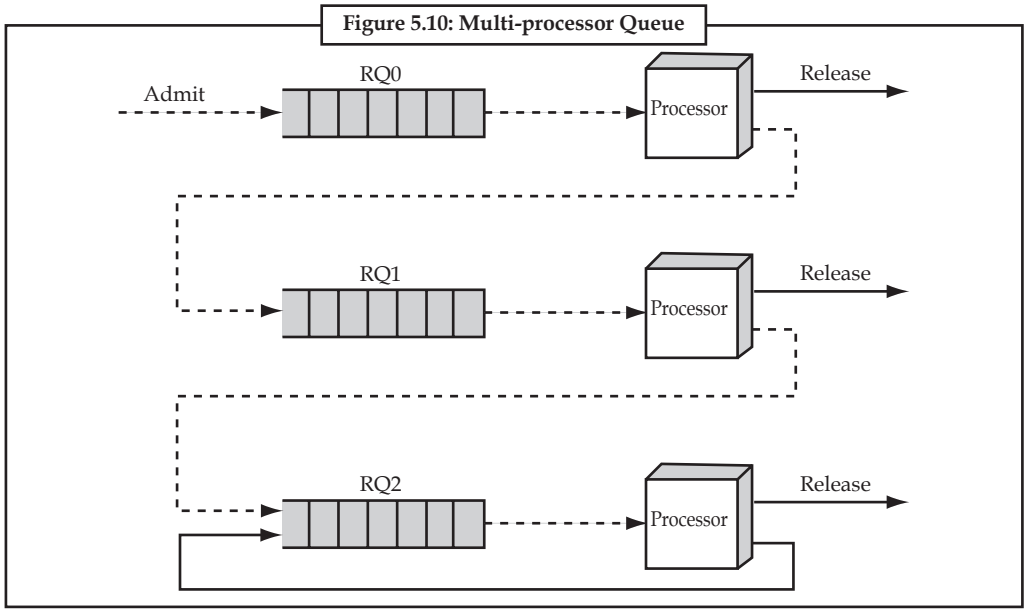
Local scheduling determines which of the set of available tasks at a processor runs next on that processor. Task migration can change the global mapping by moving a task to a new processor. If you have several jobs, each composed of many tasks, you can either assign several processors to a single job, or you can assign several tasks to a single processor. The former is known as space sharing, and the latter is called time sharing.

Global scheduling is often used to perform load sharing. Load sharing allows busy processors to off-load some of their work to less busy processors. Load balancing is a special case of load sharing, in which the goal is to keep the load even across all processors. Sender-initiated load sharing occurs when busy processors try to find idle processors to off-load some work. Receiver-initiated load sharing occurs when idle processors seek busy processors. It is now accepted wisdom that full load balancing is generally not worth doing, as the small gain in execution time over simpler load sharing is more than offset by the effort expended in maintaining the balanced load.

As the system runs, new tasks arrive while old tasks complete execution (or are served). If the arrival rate is greater than the service rate then the system is said to be unstable. If tasks are serviced as least as fast as they arrive, the system is said to be stable. If the arrival rate is just slightly less than the service rate for a system, an unstable scheduling policy can push the system into instability. A stable policy will never make a stable system unstable.

Notes

Figure 5.10: Multi-processor Queue



5.7 Thread Scheduling

The main approaches of threading scheduling are:

1. Load sharing
2. Gang scheduling
3. Dedicated processor assignment
4. Dynamic scheduling

5.7.1 Load Sharing

Processes are not assigned to a particular processor. A global queue of ready threads is maintained and each processor, when idle select a thread from the queue.

There are three versions of load sharing are these are:

1. First come first served
2. Smallest number of threads first
3. Preemptive smallest number of threads first

1. ***First come first served:*** when a job arrives each of its threads is placed consecutively at the end of the shared queue. When a processor becomes idle it picks the next ready thread, which it executes until completion or blocking.
2. ***Smallest number of thread first:*** The shared ready queue is organized as a priority queue with highest priority given to threads from jobs with the smallest number of unscheduled threads. Jobs of equal priority are ordered according to which job arrives first.
3. ***Preemptive smallest number of threads first:*** Highest is given to jobs with the smallest number of incomplete threads.

Notes

Advantages

Advantages of load sharing are:

1. The load is distributed evenly across the processors assuring that no processor is idle while work is available to do.
2. No centralized scheduler is required
3. The global queue can be organized and accessed by using any of the schemes.

Disadvantages

Disadvantages of load sharing are:

1. The central queue copies a region of memory that must be accessed in a manner that enforces mutual exclusion.
2. Preempted threads are unlikely to resume execution on the same processor.
3. If all threads are treated as a common pool of threads it is unlikely that all the threads of a program will gain access to processors at the same time.

5.7.2 Gang Scheduling

1. If closely related processes executes in parallel, synchronization blocking may be reduced.
2. Set of related threads of scheduled to run on a set of processors.
3. Gang scheduling has three parts.
   (a) Groups of related threads are scheduled as a unit, a gang
   (b) All members of a gang run simultaneously on different timeshared CPUs.
   (c) All gang members start and end their time slices together.
4. The trick that makes gang scheduling work is that all CPU are scheduled synchronously. This means that time is divided into discrete quanta.
5. An example of how gang scheduling works is given in the Table 5.1. Here you have a multiprocessor with six CPU being used by five processes, A through E, with a total of 24 ready threads.

Table 5.1: Gang Scheduling

	CPU						
		0	1	2	3	4	5
Time slot	0	A_0	A_1	A_2	A_3	A_4	A_5
	1	B_0	B_1	B_2	C_0	C_1	C_2
	2	D_0	D_1	D_2	D_3	D_4	E_0
	3	E_1	E_2	E_3	E_4	E_5	E_6
	4	A_0	A_1	A_2	A_3	A_4	A_5
	5	B_0	B_1	B_2	C_0	C_1	C_2
	6	D_0	D_1	D_2	D_3	D_4	E_0
	7	E_1	E_2	E_3	E_4	E_5	E_6

Notes

(a) During time slot 0, threads A_0 through A_5 are scheduled and run

(b) During time slot 1, threads B_0, B_1, B_2, C_0, C_1, C_2 are scheduled and run

(c) During time slot 2, D's five threads and E_0 get to run

(d) The remaining six threads belonging to process E run in the time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.

(e) Gang scheduling is useful for applications where performance severely degrades when any part of the application is not running.

5.7.3 Dedicated Processor Assignment

1. When application is scheduled its threads are assigned to a processor.
2. Some processor may be idle and no multiprogramming of processors.
3. Provides implicit scheduling defined by assignment of threads to processors. For the duration of program execution, each program is allocated a set of processors equal in number to the number of threads in the program. Processors are chosen from the available pool.

5.7.4 Dynamic Scheduling

1. Number of threads in a process are altered dynamically by the application.
2. Operating system and the application are involved in making scheduling decisions. The OS is responsible for partitioning the processors among the jobs.
3. Operating system adjusts load to improve the use:
   (a) Assign idle processors.
   (b) New arrivals may be assigned to a processor that is used by a job currently using more than one processor.
   (c) Hold request until processor is available
   (d) New arrivals will be given a processor before existing running applications.



Task List various versions of load sharing.

5.8 Summary

- The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination. Processes can be terminated in one of two ways: Normal Termination and Abnormal Termination.
- When more than one process are executing concurrently in the operating system, then they are allowed to cooperate (both mutually and destructively) with each other.
- Those processes are known as cooperating process. Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes.

Notes

- When two or more concurrent processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.
- Critical Section is a part of the program where the shared memory is accessed. Mutual Exclusion is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphoiinitislize'.
- Message passing is a form of inter process communication used in concurrent computing, where the communication is made by the sending of messages to recipients.

5.9 Keywords

CPU scheduling: It is the basic of multiprogramming where the task of selecting a waiting process from the ready queue and allocating the CPU to it.

CPU utilization: It is an important criterion in real-time system and multi-programmed systems where the CPU must be as busy as possible in performing different tasks.

Response Time: The amount of time between a request is Submitted and the first response is produced is called response time.

Throughput: The number of processes executed in a specified time period is called throughput.

Turnaround Time: The amount of time that is needed to execute a process is called turnaround time. It is the actual job time plus the waiting time.

Waiting Time: The amount of time the process has waited is called waiting time. It is the turnaround time minus actual job time.

5.10 Self Assessment

Fill in the blanks:

1. A ..... header will contain pointers to the first and last PCBs in the list.
2. ..... scheduling is the basics of multiprogramming.
3. A major task of an operating system is to manage a collection of ..... .
4. The CPU is ..... to the selected process by the dispatcher.
5. ..... is a method of CPU scheduling that is a preemptive version of shortest job next scheduling.
6. A ..... scheduling algorithm will simply put the new process at the head of the ready queue.
7. ..... scheduling is essentially concerned with memory management.
8. The most common goal of scheduling is to ..... of a task set.
9. ..... scheduling involves assigning a task to a particular processor within the system.
10. ..... scheduling is really the easiest way of scheduling.

5.11 Review Questions

Notes

1. Suppose that a scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?
2. Assume you have the following jobs to execute with one processor, with the jobs arriving in the order listed here:

i	T(pi)
0	80
1	20
2	10
3	20
4	50

   (a) Suppose a system uses FCFS scheduling. Create a Gantt chart illustrating the execution of these processes?

   (b) What is the turnaround time for process p_3?

   (c) What is the average wait time for the processes?

3. Suppose a new process in a system arrives at an average of six processes per minute and each such process requires an average of 8 seconds of service time. Estimate the fraction of time the CPU is busy in a system with a single processor.
4. A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many possible different schedules are there? Give a formula in terms of n.
5. Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

   These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum).What (if any) relation holds between the following pairs of sets of algorithms?

   (a) Priority and SJF

   (b) Multilevel Feedback Queues and FCFS

   (c) Priority and FCFS

   (d) RR and SJF

6. Distinguish between long term and short term scheduling.
7. Consider the following set of processes, with the length of the CPU burst given in milliseconds.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Notes

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 all at time 0.

(a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1).

(b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

(c) What is the waiting time of each process for each of the scheduling algorithms in part a?

8. Consider the following set of processes, with the length of the CPU burst and arrival time given in milliseconds.

Process	Burst Time	Priority
P_1	8	0
P_2	4	0.4
P_3	1	1

(a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, (the algorithm can look into the future and wait for a shorter process that will arrive).

(b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

(c) What is the waiting time of each process for each of the scheduling algorithms in part a?

9. Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:

(a) First Come First Served

(b) Round Robin

(c) Multilevel Feedback Queues

10. Write short notes on:

(a) Waiting time

(b) Response time

(c) Throughput

Answers: Self Assessment

1. ready-queue
2. CPU
3. processes
4. allocated
5. Shortest remaining time
6. non preemptive priority
7. Medium term
8. minimize the expected runtime
9. Global
10. Round-robin

5.12 Further Readings

Notes



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 6: Process Synchronization

CONTENTS

Objectives

Introduction

6.1 Synchronization Process

6.2 Critical Selection Problem

6.2.1 Mutual Exclusion Conditions

6.2.2 Proposals for Achieving Mutual Exclusion

6.3 Semaphores

6.4 Monitors

6.5 Deadlock

6.6 Deadlock Characterization

6.7 Handling of Deadlocks

6.7.1 Deadlock Prevention

6.7.2 Deadlock Avoidance

6.7.3 Deadlock Detection and Recovery

6.7.4 Ignore Deadlock

6.7.5 The Banker's Algorithm for Detecting/Preventing Deadlocks

6.8 Summary

6.9 Keywords

6.10 Self Assessment

6.11 Review Questions

6.12 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe synchronization process
- Know critical selection problem
- Define semaphores
- Explain deadlock
- Describe handling of deadlocks

Introduction

Modern operating systems, such as Unix, execute processes concurrently. Although there is a single Central Processor (CPU), which execute the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing a single process a few hundred microseconds of CPU time before replacing it with another process.)

Notes

Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system. The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU, the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPUs registers, and other data), and removes the process from the use of the CPU. The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this.

In this unit, we shall discuss about the deadlock. A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. It is often seen in a paradox like 'the chicken or the egg'.

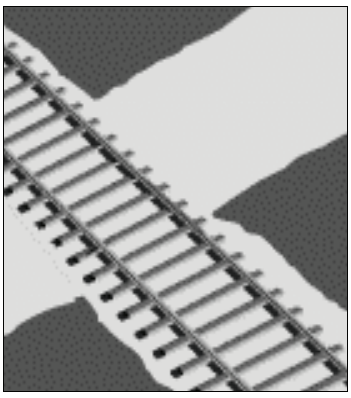
This situation may be likened to two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can't be satisfied, so a deadlock occurs.

6.1 Synchronization Process

Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action. Synchronization involves the orderly sharing of system resources by processes.

To illustrate the process synchronization, consider the above railway intersection diagram. You can think of this intersection as a system resource that is shared by two processes: the car process and the train process. If only one process is active, then no resource conflict exists. But what happens when both processes are active and they both arrive at the intersection simultaneously? In this case, the shared resource becomes a problem. They cannot both use the resource at the same time or a collision will occur. Similarly, processes sharing resources on a computer must be properly managed in order to avoid "collisions."

Figure 6.1: Railway-road Intersection



Consider a machine with a single printer running a time-sharing operation system. If a process needs to print its results, it must request that the operating system give it access to the printer's

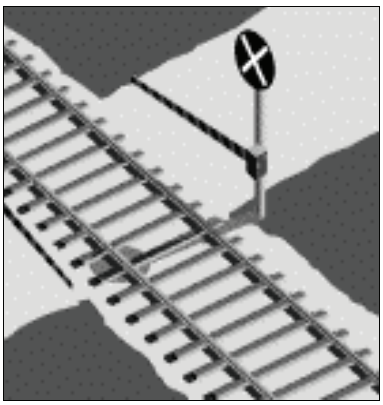
Notes

device driver. At this point, the operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes available. Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both.

Now that the problem of synchronization is properly stated, consider the following related definitions:

1. ***Critical Resource:*** A resource shared with constraints on its use (e.g., memory, files, printers, etc.)
2. ***Critical Section:*** Code that accesses a critical resource
3. ***Mutual Exclusion:*** At most one process may be executing a Critical Section with respect to a particular critical resource simultaneously

Figure 6.2: Railway-road Intersection with Signal



In the example given above, the printer is the critical resource. Let's suppose that the processes which are sharing this resource are called process A and process B. The critical sections of process A and process B are the sections of the code which issue the print command. In order to insure that both processes do not attempt to use the printer at the same, they must be granted mutually exclusive access to the printer driver. The idea of mutual exclusion with our railroad intersection by adding a semaphore to the picture.

Semaphores are used in software systems in much the same way as they are in railway systems. Corresponding to the section of track that can contain only one train at a time is a sequence of instructions that can be executed by only one process at a time. Such a sequence of instructions is called a critical section.

6.2 Critical Selection Problem

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:

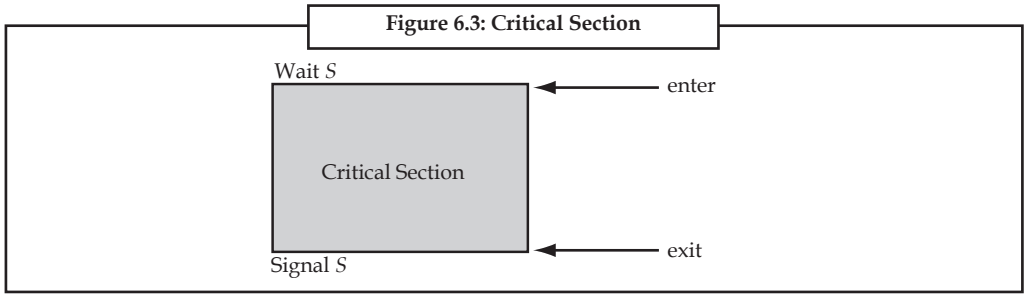
1. Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.

Notes

2. Codes that alter one or more variables that are possibly being referenced in "read-updata-write" fashion by another thread.
3. Codes use a data structure while any part of it is possibly being altered by another thread.
4. 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.

Figure 6.3: Critical Section



A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.



Note Mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

6.2.1 Mutual Exclusion Conditions

If you could arrange matters such that no two processes were ever in their critical sections simultaneously, you could avoid race conditions. You need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

1. No two processes may at the same moment inside their critical sections.
2. No assumptions are made about relative speeds of processes or number of CPUs.
3. No process outside its critical section should block other processes.
4. No process should wait arbitrary long to enter its critical section.

6.2.2 Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time.

Notes

Problem: When one process is updating shared modifiable data in its critical section, no other process should allowed to enter in its critical section.

Proposal 1: Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

Conclusion

Disabling interrupts is sometimes a useful interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for users process. The reason is that it is unwise to give user process the power to turn off interrupts.

Proposal 2: Lock Variable (Software Solution)

In this solution, you consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first test the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

Conclusion

The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

Proposal 3: Strict Alteration

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspect turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a loop continually testing 'turn' to see when it becomes 1.Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

Conclusion

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above mentioned solution do is this: when a processes wants to enter in its critical section , it checks to see if then entry is allowed. If it is not, the process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPU-time.

Now look at some interprocess communication primitives is the pair of steep-wakeup.

Notes

Sleep

It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

Wakeup

It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

Bounded Buffer Producers and Consumers

The bounded buffer producers and consumers assumes that there is a fixed buffer size i.e., a finite numbers of slots are available.

Statement

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem.

Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.

Trouble arises when:

1. The producer wants to put a new data in the buffer, but buffer is already full. *Solution*: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.

   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

Conclusion: This approaches also leads to same race conditions you have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.



Task Discuss "sleep" and "Wakeup" stages.

6.3 Semaphores

E.W. Dijkstra (1965) abstracted the key notion of mutual exclusion in his concepts of semaphores.

Definition

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'.

Notes

Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only nonnegative 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

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.

Producer-Consumer Problem using Semaphores

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

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.

Initialization

1. Set full buffer slots to 0.

   i.e., semaphore Full = 0.

2. Set empty buffer slots to N.

   i.e., semaphore empty = N.

3. 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 ( )
```

Notes

```
        V (mutex)
        V (full);
    Consumer ( )
    WHILE (true)
        P (full)
        P (mutex);
        remove-Item ( );
        V (mutex);
        V (empty);
        consume-Item (Item)
```

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

Semaphore is a machanism to resolve resources conflicts by tallying resource seekers what is the state of sought resources, achieving a mutual exclusive access to resources. Often semaphore operates as a type of mutual exclusive counters (such as mutexes) where it holds a number of access keys to the resources. Process that seeks the resources must obtain one of those access keys, one of semaphores, before it proceeds further to utilize the resource. If there is no more such a key available to the process, it has to wait for the current resource user to release the key.

A semaphore could have the value 0,indicating that no wakeups were saved, or some positive values if one or more wakeups were pending.

A semaphore s is an integer variable that apart from initialization, is accesssed only through two standard atomic operations, wait and signal. these operations were orignially termed p(for wait to test) and v(for signal to increment).

The classical defination of wait in psedocode is:

```
wait(s)
{
while(s<=0)
;// no-op
s--;
}
```

The classical defination of signal in psedocode is:

```
signal(s)
{
s++;
}
```

Modification to the integer value of smaphore in wait and signal operations must be executed individually.

That is, when one process modifies the semaphore value no other process can simultaneously modifiy that same semaphore value.

Notes

SR Program: The Dining Philosophers

This semaphore solution to the readers-writers problem can let writers starve because readers arriving after a now-waiting writer arrived earlier can still access the database to read if enough readers continually trickle in and ''keep the database in a read state'' then that waiting writer will never get to write

```
resource philosopher
   import dining_server
body philosopher(i : int; dcap : cap dining_server; thinking, eating: int)
   write("philosopher", i, "alive, max think eat delays", thinking, eating)
   procedure think()
      var napping : int
      napping := int(random(1000*thinking))
      writes("age=",age(),", philosopher ",i," thinking for ",napping," ms\n")
      nap(napping)
   end think
   procedure eat()
      var napping : int
      napping := int(random(1000*eating))
      writes("age=",age(),", philosopher ",i," eating for ",napping," ms\n")
      nap(napping)
   end eat
   process phil
      do true ->
         think()
         writes("age=", age(), ", philosopher ", i, " is hungry\n")
         dcap.take_forks(i)
         writes("age=", age(), ", philosopher ", i, " has taken forks\n")
         eat()
         dcap.put_forks(i)
         writes("age=", age(), ", philosopher ", i, " has returned forks\n")
      od
   end phil
end philosopher
resource dining_server
   op take_forks(i : int), put_forks(i : int)
body dining_server(num_phil : int)
   write("dining server for", num_phil, "philosophers is alive")
   sem mutex := 1
   type states = enum(thinking, hungry, eating)
   var state[1:num_phil] : states := ([num_phil] thinking)
   sem phil[1:num_phil] := ([num_phil] 0)
   procedure left(i : int) returns lft : int
      if i=1 -> lft := num_phil [] else -> lft := i-1 fi
```

Notes

```
   end left
   procedure right(i : int) returns rgh : int
      if i=num_phil -> rgh := 1 [] else -> rgh := i+1 fi
   end right
   procedure test(i : int)
      if state[i] = hungry and state[left(i)] ~= eating
            and state[right(i)] ~= eating ->
         state[i] := eating
         V(phil[i])
      fi
   end test
   proc take_forks(i)
      P(mutex)
      state[i] := hungry
      test(i)
      V(mutex)
      P(phil[i])
   end take_forks
   proc put_forks(i)
      P(mutex)
      state[i] := thinking
      test(left(i)); test(right(i))
      V(mutex)
   end put_forks
end dining_server
resource start()
   import philosopher, dining_server
   var num_phil : int := 5, run_time : int := 60
   getarg(1, num_phil); getarg(2, run_time)
   var max_think_delay[1:num_phil] : int := ([num_phil] 5)
   var max_eat_delay[1:num_phil] : int := ([num_phil] 2)
   fa i := 1 to num_phil ->
      getarg(2*i+1, max_think_delay[i]); getarg(2*i+2, max_eat_delay[i])
   af
   var dcap : cap dining_server
   write(num_phil, "dining philosophers running", run_time, "seconds")
   dcap := create dining_server(num_phil)
   fa i := 1 to num_phil ->
      create philosopher(i, dcap, max_think_delay[i], max_eat_delay[i])
   af
   nap(1000*run_time); write("must stop now"); stop
end start
/* ............... Example compile and run(s)
```

Notes

```
% sr -o dphi dphi.sr
% ./dphi 5 10
5 dining philosophers running 10 seconds
dining server for 5 philosophers is alive
philosopher 1 alive, max think eat delays 5 2
age=37, philosopher 1 thinking for 491 ms
philosopher 2 alive, max think eat delays 5 2
age=50, philosopher 2 thinking for 2957 ms
philosopher 3 alive, max think eat delays 5 2
age=62, philosopher 3 thinking for 1374 ms
philosopher 4 alive, max think eat delays 5 2
age=74, philosopher 4 thinking for 1414 ms
philosopher 5 alive, max think eat delays 5 2
age=87, philosopher 5 thinking for 1000 ms
age=537, philosopher 1 is hungry
age=541, philosopher 1 has taken forks
age=544, philosopher 1 eating for 1351 ms
age=1097, philosopher 5 is hungry
age=1447, philosopher 3 is hungry
age=1451, philosopher 3 has taken forks
age=1454, philosopher 3 eating for 1226 ms
age=1497, philosopher 4 is hungry
age=1898, philosopher 1 has returned forks
age=1901, philosopher 1 thinking for 2042 ms
age=1902, philosopher 5 has taken forks
age=1903, philosopher 5 eating for 1080 ms
age=2687, philosopher 3 has returned forks
age=2691, philosopher 3 thinking for 2730 ms
age=2988, philosopher 5 has returned forks
age=2991, philosopher 5 thinking for 3300 ms
age=2992, philosopher 4 has taken forks
age=2993, philosopher 4 eating for 1818 ms
age=3017, philosopher 2 is hungry
age=3020, philosopher 2 has taken forks
age=3021, philosopher 2 eating for 1393 ms
age=3947, philosopher 1 is hungry
age=4418, philosopher 2 has returned forks
age=4421, philosopher 2 thinking for 649 ms
age=4423, philosopher 1 has taken forks
age=4424, philosopher 1 eating for 1996 ms
age=4817, philosopher 4 has returned forks
age=4821, philosopher 4 thinking for 742 ms
age=5077, philosopher 2 is hungry
```

Notes

```
age=5427, philosopher 3 is hungry
age=5431, philosopher 3 has taken forks
age=5432, philosopher 3 eating for 857 ms
age=5569, philosopher 4 is hungry
age=6298, philosopher 3 has returned forks
age=6301, philosopher 3 thinking for 1309 ms
age=6302, philosopher 5 is hungry
age=6304, philosopher 4 has taken forks
age=6305, philosopher 4 eating for 498 ms
age=6428, philosopher 1 has returned forks
age=6430, philosopher 1 thinking for 1517 ms
age=6432, philosopher 2 has taken forks
age=6433, philosopher 2 eating for 133 ms
age=6567, philosopher 2 has returned forks
age=6570, philosopher 2 thinking for 3243 ms
age=6808, philosopher 4 has returned forks
age=6810, philosopher 4 thinking for 2696 ms
age=6812, philosopher 5 has taken forks
age=6813, philosopher 5 eating for 1838 ms
age=7617, philosopher 3 is hungry
age=7621, philosopher 3 has taken forks
age=7622, philosopher 3 eating for 1251 ms
age=7957, philosopher 1 is hungry
age=8658, philosopher 5 has returned forks
age=8661, philosopher 5 thinking for 4755 ms
age=8662, philosopher 1 has taken forks
age=8664, philosopher 1 eating for 1426 ms
age=8877, philosopher 3 has returned forks
age=8880, philosopher 3 thinking for 2922 ms
age=9507, philosopher 4 is hungry
age=9511, philosopher 4 has taken forks
age=9512, philosopher 4 eating for 391 ms
age=9817, philosopher 2 is hungry
age=9908, philosopher 4 has returned forks
age=9911, philosopher 4 thinking for 3718 ms
age=10098, philosopher 1 has returned forks
age=10100, philosopher 1 thinking for 2541 ms
must stop now
age=10109, philosopher 2 has taken forks
age=10110, philosopher 2 eating for 206 ms
% ./dphi 5 10 1 10 10 1 1 10 10 1 10 1
5 dining philosophers running 10 seconds
dining server for 5 philosophers is alive
```

Notes

```
philosopher 1 alive, max think eat delays 1 10
age=34, philosopher 1 thinking for 762 ms
philosopher 2 alive, max think eat delays 10 1
age=49, philosopher 2 thinking for 5965 ms
philosopher 3 alive, max think eat delays 1 10
age=61, philosopher 3 thinking for 657 ms
philosopher 4 alive, max think eat delays 10 1
age=74, philosopher 4 thinking for 8930 ms
philosopher 5 alive, max think eat delays 10 1
age=86, philosopher 5 thinking for 5378 ms
age=726, philosopher 3 is hungry
age=731, philosopher 3 has taken forks
age=732, philosopher 3 eating for 3511 ms
age=804, philosopher 1 is hungry
age=808, philosopher 1 has taken forks
age=809, philosopher 1 eating for 3441 ms
age=4246, philosopher 3 has returned forks
age=4250, philosopher 3 thinking for 488 ms
age=4252, philosopher 1 has returned forks
age=4253, philosopher 1 thinking for 237 ms
age=4495, philosopher 1 is hungry
age=4498, philosopher 1 has taken forks
age=4499, philosopher 1 eating for 8682 ms
age=4745, philosopher 3 is hungry
age=4748, philosopher 3 has taken forks
age=4749, philosopher 3 eating for 2095 ms
age=5475, philosopher 5 is hungry
age=6025, philosopher 2 is hungry
age=6855, philosopher 3 has returned forks
age=6859, philosopher 3 thinking for 551 ms
age=7415, philosopher 3 is hungry
age=7420, philosopher 3 has taken forks
age=7421, philosopher 3 eating for 1765 ms
age=9015, philosopher 4 is hungry
age=9196, philosopher 3 has returned forks
age=9212, philosopher 3 thinking for 237 ms
age=9217, philosopher 4 has taken forks
age=9218, philosopher 4 eating for 775 ms
age=9455, philosopher 3 is hungry
age=9997, philosopher 4 has returned forks
age=10000, philosopher 4 thinking for 2451 ms
age=10002, philosopher 3 has taken forks
age=10004, philosopher 3 eating for 9456 ms
must stop now
                                                    */
```

Notes

6.4 Monitors

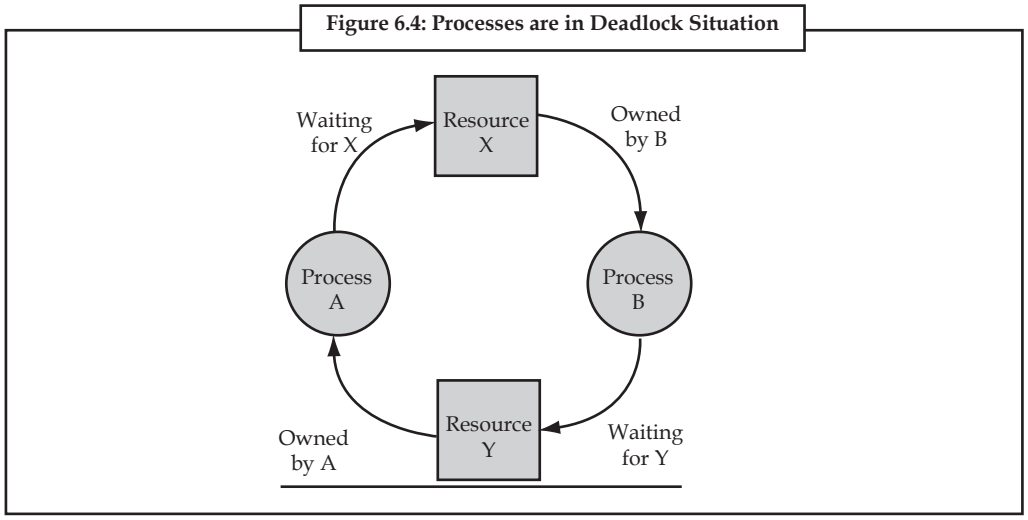
A monitor is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization. It allows only one process to be active within the monitor at a time. One simple implementation is shown below.

```
monitor monitor_name
{
    // shared variable declarations
    procedure P1 (…) { …. }
    …
    procedure Pn(…) {……}
    Initialization code ( ….) { …}
    …
}
```

6.5 Deadlock

Deadlock occurs when you have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress. Those processes are deadlocked because all the processes are waiting. None of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, I assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set. In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by another deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

Figure 6.4: Processes are in Deadlock Situation



Notes

6.6 Deadlock Characterization

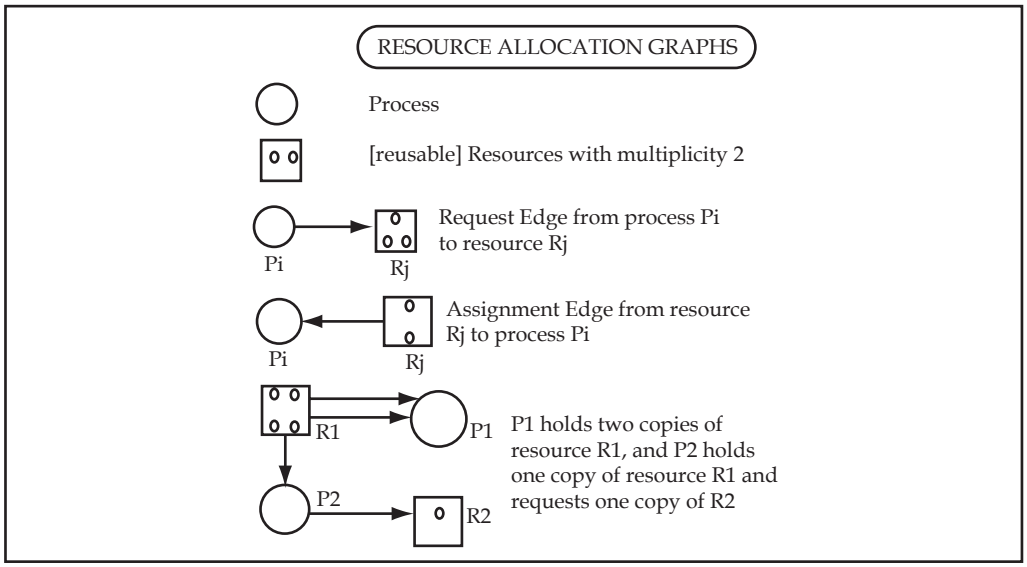
Necessary Conditions

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

1. Resources are used in mutual exclusion.
2. Resources are acquired piecemeal (i.e. not all the resources that are needed to complete an activity are obtained at the same time in a single indivisible action).
3. Resources are not preempted (i.e. a process does not take away resources being held by another process).
4. Resources are not spontaneously given up by a process until it has satisfied all its outstanding requests for resources (i.e. a process, being that it cannot obtain some needed resource it does not kindly give up the resources that it is currently holding).

Resource Allocation Graphs

Resource Allocation Graphs (RAGs) are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system.



State transitions can be represented as transitions between the corresponding resource allocation graphs. Here are the rules for state transitions:

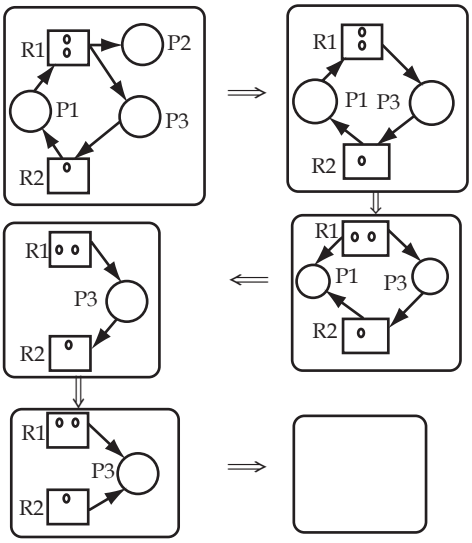
1. ***Request:*** If process Pi has no outstanding request, it can request simultaneously any number (up to multiplicity) of resources R1, R2, ..Rm. The request is represented by adding appropriate requests edges to the RAG of the current state.
2. ***Acquisition:*** If process Pi has outstanding requests and they can all be simultaneously satisfied, then the request edges of these requests are replaced by assignment edges in the RAG of the current state
3. ***Release:*** If process Pi has no outstanding request then it can release any of the resources it is holding, and remove the corresponding assignment edges from the RAG of the current state.

Here are some important propositions about deadlocks and resource allocation graphs:

Notes

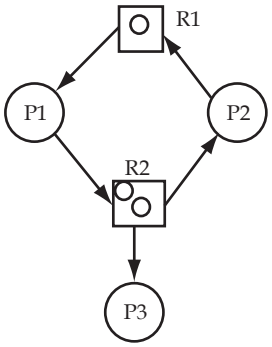
1. If a RAG of a state of a system is fully reducible (i.e. it can be reduced to a graph without any edges using ACQUISITION and RELEASE operations) then that state is not a deadlock state.
2. If a state is not a deadlock state then its RAG is fully reducible [this holds only if you are dealing with reusable resources; it is false if you have consumable resources]
3. A cycle in the RAG of a state is a necessary condition for that being a deadlock state
4. A cycle in the RAG of a state is a sufficient condition for that being a deadlock state only in the case of reusable resources with multiplicity one.

Example: Here is an example of reduction of a RAG:



Reduction of a RAG

And here is a deadlock-free system with a loop.



RAG with Loop but no Deadlock

Task A monitor is a software synchronization tool or hardware synchronization tool.

Notes

6.7 Handling of Deadlocks

There are several ways to address the problem of deadlock in an operating system.

1. Prevent
2. Avoid
3. Detection and recovery
4. Ignore

6.7.1 Deadlock Prevention

Deadlocks can be prevented by ensuring that at least one of the following four conditions occur:

1. ***Mutual exclusion:*** Removing the mutual exclusion condition means that no process may have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
2. ***Hold and wait:*** The "hold and wait" conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations); this advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens, are known as the all-or-none algorithms.)
3. ***No preemption:*** A "no preemption" (lockout) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm.



Note Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead.

Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

4. ***Circular wait:*** The circular wait condition: Algorithms that avoid circular waits include "disable interrupts during critical sections" , and "use a hierarchy to determine a partial ordering of resources" (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra's solution.

6.7.2 Deadlock Avoidance

Deadlock Avoidance, assuming that you are in a safe state (i.e. a state from which there is a sequence of allocations and releases of resources that allows all processes to terminate) and you are requested certain resources, simulates the allocation of those resources and determines if the resultant state is safe. If it is safe the request is satisfied, otherwise it is delayed until it becomes safe.

The Banker's Algorithm is used to determine if a request can be satisfied. It uses requires knowledge of who are the competing transactions and what are their resource needs. Deadlock avoidance is essentially not used in distributed systems.

Notes

6.7.3 Deadlock Detection and Recovery

Often neither deadlock avoidance nor deadlock prevention may be used. Instead deadlock detection and recovery are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS.

Detecting the possibility of a deadlock before it occurs is much more difficult and is, in fact, generally undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in specific environments, using specific means of locking resources, deadlock detection may be decidable. In the general case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

6.7.4 Ignore Deadlock

In the Ostrich Algorithm it is hoped that deadlock doesn't happen. In general, this is a reasonable strategy. Deadlock is unlikely to occur very often; a system can run for years without deadlock occurring. If the operating system has a deadlock prevention or detection system in place, this will have a negative impact on performance (slow the system down) because whenever a process or thread requests a resource, the system will have to check whether granting this request could cause a potential deadlock situation.

If deadlock does occur, it may be necessary to bring the system down, or at least manually kill a number of processes, but even that is not an extreme solution in most situations.

6.7.5 The Banker's Algorithm for Detecting/Preventing Deadlocks

Banker's Algorithm for Single Resource

This is modeled on the way a small town banker might deal with customers' lines of credit. In the course of conducting business, our banker would naturally observe that customers rarely draw their credit lines to their limits. This, of course, suggests the idea of extending more credit than the amount the banker actually has in her coffers.

Suppose we start with the following situation

Customer	Credit Used	Credit Line
Andy	0	6
Barb	0	5
Marv	0	4
Sue	0	7
Funds Available	10	
Max Commitment		22

Our banker has 10 credits to lend, but a possible liability of 22. Her job is to keep enough in reserve so that ultimately each customer can be satisfied over time: That is, that each customer will be able to access his full credit line, just not all at the same time. Suppose, after a while, the bank's credit line book shows.

Notes

Customer	Credit Used	Credit Line
Andy	1	6
Barb	1	5
Marv	2	4
Sue	4	7
Funds Available	2	
Max Commitment		22

Eight credits have been allocated to the various customers; two remain. The question then is: Does a way exist such that each customer can be satisfied? Can each be allowed their maximum credit line in some sequence? We presume that, once a customer has been allocated up to his limit, the banker can delay the others until that customer repays his loan, at which point the credits become available to the remaining customers. If we arrive at a state where no customer can get his maximum because not enough credits remain, then a deadlock could occur, because the first customer to ask to draw his credit to its maximum would be denied, and all would have to wait.

To determine whether such a sequence exists, the banker finds the customer closest to his limit: If the remaining credits will get him to that limit, The banker then assumes that that loan is repaid, and proceeds to the customer next closest to his limit, and so on. If all can be granted a full credit, the condition is safe.

In this case, Marv is closest to his limit: assume his loan is repaid. This frees up 4 credits. After Marv, Barb is closest to her limit (actually, she's tied with Sue, but it makes no difference) and 3 of the 4 freed from Marv could be used to award her maximum. Assume her loan is repaid; we have now freed 6 credits. Sue is next, and her situation is identical to Barb's, so assume her loan is repaid. We have freed enough credits (6) to grant Andy his limit; thus this state safe.

Suppose, however, that the banker proceeded to award Barb one more credit after the credit book arrived at the state immediately above:

Customer	Credit Used	Credit Line
Andy	1	6
Barb	2	5
Marv	2	4
Sue	4	7
Funds Available	1	
Max Commitment		22

Now it's easy to see that the remaining credit could do no good toward getting anyone to their maximum.

So, to recap, the banker's algorithm looks at each request as it occurs, and tests if granting it will lead to a safe state. If not, the request is delayed. To test for a safe state, the banker checks to see if enough resources will remain after granting the request to satisfy the customer closest to his maximum. If so, that loan is assumed repaid, and the next customer checked, and so on. If all loans can be repaid, then the request leads to a safe state, and can be granted. In this case, we see that if Barb is awarded another credit, Marv, who is closest to his maximum, cannot be awarded enough credits, hence Barb's request can't be granted —it will lead to an unsafe state3.

Banker's Algorithm for Multiple Resources

Suppose, for example, we have the following situation, where the first table represents resources assigned, and the second resources still required by five processes, A, B, C, D, and E.

Notes

Resources Assigned

Processes	Tapes	Plotters	Printers	Toasters
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0
Total Existing	6	3	4	2
Total Claimed by Processes	5	3	2	2
Remaining Unclaimed	1	0	2	0

Resources Still Needed

Processes	Tapes	Plotters	Printers	Toasters
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

The vectors E, P and A represent Existing, Possessed and Available resources respectively:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2)
A = (1, 0, 2, 0)
```

Notice that

```
A = E - P
```

Now, to state the algorithm more formally, but in essentially the same way as the example with Andy, Barb, Marv and Sue:

1. Look for a row whose unmet needs don't exceed what's available, that is, a row where P <= A; if no such row exists, we are deadlocked because no process can acquire the resources it needs to run to completion. If there's more than one such row, just pick one.
2. Assume that the process chosen in 1 acquires all the resources it needs and runs to completion, thereby releasing its resources. Mark that process as virtually terminated and add its resources to A.
3. Repeat 1 and 2 until all processes are either virtually terminated (safe state), or a deadlock is detected (unsafe state).

Going thru this algorithm with the foregoing data, we see that process D's requirements are smaller than A, so we virtually terminate D and add its resources back into the available pool:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2) - (1, 1, 0, 1) = (4, 2, 2, 1)
A = (1, 0, 2, 0) + (1, 1, 0, 1) = (2, 1, 2, 1)
```

Now, A's requirements are less than A, so do the same thing with A:

```
P = (4, 2, 2, 1) - (3, 0, 1, 1) = (1, 2, 1, 0)
A = (2, 1, 2, 1) + (3, 0, 1, 1) = (5, 1, 3, 2)
```

At this point, we see that there are no remaining processes that can't be satisfied from available resources, so the illustrated state is safe.

Notes

6.8 Summary

- Race condition is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.
- It may arise in multi-process environment, especially when communicating between separate processes or threads of execution.
- Mutual exclusion means that only one of the processes is allowed to execute its critical section at a time. Mutex, semaphores and motors are some of the process synchronization tools. Mutex is a software tool used in concurrency control. It is short form of mutual exclusion.
- A mutex is a program element that allows multiple program processes to share the same resource but not simultaneously. Semaphore is a software concurrency control tool. It bears analogy to old Roman system of message transmission using flags. It enforces synchronization among communicating processes and does not require busy waiting.
- In counting semaphore the integer value can range over an unrestricted domain. In binary semaphore the integer value can range only between 0 and 1.
- A monitor is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization. It allows only one process to be active within the monitor at a time.
- Bounded Buffer Problem, readers and writers problem, sleeping barber problem, and dining philosopher problem are some of the classical synchronization problems taken from real life situations.
- A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. Resource Allocation Graphs (RAGs) are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system. There are several ways to address the problem of deadlock in an operating system – Prevent, Avoid, Detection and recovery and Ignore.

6.9 Keywords

Deadlock: A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.

Monitor: It is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization.

Mute: It is a program element that allows multiple program processes to share the same resource but not simultaneously.

Mutex: It is a software tool used in concurrency control. It is short form of mutual exclusion.

Mutual exclusion: It means that only one of the processes is allowed to execute its critical section at a time.

Race condition: It is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.

Resource Allocation Graphs (RAGs): Those are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system.

Semaphore: It is a software concurrency control tool.

Notes

6.10 Self Assessment

Fill in the blanks:

1. ..... involves the orderly sharing of system resources by processes.
2. ..... are used in software systems in much the same way as they are in railway systems.
3. Part of the program where the shared memory is accessed is called the ..... .
4. A ..... is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization.
5. Resource Allocation Graphs (RAGs) are ..... labeled graphs.
6. Algorithms that avoid mutual exclusion are called ..... synchronization algorithms.
7. ..... abstracted the key notion of mutual exclusion in his concepts of semaphores.
8. "No preemption" condition also known as ..... .
9. ..... processes share a common, fixed-size (bounded) buffer.
10. Binary Semaphores can assume only the value 0 or the value ..... .

6.11 Review Questions

1. What is a safe state? What is its use in deadlock avoidance?
2. Describe briefly any one method of deadlock prevention.
3. What is concurrency? Explain with example deadlock and starvation.
4. Explain the different deadlock strategies.
5. Can a process be allowed to request multiple resources simultaneously in a system where deadlock are avoided? Discuss why or why not.
6. How deadlock situation are avoided and prevented so that no systems are locked by deadlock?
7. Consider the following resource allocation situation:

   Process P = {P1, P2, P3, P4, P5}

   Resources R = {R1, R2, R3}

   Allocation E = {P1®R1, P1®R2, P2®R2, P3®R2, P4®R3, P5®R2, R2®P4, R3®P1}

   Resource instances n(R1)=3, n(R2)=4, n(R3)=1

   Draw the precedence graph. Determine whether there is a deadlock in the above situation.
8. Explain process synchronization process.
9. What do you mean by mutual exclusion conditions? Explain
10. Write short note on semaphore.

Notes

Answers: Self Assessment

1.	Synchronization	2.	Semaphores	3.	Critical Section	4.	monitor
5.	directed	6.	non-blocking	7.	E.W. Dijkstra (1965)		
8.	lockout	9.	Two	10.	1		

6.12 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 7: Memory Management

Notes

CONTENTS

Objectives

Introduction

7.1 Memory Management

7.2 Logical and Physical Address Space

7.3 Swapping

7.4 Contiguous Memory Allocation

7.5 Paging

7.6 Segmentation

7.7 Segmentation with Paging

7.8 Virtual Memory

7.9 Demand Paging

7.10 Page Replacement

7.10.1 Static Page Replacement Algorithms

7.10.2 Dynamic Page Replacement Algorithms

7.11 Page Allocation Algorithm

7.12 Thrashing

7.13 Summary

7.14 Keywords

7.15 Self Assessment

7.16 Review Questions

7.17 Further Readings

Objectives

After studying this unit, you will be able to:

- Define memory management
- Describe swapping
- Explain segmentation with paging
- Know virtual memory
- Describe demand paging

Introduction

Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly. When the computer is in normal operation, its memory usually contains the main parts of the operating system and some or all of the application programs and related data

Notes

that are being used. Memory is often used as a shorter synonym for random access memory (RAM). This kind of memory is located on one or more microchips that are physically close to the microprocessor in the computer. Most desktop and notebook computers sold today include at least 16 megabytes of RAM, and are upgradeable to include more. The more RAM you have, the less frequently the computer has to access instructions and data from the more slowly accessed hard disk form of storage.

Memory is sometimes distinguished from storage, or the physical medium that holds the much larger amounts of data that won't fit into RAM and may not be immediately needed there. Storage devices include hard disks, floppy disks, CD-ROM, and tape backup systems. The terms auxiliary storage, auxiliary memory, and secondary memory have also been used for this kind of data repository.

Additional kinds of integrated and quickly accessible memory are read-only memory (ROM), programmable ROM (PROM), erasable programmable ROM (EPROM). These are used to keep special programs and data, such as the basic input/output system, that need to be in the computer all the time.

The memory is a resource that needs to be managed carefully. Most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, some number of megabytes of medium-speed, medium-price, volatile main memory (RAM), and hundreds of thousands of megabytes of slow, cheap, non-volatile disk storage. It is the job of the operating system to coordinate how these memories are used.

7.1 Memory Management

In addition to the responsibility of managing processes, the operating system must efficiently manage the primary memory of the computer. The part of the operating system which handles this responsibility is called the memory manager. Since every process must have some amount of primary memory in order to execute, the performance of the memory manager is crucial to the performance of the entire system. The memory manager is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory. Managing the sharing of primary memory and minimizing memory access time are the basic goals of the memory manager.

When an operating system manages the computer's memory, there are two broad tasks to be accomplished:

1. Each process must have enough memory in which to execute, and it can neither run into the memory space of another process nor be run into by another process.
2. The different types of memory in the system must be used properly so that each process can run most effectively.

The first task requires the operating system to set up memory boundaries for types of software and for individual applications.



Example: Let's look at an imaginary small system with 1 megabyte (1,000 kilobytes) of RAM. During the boot process, the operating system of our imaginary computer is designed to go to the top of available memory and then "back up" far enough to meet the needs of the operating system itself. Let's say that the operating system needs 300 kilobytes to run. Now, the operating system goes to the bottom of the pool of RAM and starts building up with the various driver software required to control the hardware subsystems of the computer. In our imaginary computer, the drivers take up 200 kilobytes. So after getting the operating system completely loaded, there are 500 kilobytes remaining for application processes.

Notes

When applications begin to be loaded into memory, they are loaded in block sizes determined by the operating system. If the block size is 2 kilobytes, then every process that is loaded will be given a chunk of memory that is a multiple of 2 kilobytes in size. Applications will be loaded in these fixed block sizes, with the blocks starting and ending on boundaries established by words of 4 or 8 bytes. These blocks and boundaries help to ensure that applications won't be loaded on top of one another's space by a poorly calculated bit or two. With that ensured, the larger question is what to do when the 500-kilobyte application space is filled.

In most computers, it's possible to add memory beyond the original capacity. For example, you might expand RAM from 1 to 2 megabytes. This works fine, but tends to be relatively expensive. It also ignores a fundamental fact of computing - most of the information that an application stores in memory is not being used at any given moment. A processor can only access memory one location at a time, so the vast majority of RAM is unused at any moment. Since disk space is cheap compared to RAM, then moving information in RAM to hard disk can greatly expand RAM space at no cost. This technique is called virtual memory management.

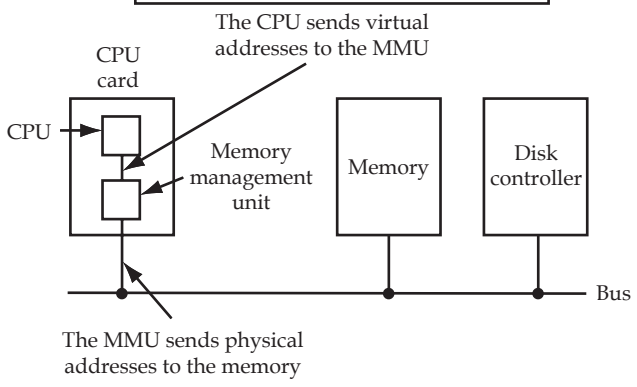
Disk storage is only one of the memory types that must be managed by the operating system, and is the slowest. Ranked in order of speed, the types of memory in a computer system are:

1. ***High-speed cache:*** This is fast, relatively small amounts of memory that are available to the CPU through the fastest connections. Cache controllers predict which pieces of data the CPU will need next and pull it from main memory into high-speed cache to speed up system performance.
2. ***Main memory:*** This is the RAM that you see measured in megabytes when you buy a computer.
3. ***Secondary memory:*** This is most often some sort of rotating magnetic storage that keeps applications and data available to be used, and serves as virtual RAM under the control of the operating system.

The operating system must balance the needs of the various processes with the availability of the different types of memory, moving data in blocks (called pages) between available memory as the schedule of processes dictates.

Systems for managing memory can be divided into two categories: the system of moving processes back and forth between main memory and disk during execution (known as swapping and paging) and the process that does not do so (that is, no swapping and ping).

Figure 7.1: Memory Management



Task What is the concept of primary and secondary memory?

Notes

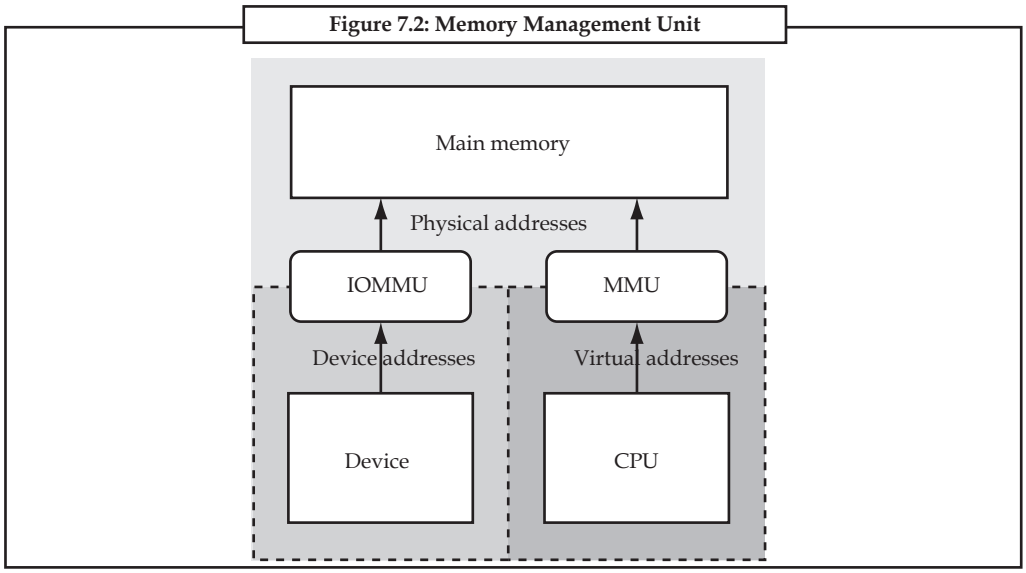
7.2 Logical and Physical Address Space

A memory address identifies a physical location in computer memory, somewhat similar to a street address in a town. The address points to the location where data is stored, just like your address points to where you live.

In the analogy of a person's address, the address space would be an area of locations, such as a neighborhood, town, city, or country. Two addresses may be numerically the same but refer to different locations, if they belong to different address spaces. This is similar to your address being, say, "32, Main Street", while another person may reside in "32, Main Street" in a different town from yours.

Many programmers prefer to use a flat memory model, in which there is no distinction between code space, data space, and virtual memory – in other words, numerically identical pointers refer to exactly the same byte of RAM in all three address spaces.

Figure 7.2: Memory Management Unit



Physical Address

A physical address, also real address, or binary address, is the memory address, that is electronically (in the form of binary number) presented on the computer address bus circuitry in order to enable the data bus to access a particular storage cell of main memory.

Logical Address

Logical address is the address at which a memory location appears to reside from the perspective of an executing application program. This may be different from the physical address due to the operation of a Memory Management Unit (MMU) between the CPU and the memory bus.

Physical memory may be mapped to different logical addresses for various purposes.

Example: The same physical memory may appear at two logical addresses and if accessed by the program at one address, data will pass through the processor cache whereas if it is accessed at the other address, it will bypass the cache.

In a system supporting virtual memory, there may actually not be any physical memory mapped to a logical address until an access is attempted. The access triggers special functions of the

Notes

operating system which reprogram the MMU to map the address to some physical memory, perhaps writing the old contents of that memory to disk and reading back from disk what the memory should contain at the new logical address. In this case, the logical address may be referred to as a virtual address.

Logical vs. 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, you usually refer to the logical address as a virtual address.

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.

7.3 Swapping

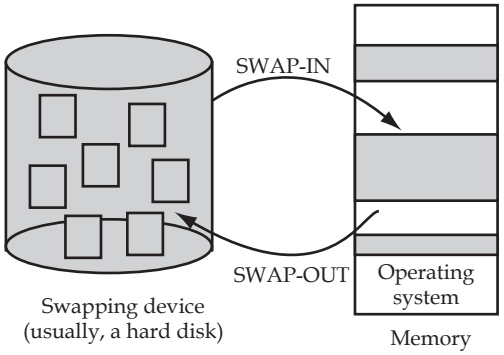
Any operating system has a fixed amount of physical memory available. Usually, application need more than the physical memory installed on your system, for that purpose the operating system uses a swap mechanism: instead of storing data in physical memory, it uses a disk file.

Swapping is the act of moving processes between memory and a backing store. This is done to free up available memory. Swapping is necessary when there are more processes than available memory. At the coarsest level, swapping is done a process at a time.

To move a program from fast-access memory to a slow-access memory is known as "swap out", and the reverse operation is known as "swap in". The term often refers specifically to the use of a hard disk (or a swap file) as virtual memory or "swap space".

When a program is to be executed, possibly as determined by a scheduler, it is swapped into core for processing; when it can no longer continue executing for some reason, or the scheduler decides its time slice has expired, it is swapped out again.

Figure 7.3: Memory Swapping



Task Differentiate PROM and EPROM type memory.

Notes

7.4 Contiguous Memory Allocation

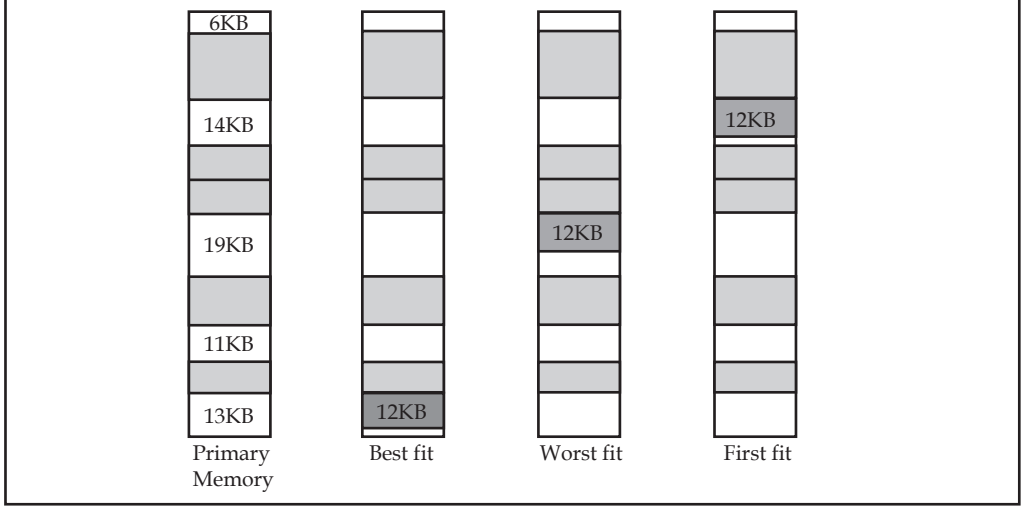
The real challenge of efficiently managing memory is seen in the case of a system which has multiple processes running at the same time. Since primary memory can be space-multiplexed, the memory manager can allocate a portion of primary memory to each process for its own use. However, the memory manager must keep track of which processes are running in which memory locations, and it must also determine how to allocate and deallocate available memory when new processes are created and when old processes complete execution. While various different strategies are used to allocate space to processes competing for memory, three of the most popular are Best fit, Worst fit, and First fit. Each of these strategies is described below:

1. ***Best fit:*** The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

2. ***Worst fit:*** The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

3. ***First fit:*** There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.

Figure 7.4: Best Fit, Worst Fit and First Fit Memory Allocation Method



Notice in the above figure that the Best fit and First fit strategies both leave a tiny segment of memory unallocated just beyond the new process. Since the amount of memory is small, it is not likely that any new processes can be loaded here. This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as fragmentation. The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes. Thus, a larger amount of space will be left as seen in the Figure 7.4.

Buddy System

Notes

Memory management, especially memory allocation to processes, is a fundamental issue in operating systems. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system.

In a buddy system, the entire memory space available for allocation is initially treated as a single block whose size is a power of 2. When the first request is made, if its size is greater than half of the initial block then the entire block is allocated. Otherwise, the block is split in two equal companion buddies. If the size of the request is greater than half of one of the buddies, then allocate one to it. Otherwise, one of the buddies is split in half again. This method continues until the smallest block greater than or equal to the size of the request is found and allocated to it.

In this method, when a process terminates the buddy block that was allocated to it is freed. Whenever possible, an unnallocated buddy is merged with a companion buddy in order to form a larger free block. Two blocks are said to be companion buddies if they resulted from the split of the same direct parent block.

The following Figure 7.5 illustrates the buddy system at work, considering a 1024k (1-megabyte) initial block and the process requests as shown at the left of the table.

Figure 7.5: Diagram of Buddy System

	0	128k	256k		512k	1024k
start	1024k					
A=70K	A	128		256		512
B=35K	A	B	64	256		512
C=80K	A	B	64	C	128	512
A ends	128	B	64	C	128	512
D=60K	128	B	D	C	128	512
B ends	128	64	D	C	128	512
D ends	256			C	128	512
C ends	512					512
end	1024k					

Notice that, whenever there is a request that corresponds to a block of sizes, your program should select the block of that size that was most recently declared free. Furthermore, when a block is split in two, the left-one (lower addresses) should be selected before the right-one.

You can assume that the list of requests is such that all requests can always be served. In other words, you can make the following assumptions: no process will request more than the available memory; processes are uniquely identified while active; and no request for process termination is issued before its corresponding request for memory allocation.

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache related and memory access latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible.

Notes

7.5 Paging

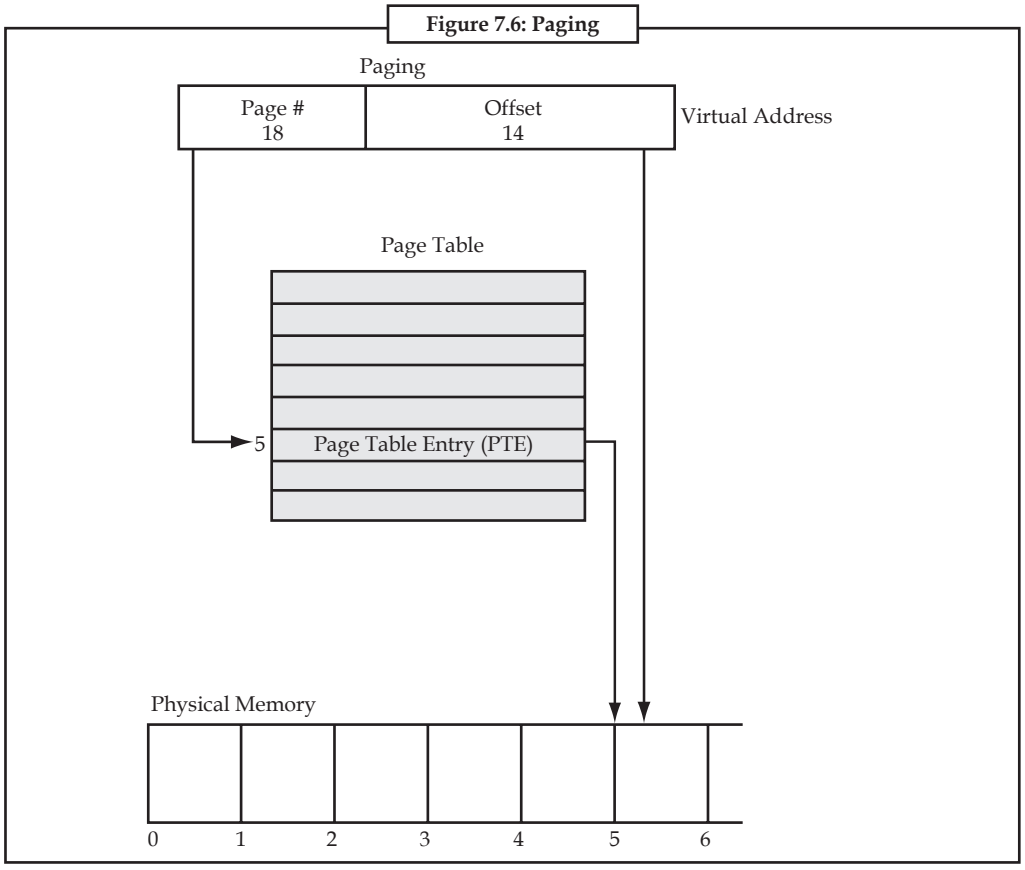
It is a technique for increasing the memory space available by moving infrequently-used parts of a program's working memory from RAM to a secondary storage medium, usually hard disk. The unit of transfer is called a page.

A memory management unit (MMU) monitors accesses to memory and splits each address into a page number (the most significant bits) and an offset within that page (the lower bits). It then looks up the page number in its page table. The page may be marked as paged in or paged out. If it is paged in then the memory access can proceed after translating the virtual address to a physical address. If the requested page is paged out then space must be made for it by paging out some other page, i.e. copying it to disk. The requested page is then located on the area of the disk allocated for "swap space" and is read back into RAM. The page table is updated to indicate that the page is paged in and its physical address recorded.

The MMU also records whether a page has been modified since it was last paged in. If it has not been modified then there is no need to copy it back to disk and the space can be reused immediately.

Paging allows the total memory requirements of all running tasks (possibly just one) to exceed the amount of physical memory, whereas swapping simply allows multiple processes to run concurrently, so long as each process on its own fits within physical memory.

Figure 7.6: Paging



On operating systems, such as Windows NT, Windows 2000 or UNIX, the memory is logically divided in pages. When the system needs a certain portion of memory which is currently in the swap (this is called a page fault) it will load all the corresponding pages into RAM. When a page is not accessed for a long time, it is saved back to disk and discarded.

Notes

In a virtual memory system, it is common to map between virtual addresses and physical addresses by means of a data structure called a page table. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the CPU, i.e., RAM.

The page number of an address is usually found from the most significant bits of the address; the remaining bits yield the offset of the location within the page. The page table is normally indexed by page number and contains information on whether the page is currently in main memory, and where it is in main memory or on disk.

Conventional page tables are sized to the virtual address space and store the entire virtual address space description of each process. Because of the need to keep the virtual-to-physical translation time low, a conventional page table is structured as a fixed, multi-level hierarchy, and can be very inefficient at representing a sparse virtual address space, unless the allocated pages are carefully aligned to the page table hierarchy.

7.6 Segmentation

It is very common for the size of program modules to change dynamically. For instance, the programmer may have no knowledge of the size of a growing data structure. If a single address space is used, as in the paging form of virtual memory, once the memory is allocated for modules they cannot vary in size. This restriction results in either wastage or shortage of memory. To avoid the above problem, some computer systems are provided with many independent address spaces. Each of these address spaces is called a segment. The address of each segment begins with 0 and segments may be compiled separately. In addition, segments may be protected individually or shared between processes. However, segmentation is not transparent to the programmer like paging. The programmer is involved in establishing and maintaining the segments.

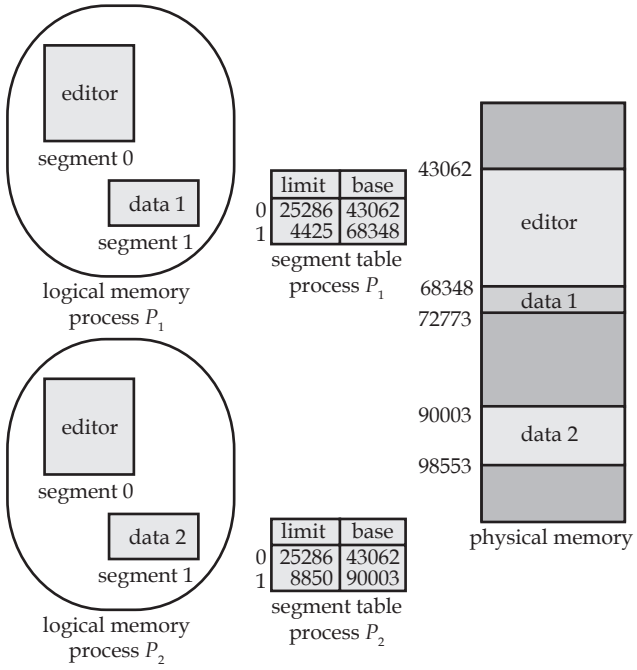
Segmentation is one of the most common ways to achieve memory protection like paging. An instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a set of permissions, and a length, associated with it. If the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is delivered.

In addition to the set of permissions and length, a segment also has associated with it information indicating where the segment is located in memory. It may also have a flag indicating whether the segment is present in main memory or not; if the segment is not present in main memory, an exception is delivered, and the operating system will read the segment into memory from secondary storage. The information indicating where the segment is located in memory might be the address of the first location in the segment, or might be the address of a page table for the segment. In the first case, if a reference to a location within a segment is made, the offset within the segment will be added to address of the first location in the segment to give the address in memory of the referred-to item; in the second case, the offset of the segment is translated to a memory address using the page table.

A memory management unit (MMU) is responsible for translating a segment and offset within that segment into a memory address, and for performing checks to make sure the translation can be done and that the reference to that segment and offset is permitted.

Notes

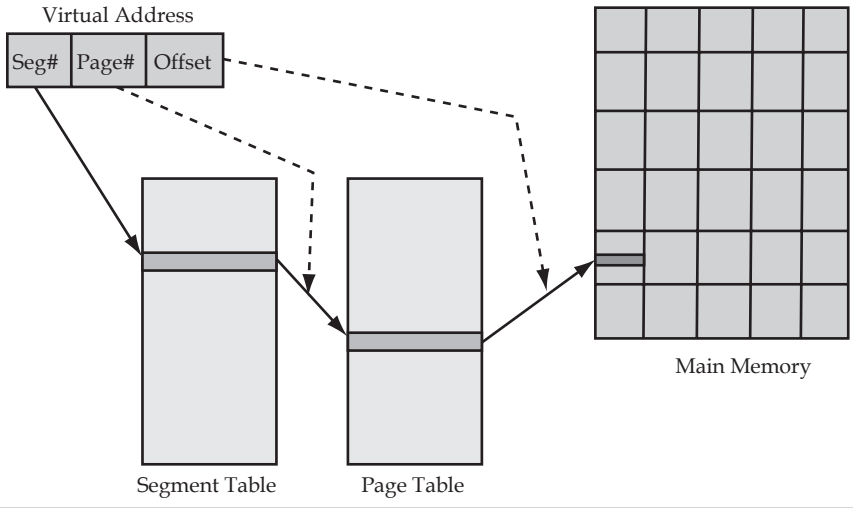
Figure 7.7: Sharing of Segments in a Segmented Memory System



Task Differentiate between paging and segmentation technique.

7.7 Segmentation with Paging

Figure 7.8: Diagram of Segmentation with Paging

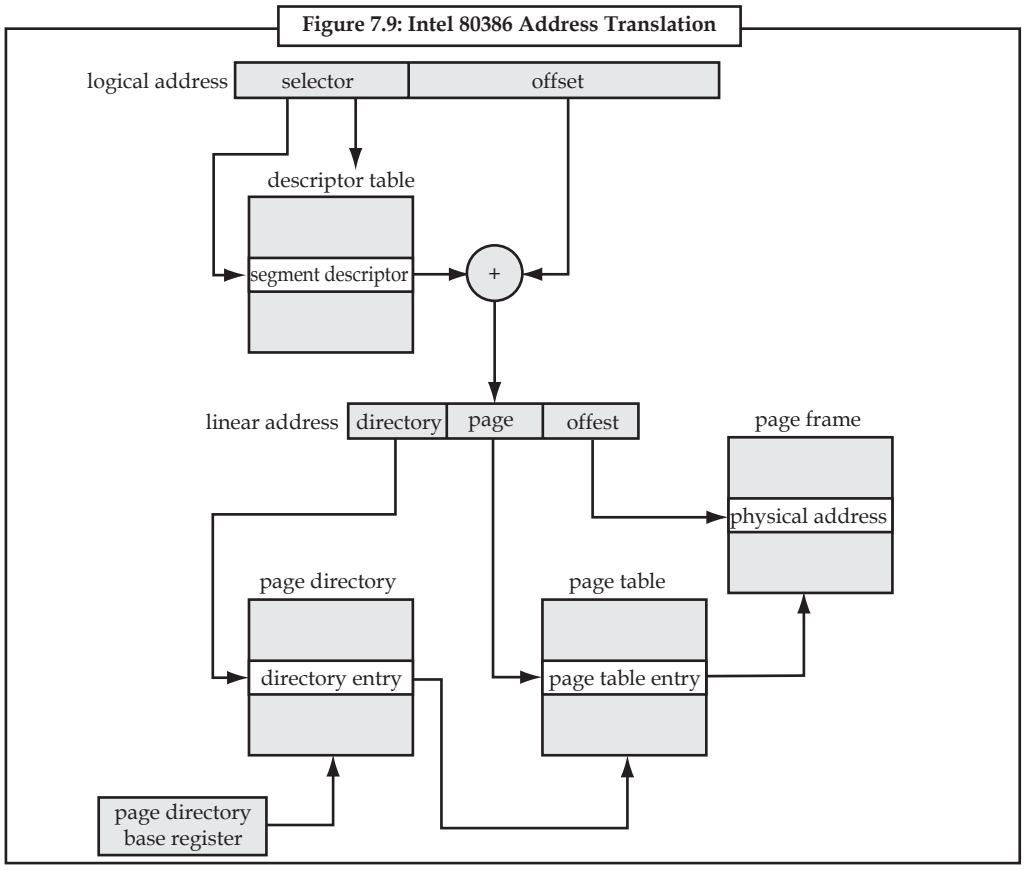


Segments can be of different lengths, so it is harder to find a place for a segment in memory than a page. With segmented virtual memory, you get the benefits of virtual memory but you still have to do dynamic storage allocation of physical memory. In order to avoid this, it is possible to combine segmentation and paging into a two-level virtual memory system. Each segment

descriptor points to page table for that segment. This give some of the advantages of paging (easy placement) with some of the advantages of segments (logical division of the program).

Notes

Figure 7.9: Intel 80386 Address Translation



Some operating systems allow for the combination of segmentation with paging. If the size of a segment exceeds the size of main memory, the segment may be divided into equal size pages.

The virtual address consists of three parts: (1) segment number (2) the page within the segment and (3) the offset within the page. The segment number is used to find the segment descriptor and the address within the segment is used to find the page frame and the offset within that page.

7.8 Virtual Memory

Many of us use computers on a daily basis. Although you use it for many different purposes in many different ways, you share one common reason of using them; to make our job more efficient and easier.

However, there are times when computers cannot run as fast as you want it to or just cannot handle certain processes effectively, due to the shortage of system resources. When the limitations of system resources become a major barrier to achieving your maximum productivity, you often consider the apparent ways of upgrading the system, such as switching to a faster CPU, adding more physical memory (RAM), installing utility programs, and so on. As a computer user, you want to make the most of the resources available; the process of preparing plans to coordinate the total system to operate in the most efficient manner. This is called a system optimization.

When it comes to system optimization, there is one great invention of modern computing called virtual memory. It is an imaginary memory area supported by some operating system (for

Notes

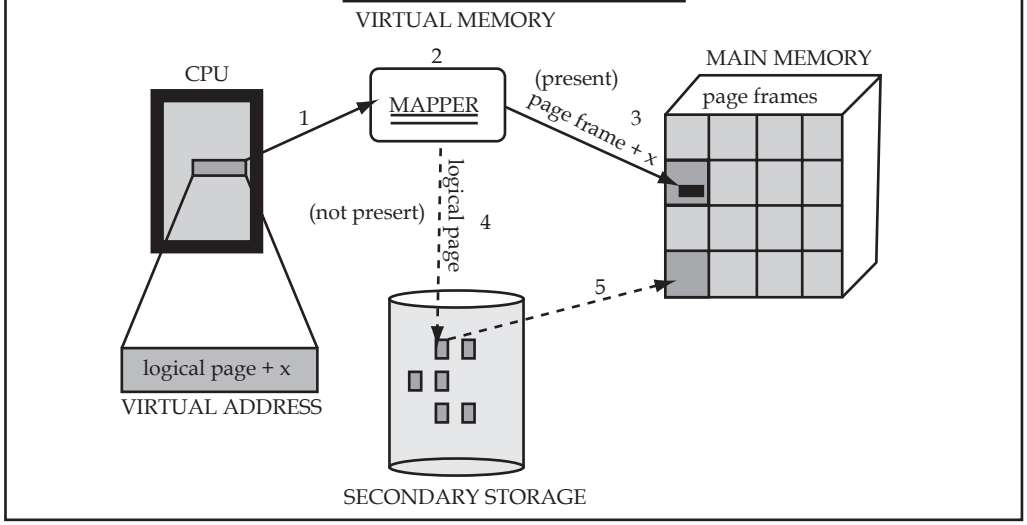
example, Windows but not DOS) in conjunction with the hardware. You can think of virtual memory as an alternate set of memory addresses. Programs use these virtual addresses rather than real addresses to store instructions and data. When the program is actually executed, the virtual addresses are converted into real memory addresses.

The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilize.

Example: Virtual memory might contain twice as many addresses as main memory.

A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into main memory those portions of the program needed at any given point during execution.

Figure 7.10: Virtual Memory



To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

The process of translating virtual addresses into real addresses is called mapping. The copying of virtual pages from disk to main memory is known as paging or swapping.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimization it provides, is that it is faster to read the information from physical memory, than use the relevant I/O channel to get that information. This is called caching. It is implemented inside the OS.

Task Why operating system need virtual memory? Discuss.

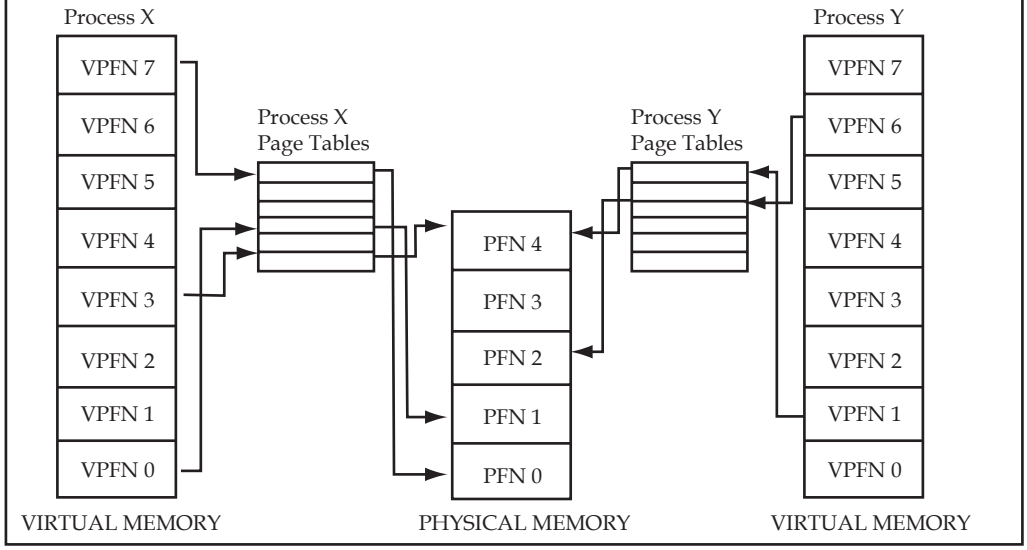
7.9 Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to

Notes

only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

Figure 7.11: Abstract Model of Virtual to Physical Address Mapping



When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in Figure 7.11 there is no entry in Process X's page table for virtual PFN 2 and so if Process X attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a page fault has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory. This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

7.10 Page Replacement

When the number of available real memory frames on the free list becomes low, a page stealer is invoked. A page stealer moves through the Page Frame Table (PFT), looking for pages to steal.

The PFT includes flags to signal which pages have been referenced and which have been modified. If the page stealer encounters a page that has been referenced, it does not steal that page, but instead, resets the reference flag for that page. The next time the clock hand (page stealer) passes that page and the reference bit is still off, that page is stolen. A page that was not referenced in the first pass is immediately stolen.

Notes

The modify flag indicates that the data on that page has been changed since it was brought into memory. When a page is to be stolen, if the modify flag is set, a pageout call is made before stealing the page. Pages that are part of working segments are written to paging space; persistent segments are written to disk.

All paging algorithms function on three basic policies: a fetch policy, a replacement policy, and a placement policy. In the case of static paging, describes the process with a shortcut: the page that has been removed is always replaced by the incoming page; this means that the placement policy is always fixed. Since you are also assuming demand paging, the fetch policy is also a constant; the page fetched is that which has been requested by a page fault. This leaves only the examination of replacement methods.

7.10.1 Static Page Replacement Algorithms

Optimal Replacement Theory

In a best case scenario the only pages replaced are those that will either never be needed again, or have the longest number of page requests before they are referenced. This "perfect" scenario is usually used only as a benchmark by which other algorithms can be judged, and is referred to as either Belady's Optimal Algorithm or Perfect Prediction (PP). Such a feat cannot be ccomplished without full prior knowledge of the reference stream, or a record of past behavior that is incredibly consistent. Although usually a pipe dream for system designers, suggests it can be seen in very rare cases, such as large weather prediction programs that carry out the same operations on consistently sized data.

Random Replacement

On the flip-side of complete optimization is the most basic approach to page replacement: simply choosing the victim, or page to be removed, at random. Each page frame involved has an equal chance of being chosen, without taking into consideration the reference stream or locality principals. Due to its random nature, the behavior of this algorithm is quite obviously, random and unreliable. With most reference streams this method produces an unacceptable number of page faults, as well as victim pages being thrashed unnecessarily. A better performance can almost always be achieved by employing a different algorithm. Most systems stopped experimenting with this method as early as the 1960's.

First-In, First-Out (FIFO)

First-in, first-out is as easy to implement as Random Replacement, and although its performance is equally unreliable or worse, its behavior does follow a predictable pattern. Rather than choosing a victim page at random, the oldest page (or first-in) is the first to be removed. Conceptually compares FIFO to a limited size queue, with items being added to the queue at the tail. When the queue fills (all of the physical memory has been allocated), the first page to enter is pushed out of head of the queue. Similar to Random Replacement, FIFO blatantly ignores trends, and although it produces less page faults, still does not take advantage of locality trends unless by coincidence as pages move along the queue. A modification to FIFO that makes its operation much more useful is First-In Not-Used First-Out (FINUFO). The only modification here is that a single bit is used to identify whether or not a page has been referenced during its time in the FIFO queue. This utility, or referenced bit, is then used to determine if a page is identified as a victim. If, since it has been fetched, the page has been referenced at least once, its bit becomes set. When a page must be swapped out, the first to enter the queue whose bit has not been set is removed; if every active page has been referenced, a likely occurrence taking locality into consideration, all of the bits are reset. In a worst-case scenario this could cause minor and temporary thrashing, but is generally very effective given its low cost.

Least Recently Used (LRU)

Notes

You have seen that an algorithm must use some kind of behavior prediction if it is to be efficient. One of the most basic page replacement approaches uses the usage of a page as an indication of its "worth" when searching for a victim page: the Least Recently Used (LRU) Algorithm. LRU was designed to take advantage of "normal" program operation, which generally consists of a series of loops with calls to rarely executed code. In terms of the virtual addressing and pages, this means that the majority of code executed will be held in a small number of pages; essentially the algorithm takes advantage of the locality principal. As per the previous description of locality, LRU assumes that a page recently referenced will most likely be referenced again soon. To measure the "time" elapsed since a page has been a part of the reference stream, a backward distance is stored. This distance must always be greater than zero, the point for the current position in the reference stream, and can be defined as infinite in the case of a page that has never been referenced. Thus, the victim page is defined as the one with the maximum backward distance; if two or more points meet this condition, a page is chosen arbitrarily. Actual implementation of the backward distance number can vary, and does play an important role in the speed and efficiency of this algorithm. This can be done by sorting page references in order of their age into a stack, allowing quick identification of victims. However the overhead associated with sorting does not generally justify the speed of identification, unless specific hardware exists to perform this operation. Many operating systems do not assume this hardware exists (such as UNIX), and instead increment an age counter for every active page during the page stream progression, as described by. When a page is referenced once again, or is brought in due to a page fault, its value is simply set to zero. Since storage for the backward age is limited, a maximum value may also be defined; generally any page that has reached this age becomes a valid target for replacement. As with any algorithm, modifications can be made to increase performance when additional hardware resources are available.

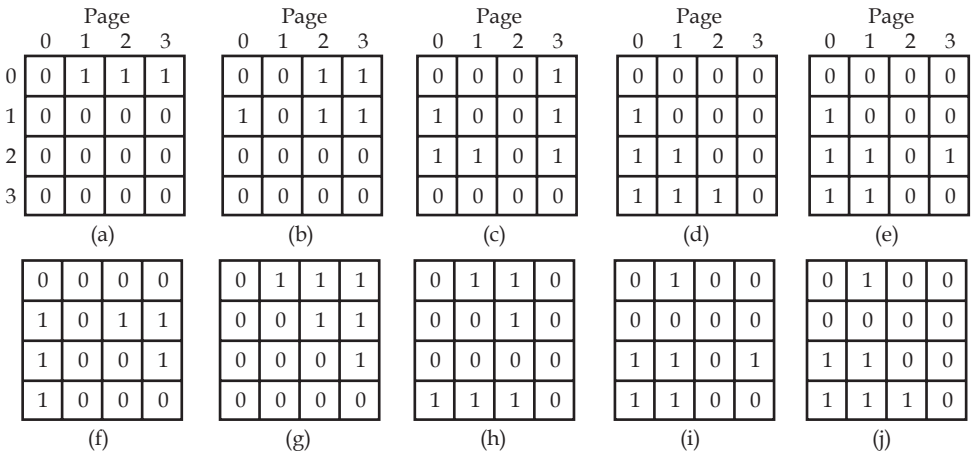


Example: A machine with n page frames, the LRU hardware can maintain a matrix of n × n bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Figure 7.12 for four page frames and page references in the order.

0 1 2 3 2 1 0 3 2 3

After page 0 is referenced, we have the situation of Figure 7.12(a). After page 1 is reference, we have the situation of Figure 7.12(b), and so forth.

Figure 7.12: LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.



Notes

Least Frequently Used (LFU)

Often confused with LRU, Least Frequently Used (LFU) selects a page for replacement if it has not been used often in the past. Instead of using a single age as in the case of LRU, LFU defines a frequency of use associated with each page. This frequency is calculated throughout the reference stream, and its value can be calculated in a variety of ways. The most common frequency implementation begins at the beginning of the page reference stream, and continues to calculate the frequency over an ever-increasing interval. Although this is the most accurate representation of the actual frequency of use, it does have some serious drawbacks. Primarily, reactions to locality changes will be extremely slow. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less than the pages associated with the previous program. Since the context has changed, and the pages swapped out will most likely be needed again soon (due to the new program's principal of locality), a period of thrashing will likely occur. If the beginning of the reference stream is used, initialization code of a program can also have a profound influence. The pages associated with initial code can influence the page replacement policy long after the main body of the program has begun execution. One way to remedy this is to use a popular variant of LFU, which uses frequency counts of a page since it was last loaded rather than since the beginning of the page reference stream. Each time a page is loaded, its frequency counter is reset rather than being allowed to increase indefinitely throughout the execution of the program. Although this policy will for the most part prevent "old" pages from having a huge influence in the future of the stream, it will still tend to respond slowly to locality changes.

7.10.2 Dynamic Page Replacement Algorithms

All of the static page replacement algorithms considered have one thing in common: they assumed that each program is allocated a fixed amount of memory when it begins execution, and does not request further memory during its lifetime. Although static algorithms will work in this scenario, they are hardly optimized to handle the common occurrence of adjusting to page allocation changes. This can lead to problems when a program rapidly switches between needing relatively large and relatively small page sets or localities. Depending on the size of the memory requirements of a program, the number of page faults may increase or decrease rapidly; for Stack Algorithms, you know that as the memory size is decreased, the numbers of page faults will increase. Other static algorithms may become completely unpredictable. Generally speaking, any program can have its number of page faults statistically analyzed for a variety of memory allocations. At some point the rate of increase of the page faults (derivative of the curve) will peak; this point is sometimes referred to as the hysteresis point. If the memory allocated to the program is less than the hysteresis point, the program is likely to thrash its page replacement. Past the point, there is generally little noticeable change in the fault rate, making the hysteresis the target page allocation. Since a full analysis is rarely available to a virtual memory controller, and that program behavior is quite dynamic, finding the optimal page allocation can be incredibly difficult. A variety of methods must be employed to develop replacement algorithms that work hand-in-hand with the locality changes present in complex programs. Dynamic paging algorithms accomplish this by attempting to predict program memory requirements, while adjusting available pages based on reoccurring trends. This policy of controlling available pages is also referred to as "prefetch" paging, and is contrary to the idea of demand paging. Although localities (within the scope of a set of operations) may change, states, it is likely that within the global locality (encompassing the smaller clusters), locality sets will be repeated.

7.11 Page Allocation Algorithm

How do you allocate the fixed amount of free memory among the various processes? If you have 93 free frames and two processes, how many frames does each process get? The simplest case of

Notes

virtual memory is the single-user system. Consider a single-user system with 128 KB memory composed of pages of size 1 KB. Thus, there are 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. You can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. You can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

Other variants are also possible, but the basic strategy is clear. The user process is allocated any free frame.



Task Least frequency is calculated throughout the reference stream, and its value can be calculated in a various ways. Discuss those ways.

7.12 Thrashing

Thrashing happens when a hard drive has to move its heads over the swap area many times due to the high number of page faults. This happens when memory accesses are causing page faults as the memory is not located in main memory. The thrashing happens as memory pages are swapped out to disk only to be paged in again soon afterwards. Instead of memory access happening mainly in main memory, access is mainly to disk causing the processes to become slow as disk access is required for many memory pages and thus thrashing.

The OS can reduce the effects of thrashing and improve performance by choosing a more suitable replacement strategy for pages. Having a replacement strategy that does not cause memory areas to be written to disk that have not modified since been retrieved reduces thrashing. Using replacement strategies that allow little used rarely accessed pages to remain in memory while the most required pages are swapped in and out.

Thrashing is a situation where large amounts of computer resources are used to do a minimal amount of work, with the system in a continual state of resource contention. Once started, thrashing is typically self-sustaining until something occurs to remove the original situation that led to the initial thrashing behavior.

Usually thrashing refers to two or more processes accessing a shared resource repeatedly such that serious system performance degradation occurs because the system is spending a disproportionate amount of time just accessing the shared resource. Resource access time may generally be considered as wasted, since it does not contribute to the advancement of any process. This is often the case when a CPU can process more information than can be held in available RAM; consequently the system spends more time preparing to execute instructions than actually executing them.

Concept of Thrashing

If the number of frames allocated to a low priority process is lower than the minimum number required by the computer architecture then in this case we must suspend the execution of this

Notes

low priority process. After this we should page out all of its remaining pages and freeing all of its allocated frames. This provision introduces a swap in, swap-out level of intermediate CPU scheduling. Let take a example of a process that does not have enough number of frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page fault. The only option remains here for process is to replace some active pages with the page that requires a frame. However, since all of 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 that mean replacing pages that it must bring back in immediately. This high paging activity is called Thrashing. Or we can say that a process is Thrashing if it is spending more time in paging then executing. Thrashing results in severe performance problems.

7.13 Summary

- The part of the operating system that manages the memory hierarchy is the memory manager.
- It keeps track of parts of memory that are in use and those that are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- Overlaying means replacement of a block of stored instructions or data with another. Overlay Manager is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.
- An address generated by the CPU is commonly referred to as a logical address and 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.
- Memory Management Unit (MMU) is a computer hardware component responsible for handling accesses to memory requested by the CPU. It is also known as Paged Memory Management Unit (PMMU).

7.14 Keywords

Logical Address: An address generated by the CPU is commonly referred to as a logical address.

Memory Management Unit (MMU): It is a computer hardware component responsible for handling accesses to memory requested by the CPU.

Memory Manager: The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.

Memory: It is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.

Overlay Manager: It is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.

Notes

Overlaying: It means replacement of a block of stored instructions or data with another.

Paged Memory Management Unit (PMMU): Same as MMU.

Physical Address: 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.

7.15 Self Assessment

Fill in the blanks:

1. The method assumes dividing a program into self-contained object code blocks called ....
2. The place in memory where an overlay is loaded is called a ..... region.
3. In multiprogramming several programs run at the same time on a ..... .
4. In ..... memory allocation method the memory manager places a process in the largest block of unallocated memory available.
5. To move a program from fast-access memory to a slow-access memory is known as ....
6. The process of translating virtual addresses into real addresses is called ..... .
7. Belady's Optimal Algorithm is also known as ..... .
8. ..... happens when a hard drive has to move its heads over the swap area many times due to the high number of page faults.
9. The full form of FINUFO is ..... .
10. EPROM stands for ..... .
11. RAM stands for ..... .
12. 1 MB equals to ..... .

7.16 Review Questions

1. Write a short description on:
   (a) Binding of Instructions and Data to Memory
   (b) Memory-Management Unit
   (c) CPU utilization
   (d) Memory Relocation
2. What is high-speed cache?
3. What is overlaying? Explain it.
4. Consider a logical address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.
5. How many bits are there in the logical address?
6. How many bits are there in the physical address?
7. Why are segmentation and paging sometimes combined into one scheme?
8. Describe a mechanism by which one segment could belong to the address space of two different processes.

Notes

9. Given memory partitions of l00K, 500K, 200K, 300K, and 600K (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?
10. Why is it that, on a system with paging, a process cannot access memory that it does not own? How could the operating system allow access to other memory? Why should it or should it not?
11. 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?
12. What is virtual memory? Explain the working of virtual memory.
13. Describe the dynamic page replacement method.

Answers: Self Assessment

1.	overlays	2.	destination	3.	uniprocessor
4.	worst fit	5.	swap out	6.	mapping
7.	perfect prediction	8.	Thrashing	9.	First-In Not-Used First-Out
10.	Erasable Programmable Read Only Memory			11.	Random Access Memory
12.	1024 KB				

7.17 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

Colin Ritchie, *Operating Systems*, BPB Publications.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 8: File Management

Notes

CONTENTS

Objectives

Introduction

8.1 File Systems

8.1.1 Types of File Systems

8.1.2 File Systems and Operating Systems

8.2 File Concept

8.3 Access Methods

8.4 Directory Structure

8.5 File System Mounting

8.6 File Sharing

8.7 Protection

8.8 File System Implementation

8.9 Allocation Methods

8.9.1 Contiguous Allocation

8.9.2 Linked Allocation

8.9.3 Indexed Allocation

8.10 Free-space Management

8.10.1 Bit-Vector

8.10.2 Linked List

8.10.3 Grouping

8.10.4 Counting

8.11 Directory Implementation

8.12 Summary

8.13 Keywords

8.14 Self Assessment

8.15 Review Questions

8.16 Further Readings

Objectives

After studying this unit, you will be able to:

- Define file systems
- Explain access methods
- Know directory structure
- Describe file system implementation
- Explain allocation methods

Notes

Introduction

Another part of the operating system is the file manager. While the memory manager is responsible for the maintenance of primary memory, the file manager is responsible for the maintenance of secondary storage (e.g., hard disks).

Each file is a named collection of data stored in a device. The file manager implements this abstraction and provides directories for organizing files. It also provides a spectrum of commands to read and write the contents of a file, to set the file read/write position, to set and use the protection mechanism, to change the ownership, to list files in a directory, and to remove a file. The file manager provides a protection mechanism to allow machine users to administer how processes executing on behalf of different users can access the information in files. File protection is a fundamental property of files because it allows different people to store their information on a shared computer, with the confidence that the information can be kept confidential.

8.1 File Systems

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data.

More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data. File systems share much in common with database technology, but it is debatable whether a file system can be classified as a special-purpose database (DBMS).

8.1.1 Types of File Systems

File system types can be classified into disk file systems, network file systems and special purpose file systems.

1. ***Disk file systems:*** A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.

   Example: Disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.

2. ***Flash file systems:*** A flash file system is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives.

   While a block device layer can emulate a disk drive so that a disk file system can be used on a flash device, this is suboptimal for several reasons:

   (a) *Erasing blocks:* Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.

   (b) *Random access:* Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.

   (c) *Wear leveling:* Flash memory devices tend to wear out when a single block is repeatedly overwritten; flash file systems are designed to spread out writes evenly.

Notes

Log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

3. ***Database file systems:*** A new concept for file management is the concept of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Example: dbfs.

4. ***Transactional file systems:*** Each disk operation may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

   Transaction processing introduces the guarantee that at any point while it is running, a transaction can either be finished completely or reverted completely (though not necessarily both at any given point). This means that if there is a crash or power failure, after recovery, the stored state will be consistent. (Either the money will be transferred or it will not be transferred, but it won't ever go missing "in transit".)

   This type of file system is designed to be fault tolerant, but may incur additional overhead to do so.

   Journaling file systems are one technique used to introduce transaction-level consistency to file system structures.

5. ***Network file systems:*** A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.

   Example: Network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

6. ***Special purpose file systems:*** A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space.

   Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the procfs (/proc) file system used by some Unix variants, which grants access to information about processes and other operating system features.

   Deep space science exploration craft, like Voyager I & II used digital tape based special file systems. Most modern space exploration craft like Cassini-Huygens used Real-time operating system file systems or RTOS influenced file systems. The Mars Rovers are one such example of an RTOS file system, important in this case because they are implemented in flash memory.

Task Discuss NTFS type of file system. Also explain the various benefits of NTFS file system over FAT file.

Notes

8.1.2 File Systems and Operating Systems

Most operating systems provide a file system, as a file system is an integral part of any modern operating system. Early microcomputer operating systems' only real task was file management - a fact reflected in their names. Some early operating systems had a separate component for handling file systems which was called a disk operating system. On some microcomputers, the disk operating system was loaded separately from the rest of the operating system. On early operating systems, there was usually support for only one, native, unnamed file system; for example, CP/M supports only its own file system, which might be called "CP/M file system" if needed, but which didn't bear any official name at all.

Because of this, there needs to be an interface provided by the operating system software between the user and the file system. This interface can be textual (such as provided by a command line interface, such as the Unix shell, or OpenVMS DCL) or graphical (such as provided by a graphical user interface, such as file browsers). If graphical, the metaphor of the folder, containing documents, other files, and nested folders is often used.

Flat file systems: In a flat file system, there are no subdirectories-everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc. While simple, this system rapidly becomes inefficient as the number of files grows, and makes it difficult for users to organise data into related groups.

Like many small systems before it, the original Apple Macintosh featured a flat file system, called Macintosh File System. Its version of Mac OS was unusual in that the file management software (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of MFS. This structure meant that every file on a disk had to have a unique name, even if it appeared to be in a separate folder. MFS was quickly replaced with Hierarchical File System, which supported real directories.

8.2 File Concept

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file. Whether you copied a file from elsewhere or created your own, you will need to return to it later in order to edit its contents.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sector, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called "clusters" or "blocks" which contain a certain number of disk sectors (usually 1-64). This is the smallest logical amount of disk space that can be allocated to hold a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g, from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

Notes

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's meta-data was changed.



Note Many early PC operating systems did not keep track of file times. Other information can include the file's device type (e.g., block, character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts.



Example: Interprocess pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.

Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

8.3 Access Methods

There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, many different access methods are supported.

Sequential Access

Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, computer editors usually access files in this fashion. A read operation reads the next portion of the file and automatically advances the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and, on some systems, a program may be able to skip forward or backward n records, for some integer n. This scheme is known as sequential access to a file. Sequential access is based on a tape model of a file.

A sequential file may consist of either formatted or unformatted records. If the records are formatted, you can use formatted I/O statements to operate on them. If the records are

Notes

unformatted, you must use unformatted I/O statements only. The last record of a sequential file is the end-of-file record.

Direct Access

Direct access is based on a disk model of a file. For direct access, the file is viewed as a numbered sequence of block or records. A direct-access file allows arbitrary blocks to be read or written. Thus, after block 18 has been read, block 57 could be next, and then block 3. There are no restrictions on the order of reading and writing for a direct access file. Direct access files are of great use for intermediate access to large amounts of information.

The file operations must be modified to include the block number as a parameter. Thus, you have "read n", where n is the block number, rather than "read next", and "write n", rather that "write next". An alternative approach is to retain "read next" and "write next" and to add an operation; "position file to n" where n is the block number. Then, to effect a "read n", you would issue the commands "position to n" and then "read next".

Not all OS support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

Direct-access files support both formatted and unformatted record types. Both formatted and unformatted I/O work exactly as they do for sequential files.

Other Access Methods

Other access methods can be built on top of a direct-access method. These additional methods generally involve the construction of an index for a file. The index contains pointers to the various blocks. To find an entry in the file, the index is searched first and the pointer is then used to access the file directly to find the desired entry. With a large file, the index itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items. For example, IBM's indexed sequential access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, I first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most direct access reads.

8.4 Directory Structure

The directories themselves are simply files indexing other files, which may in turn be directories if a hyerarchical indexing scheme is used. In order to protect the integrity of the file system in spite of user of program error, all modifications to these particular directory files are commonly restricted to the file management system. The typical contents of a directory are:

1. file name (string uniquely identifying the file), type (e.g. text, binary data, executable, library), organization (for systems that support different organizations);
2. device (where the file is physically stored), size (in blocks), starting address on device (to be used by the device I/O subsystem to physically locate the file);
3. creator, owner, access information (who is allowed to access the file, and what they may do with it);

Notes

4. date of creation/of last modification;

5. locking information (for the system that provide file/record locking).

As far as organization, by far the most common scheme is the hyerarchical one: a multi-level indexing scheme is used, in which a top-level directory indexes both files and other directories, which in turn index files and directories, and so on. Usually this scheme is represented in the form of a tree.

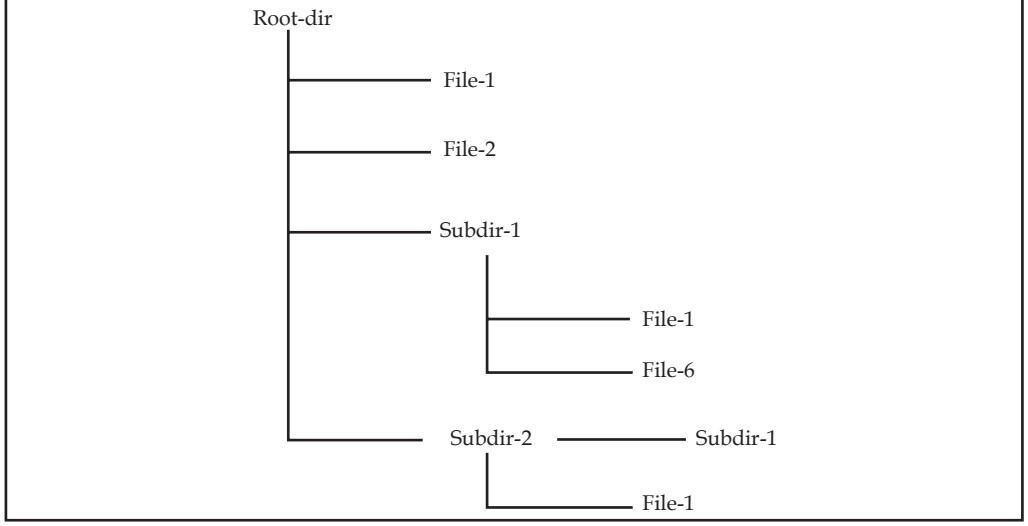
The hyerarchical architecture has disctinct advantages over a simple, one-level indexing one: the tree structure can be effectively used to reflect a logical organization of the data stored in the files; names can be reused (they must uniquely identify files within each directory, not across the whole fle system); in a multi-user system, name conflicts between files owned by different users can be solved by assigning to each user a directory for her own files and sub-directories, the so called user's "home" directory.

A complete indexing of a file is obtained by navigating the tree starting from the top-level, "root", directory, and walking along a path to the tree leaf coresponding to the file.

A "pathname" is thus obtained, which uniquely identifies the file within the whole file system.

Example: The pathname for file "File-6" in Figure 8.1 is "Root-dir:Subdir-1:File-6", where a colon is used to separate tree nodes.

Figure 8.1: Tree Representation of a Hyerarchical Directory Structure



A complete pathname is not the only way to identify a file in the directory tree structure: a "relative" pathname, starting from a parent directory is suited just as well, provided that the FMS already knows about that directory. This addressing methods can be usefully exploited by making the FMS assign to all processes a "current working directory" (CWD) attribute, i.e. the complete patname of a directory of interest, and defining a way for the process to identify files by just specifying a "relative" pathname starting from that directory. In the same example, if ":Root-dir:Subdir-1" is the CWD of a process, the above file might be identified simply as "File-6", using the convention that patnames not starting with a color are relative to the CWD. The advantage is twofold: the entire file system structure up to the CWD need not be known by a program (hence its data can be safely moved in other directories withouth having to rewrite the program), and file access time is decreased, since it's no longer necessary to navigate the whole tree in order to find the address of a file.

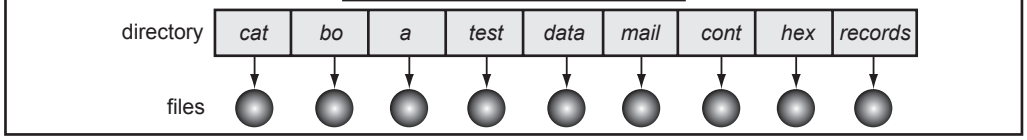
Notes

Single Level Directory

In single level directory all files are contained in the same directory. It is easy to support and understand. It has some limitations like:

1. Large number of files (naming).
2. Ability to support different users/topics (grouping).

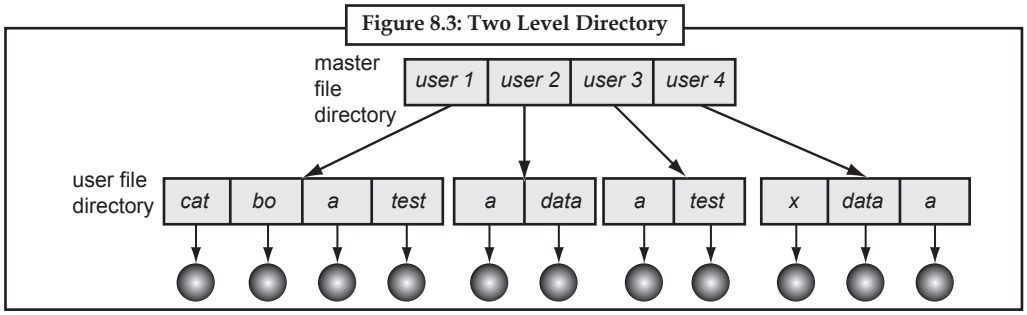
Figure 8.2: Single Level Directory



Two Level Directory

In two level directory structure one is master file directory and the other is user file directory. Here each user has their own user file directory. Each entry in the master file directory points to a user file directory. Each user has rights to access their own directory but can't access other user's directory, if permission is not given by the owner of the second one.

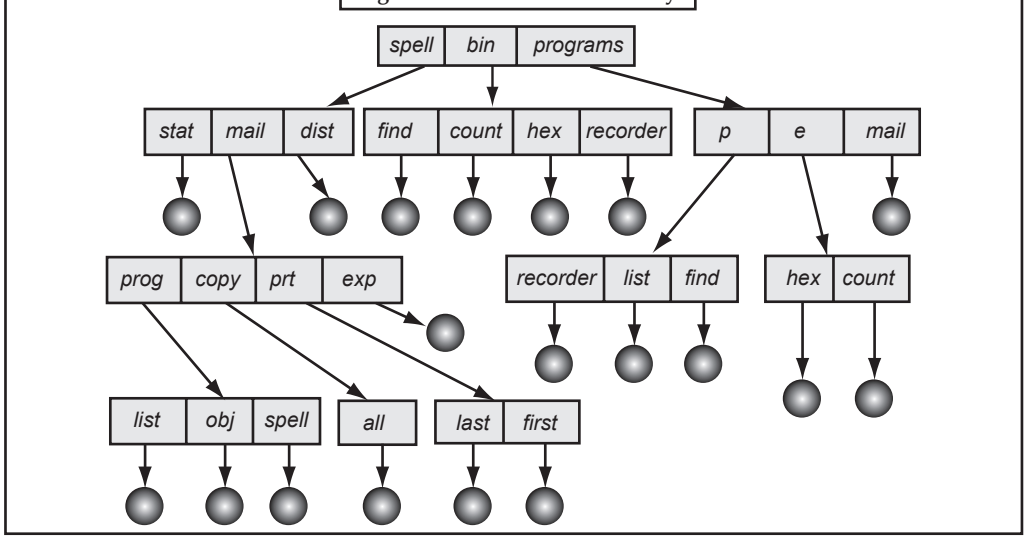
Figure 8.3: Two Level Directory



Three Level Directory

In three level directory the directory structure is a tree with arbitrary height. Here users may create their own subdirectories.

Figure 8.4: Three Level Directory



8.5 File System Mounting

Notes

The file system structure is the most basic level of organization in an operating system. Almost all of the ways an operating system interacts with its users, applications, and security model are dependent upon the way it organizes files on storage devices. Providing a common file system structure ensures users and programs are able to access and write files.

File systems break files down into two logical categories:

1. Shareable vs. unsharable files
2. Variable vs. static files

Shareable files are those that can be accessed locally and by remote hosts; unsharable files are only available locally. Variable files, such as documents, can be changed at any time; static files, such as binaries, do not change without an action from the system administrator.

The reason for looking at files in this manner is to help correlate the function of the file with the permissions assigned to the directories which hold them. The way in which the operating system and its users interact with a given file determines the directory in which it is placed, whether that directory is mounted with read-only or read/write permissions, and the level of access each user has to that file. The top level of this organization is crucial. Access to the underlying directories can be restricted or security problems could manifest themselves if, from the top level down, it does not adhere to a rigid structure.

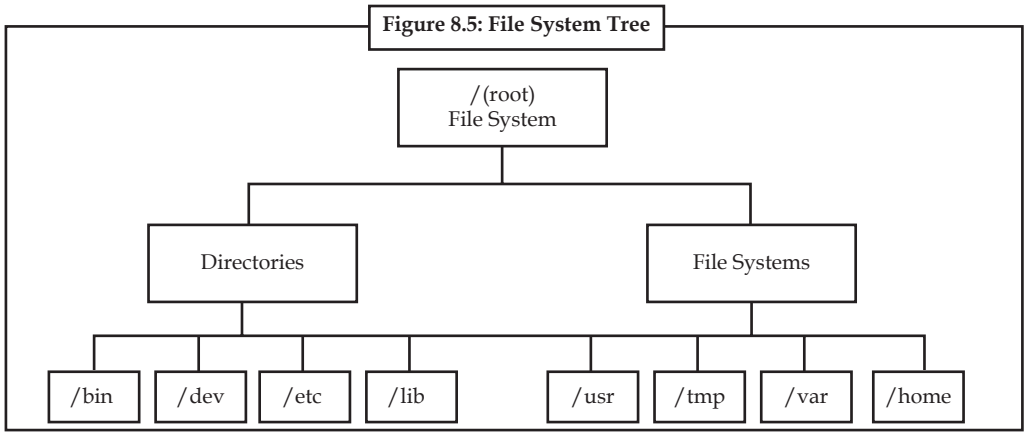
It is important to understand the difference between a file system and a directory. A file system is a section of hard disk that has been allocated to contain files. This section of hard disk is accessed by mounting the file system over a directory. After the file system is mounted, it looks just like any other directory to the end user.

However, because of the structural differences between the file systems and directories, the data within these entities can be managed separately.

When the operating system is installed for the first time, it is loaded into a directory structure, as shown in the following illustration.

Figure 8.5 File System Tree. This tree chart shows a directory structure with the/(root) file system at the top, branching downward to directories and file systems. Directories branch to/bin, /dev, /etc, and /lib. File systems branch to /usr, /tmp, /var, and /home.

Figure 8.5: File System Tree



The directories on the right (/usr, /tmp, /var, and /home) are all file systems so they have separate sections of the hard disk allocated for their use. These file systems are mounted automatically when the system is started, so the end user does not see the difference between these file systems and the directories listed on the left (/bin, /dev, /etc, and /lib).

Notes

On standalone machines, the following file systems reside on the associated devices by default:

/File System	/Device
/dev/hd1	/home
/dev/hd2	/usr
/dev/hd3	/tmp
/dev/hd4	/(root)
/dev/hd9var	/var
/proc	/proc
/dev/hd10opt	/opt

The file tree has the following characteristics:

1. Files that can be shared by machines of the same hardware architecture are located in the /usr file system.
2. Variable per-client files, for example, spool and mail files, are located in the /var file system.
3. The /(root) file system contains files and directories critical for system operation.



Example: It contains

   (a) A device directory (/dev)

   (b) Mount points where file systems can be mounted onto the root file system, for example, /mnt

4. The /home file system is the mount point for users' home directories.
5. For servers, the /export directory contains paging-space files, per-client (unshared) root file systems, dump, home, and /usr/share directories for diskless clients, as well as exported /usr directories.
6. The/proc file system contains information about the state of processes and threads in the system.
7. The/opt file system contains optional software, such as applications.

The following list provides information about the contents of some of the subdirectories of the /(root) file system.

/bin	Symbolic link to the /usr/bin directory.
/dev	Contains device nodes for special files for local devices. The /dev directory contains special files for tape drives, printers, disk partitions, and terminals.
/etc	Contains configuration files that vary for each machine. Examples include: 1. /etc/hosts 2. /etc/passwd
/export	Contains the directories and files on a server that are for remote clients.
/home	Serves as a mount point for a file system containing user home directories. The /home file system contains per-user files and directories. In a standalone machine, a separate local file system is mounted over the /home directory. In a network, a server might contain user files that should be accessible from several machines. In this case, the server's copy of the /home directory is remotely mounted onto a local /home file system.
/lib	Symbolic link to the /usr/lib directory, which contains architecture-independent libraries with names in the form lib*.a.
	Contd....

Notes

/sbin	Contains files needed to boot the machine and mount the /usr file system. Most of the commands used during booting come from the boot image's RAM disk file system; therefore, very few commands reside in the /sbin directory.
/tmp	Serves as a mount point for a file system that contains system-generated temporary files.
/u	Symbolic link to the /home directory.
/usr	Serves as a mount point for a file system containing files that do not change and can be shared by machines (such as executable programs and ASCII documentation). Standalone machines mount a separate local file system over the /usr directory. Diskless and disk-poor machines mount a directory from a remote server over the /usr file system.
/var	Serves as a mount point for files that vary on each machine. The /var file system is configured as a file system because the files that it contains tend to grow. For example, it is a symbolic link to the /usr/tmp directory, which contains temporary work files.



Task How will you mount a file? Explain.

8.6 File Sharing

In today's world where the working is a multiuser environment a file is required to be shared among more than one users. There are several techniques and approaches to effects this operation. Simple approach is to copy the file at the users local hard disk. This approach essentially creates to different files, in therefore cannot be treated as file sharing.

A file can be shared in three different modes:

1. ***Read only:*** The user can only read or copy the file.
2. ***Linked shared:*** *All* the users can share the file and can make the changes but the changes are reflected in the order defined by the operating systems.
3. ***Exclusive mode:*** The file is acquired by one single user who can make the changes while others can only read or copy it.

Sharing can also be done through symbolic links, but there occurs certain problems like concurrent updation problem, deletion problem. Updation cannot be done simultaneously by two users at a time, also one cannot delete a file if it in use by another user. The solution for this problem is done by locking file techniques.

8.7 Protection

The data in the computer system should be protected and kept secure. A major concern is to protect data from both physical damage (reliability) and improper access (protection). There is a mechanisms in the computer system that a system program or manually it can take the backup or duplicate the files automatically. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes. Also the data can be lost due to bugs on system. Protection can be provided in many ways. For a small single-user system, you might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

8.8 File System Implementation

The most important issue in file storage is keeping track of which disk blocks go with which file. Different operating systems use different methods - contiguous allocation and linked list

Notes

allocation are important to know. In the former, each file is stored as a contiguous block of data on the disk, in the latter, the file is kept as a linked list of disk blocks - the first word of each block is used as a pointer to the next one. UNIX uses i-nodes to keep track of which blocks belong to each file. An i-node is a table that lists the attributes and disk addresses of the file's blocks. The first few disk addresses are stored in the i-node itself, so for small files, all the necessary information is in the i-node itself which is fetched from disk to main memory when the file is opened. For larger files, one of the addresses in the i-node is the address of a disk block called a single indirect block which contains additional disk addresses. If this is insufficient, another address called the double indirect block may contain the address of a block that contains a list of single indirect blocks.

In order to create the illusion of files from the block oriented disk drives, the OS must keep track of the location of the sectors containing the data of the file. This is accomplished by maintaining a set of data structures both in memory and on disk that keep track of where data is allocated to each file, and the name to file mapping encoded in the directory structure.

The simplest allocation of files is a contiguous allocation of sectors to each file. A directory entry would contain the name of the file, its size in bytes and the first and last sector of the file. This results in a fast read of a given file and a compact representation, but also of sizable external fragmentation which can require compaction to correct. The analog in memory management is the base/limit register system of memory allocation.

As with memory management, you turn to more complex data structures and non contiguous allocation to solve the problems. I can use a bitmap to record the allocated and unallocated sectors on the disk, and keep a list of sectors assigned to each file in its directory entry. This isn't often used, because it makes searching for free space difficult and replicates the allocation information in the file itself. (When it is used, the bitmap is kept both on disk and in memory).

The other system that is used in file systems and in memory management is a linked list. A simple mechanism is to take one integer out of every file block and us that as the next sector following this one (similar to linking the holes in memory management).

This is an improvement over bitmaps in efficiency of storage use, but has a significant drawback in that finding the proper sector for a random access is expensive. Finding the right sector containing a random sector is as expensive as reading to that point in the file.

To solve this we collect the sector pointers into a table (usually cached in main memory) separate from the files. Now the OS can follow the separate pointers to find the appropriate sector for a random access without reading each disk block. Furthermore, the conceptual disk blocks and the physical disk blocks now have the same size. This is essentially the FAT file system of MS-DOS.

Another organization is one optimized for small files (which research has shown dominate the file system, in the sense that most files are small) while accommodating large ones. The system is called the index node or i-node system. An i-node contains the attributes of the file and pointers to its first few blocks.

The last 3 sector pointers are special. The first points to inode structures that contain only pointers to sectors; this is an indirect block. The second to pointers to pointers to sectors (a double indirect node) and the third to pointers to pointers to sectors (triple indirect).

This results in increasing access times for blocks later in the file. Large files will have longer access times to the end of the file. I-nodes specifically optimize for short files.

8.9 Allocation Methods

One main problem in file management is how to allocate space for files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk

Notes

space are contiguous, linked, and indexed. Each method has its advantages and disadvantages. Accordingly, some systems support all three (e.g. Data General's RDOS). More commonly, a system will use one particular method for all files.

8.9.1 Contiguous Allocation

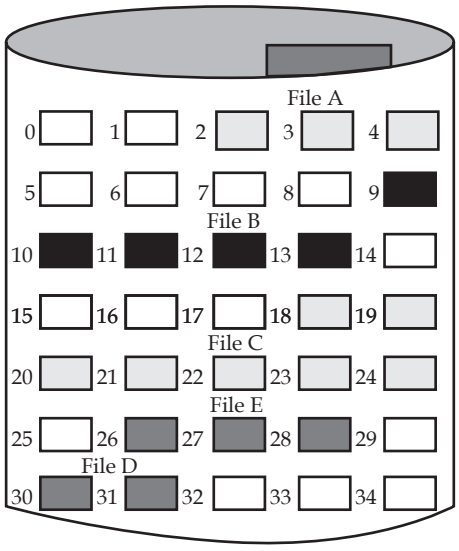
The contiguous allocation method requires each file to occupy a set of contiguous address on the disk. Disk addresses define a linear ordering on the disk. Notice that, with this ordering, accessing block b+1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files in minimal, as is seek time when a seek is finally needed. Contiguous allocation of a file is defined by the disk address and the length 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.

The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks. First-fit, best-fit, and worst-fit strategies 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 better than worst-fit in terms of both time storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms also suffer from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but this space not contiguous; storage is fragmented into a large number of small holes.

Another problem with contiguous allocation is determining how much disk space is needed for a file. When the file is created, the total amount of space it will need must be known and allocated. How does the creator (program or person) know the size of the file to be created. In some cases, this determination may be fairly simple (e.g. copying an existing file), but in general the size of an output file may be difficult to estimate.

Figure 8.6: Diagram of Contiguous Allocation



directory

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Since blocks are allocated contiguously, external fragmentation may occur. Thus, compaction may be needed.

Notes

8.9.2 Linked Allocation

The problems in contiguous allocation can be traced directly to the requirement that the spaces be allocated contiguously and that the files that need these spaces are of different sizes. These requirements can be avoided by using linked allocation.

In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer. The value -1 may be used for NIL to differentiate it from block 0.

With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. A write to a file removes the first free block and writes to that block. This new block is then linked to the end of the file. To read a file, the pointers are just followed from block to block.

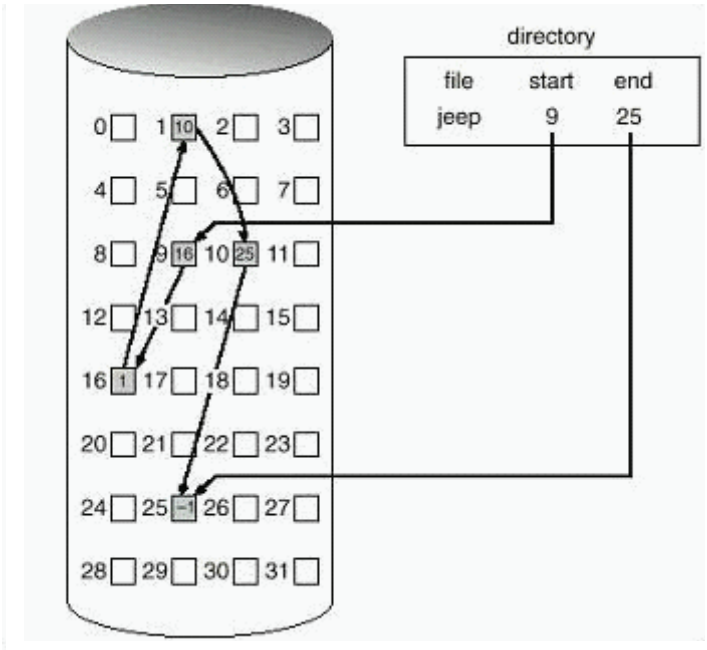
There is no external fragmentation with linked allocation. Any free block can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Linked allocation, does have disadvantages, however. The major problem is that it is inefficient to support direct-access; it is effective only for sequential-access files. To find the ith block of a file, it must start at the beginning of that file and follow the pointers until the ith block is reached.



Note Note that each access to a pointer requires a disk read.

Another severe problem is reliability. A bug in OS or disk hardware failure might result in pointers being lost and damaged. The effect of which could be picking up a wrong pointer and linking it to a free block or into another file.

Figure 8.7: Diagram of Linked Allocation

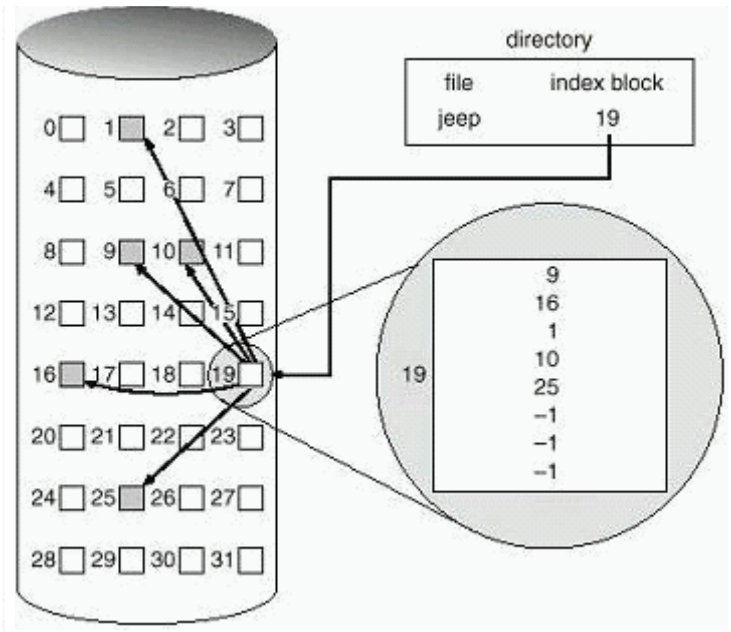


Notes

8.9.3 Indexed Allocation

The indexed allocation method is the solution to the problem of both contiguous and linked allocation. This is done by bringing all the pointers together into one location called the index block. Of course, the index block will occupy some space and thus could be considered as an overhead of the method. In indexed allocation, each file has its own index block, which is an array of disk sector of addresses. The ith entry in the index block points to the ith sector of the file. The directory contains the address of the index block of a file. To read the ith sector of the file, the pointer in the ith index block entry is read to find the desired sector. Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

Figure 8.8: Diagram of Indexed Allocation



Task File management is a big problem is operating system. How it will be resolved?

8.10 Free-space Management

Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a free-space list. The free-space list records all disk blocks that are free (i.e., are not allocated to some file). To create a file, the free-space list has to be searched for the required amount of space, and allocate that space to a 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.

8.10.1 Bit-Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1.

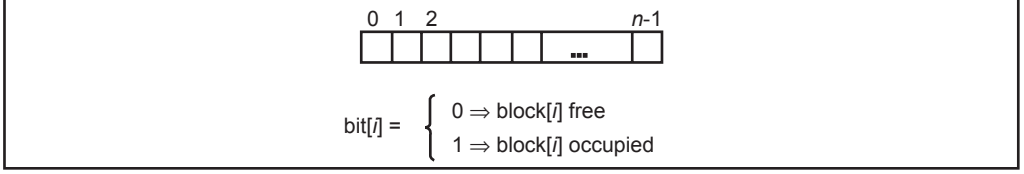
Notes

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:

11000011000000111001111110001111…

The main advantage of this approach is that it is relatively simple and efficient to find n consecutive free blocks on the disk. Unfortunately, bit vectors are inefficient unless the entire vector is kept in memory for most accesses. Keeping it main memory is possible for smaller disks such as on microcomputers, but not for larger ones.

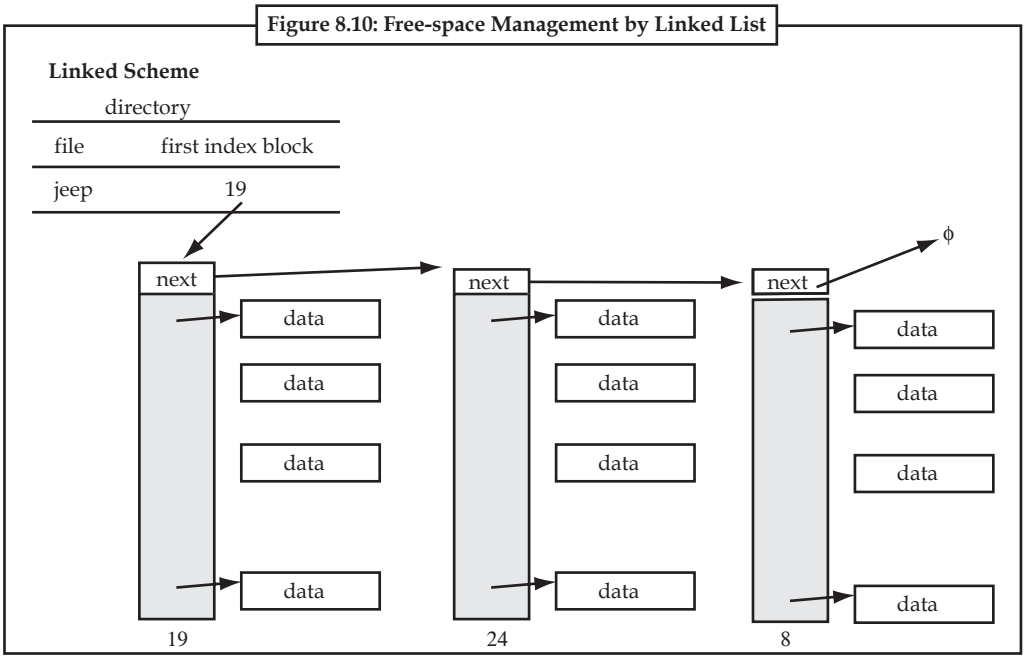
Figure 8.9: Free-space Management by Bit-Vector



8.10.2 Linked List

Another approach is to link all the free disk blocks together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on. In the previous example, a pointer could be kept to 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, and so on. This scheme is not efficient; to traverse the list, each block must be read, which requires substantial I/O time.

Figure 8.10: Free-space Management by Linked List



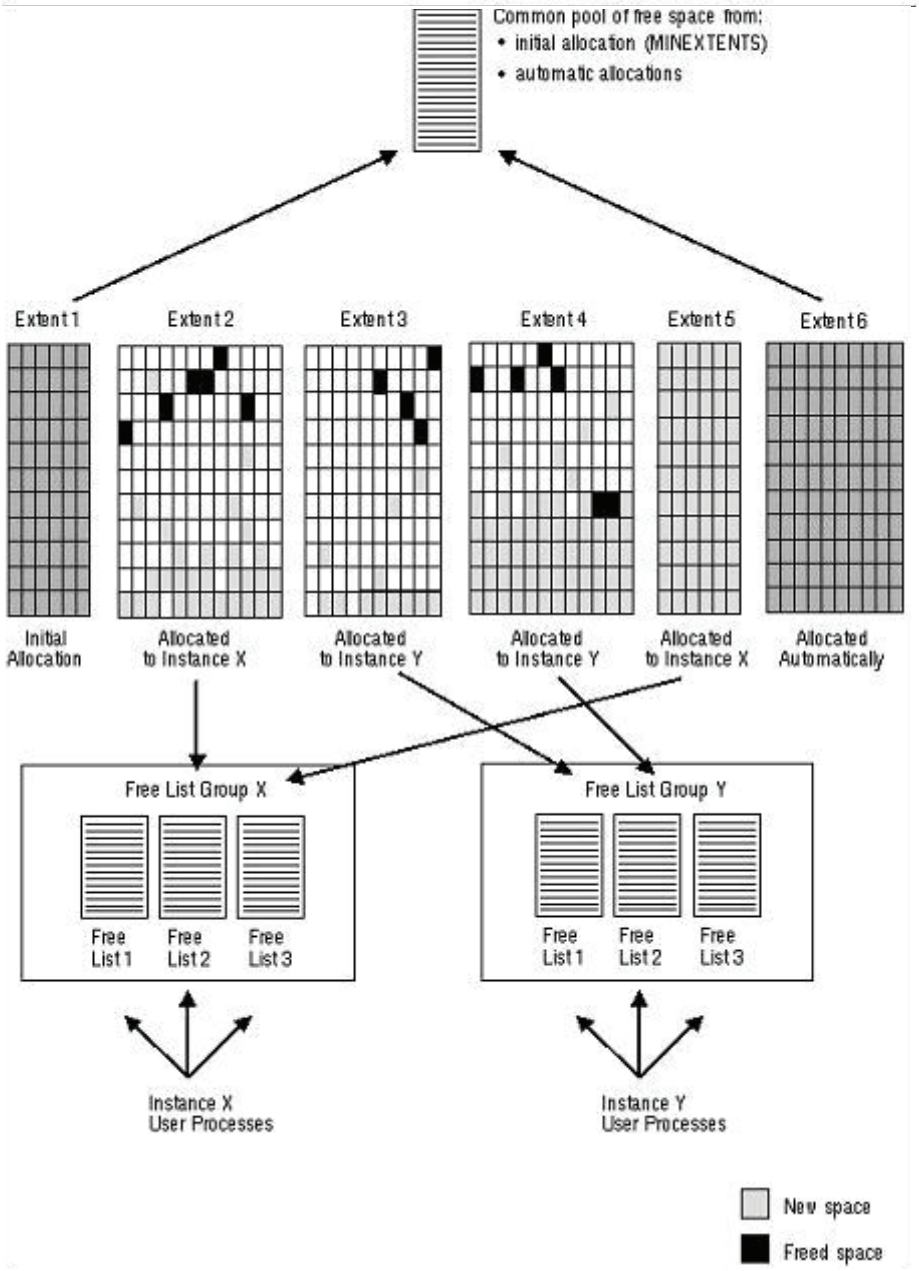
8.10.3 Grouping

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 are actually free. The last one is the disk address of another block

containing addresses of another n free blocks. The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Notes

Figure 8.11: Free-space Management by Free List Grouping



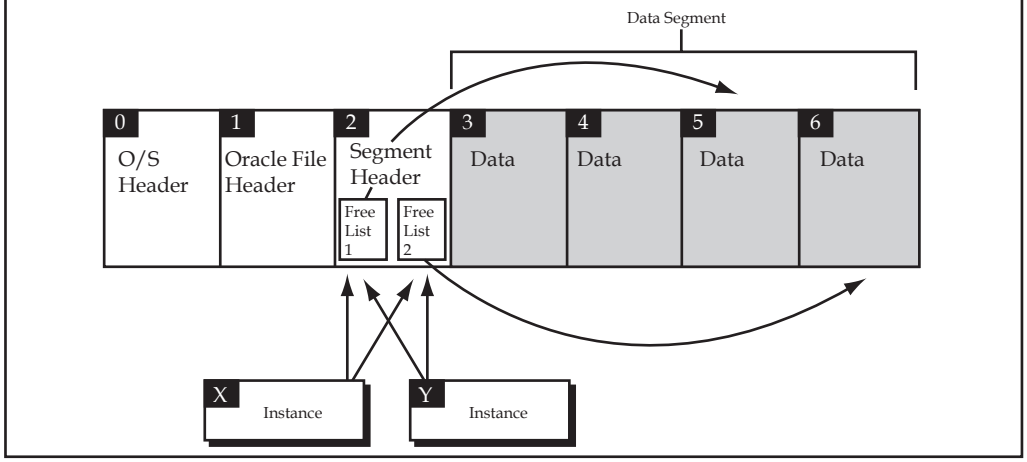
8.10.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept 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

Notes

a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

Figure 8.12: Free-space Management by Free Block Counting



8.11 Directory Implementation

Directories are generally simply files with a special interpretation. Some directory structures contain the name of a file, its attributes and a pointer3 either into its FAT list or to its i-node.

This choice bears directly on the implementation of linking. If attributes are stored directly in the directory node, (hard) linking is difficult because changes to the file must be mirrored in all directories. If the directory entry simply points to a structure (like an i-node) that holds the attributes internally, only that structure needs to be updated.

The simplest method is to use a linear list of file names with pointers to the data blocks. This requires a linear search to find a particular entry. Hash tables are also used by some operating systems - a linear list stores the directory entries but a hash function based on some computation from the file name returns a pointer to the file name in the list. Thus, directory search time is greatly reduced.

In UNIX, each entry in the directory contains just a file name and its i-node number. When a file is opened, the file system takes the file name and locates its disk blocks. The i-node is read into memory and kept there until the file is closed.

8.12 Summary

- File is a named collection of data stored in a device.
- File manager is an integral part of the operating system which is responsible for the maintenance of secondary storage.
- File system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.
- Disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.
- Flash file system is a file system designed for storing files on flash memory devices. Network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.

Notes

- Flat file system is a file system where is no subdirectories and everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc.
- Directory is simple file containing the indexing of other files, which may in turn be directories if a hyerarchical indexing scheme is used.

8.13 Keywords

Directory: It is simple file containing the indexing of other files, which may in turn be directories if a hyerarchical indexing scheme is used.

Disk file system: It is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.

File manager: It is an integral part of the operating system which is responsible for the maintenance of secondary storage.

File system: It is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.

File: It is a named collection of data stored in a device.

Flash file system: It is a file system designed for storing files on flash memory devices.

Flat file system: It is a file system where no subdirectories are present and everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc.

Network file system: It is a file system that acts as a client for a remote file access protocol, providing access to files on a server.

8.14 Self Assessment

Fill in the blanks:

1. Shareable files are those that can be accessed ..... and by ..... .
2. Three major methods of allocating disk space are ....., ... and ..... .
3. The difficulty with contiguous allocation is ..... for a new file.
4. There is no external fragmentation with ..... allocation.
5. FAT stands for ..... .
6. NTFS stands for ..... .
7. Direct access is based on a ..... of a file.
8. ..... files support both formatted and unformatted record types.
9. A ..... system is a file system designed for storing files on flash memory devices.
10. A ..... is a collection of letters, numbers and special characters.
11. The hierarchical file system was an early research interest of ..... .

8.15 Review Questions

1. What is a directory? Can we consider a directory as a file? Explain your answer.
2. What is a flash file system? Give an example of it.

Notes

3. What are the differences between file system and file manager?
4. Write short notes on:
   (a) Disk file system
   (b) Flat file system
   (c) Network file system
5. What are the differences between a file system and a directory?
6. What is logical damage of data? How can it be recovered?
7. Write a short note on free space management.
8. What is indexed allocation method? How differs does it from linked list allocation?
9. Compare and contrast between contiguous disk space allocation method and linked list allocation method.
10. What is disk scheduling? Describe different disk scheduling policies.

Answers: Self Assessment

1.	locally, remote hosts	2.	contiguous, linked, and indexed		
3.	finding space	4.	linked	5.	File allocation table
6.	NT File system	7.	disk model	8.	direct-access
9.	flash file	10.	file	11.	Dennis Ritchie

8.16 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 9: I/O & Secondary Storage Structure

Notes

CONTENTS

Objectives
Introduction
9.1 I/O Systems
9.2 I/O Hardware
9.2.1 Input Device
9.2.2 Output Device
9.3 Application I/O Interface
9.4 Functions of I/O Interface
9.5 Kernel I/O Sub-system
9.6 Disk Scheduling
9.7 Disk Management
9.8 Swap Space Management
9.8.1 Pseudo-Swap Space
9.8.2 Physical Swap Space
9.8.3 Three Rules of Swap Space Allocation
9.9 RAID Structure
9.10 Summary
9.11 Keywords
9.12 Self Assessment
9.13 Review Questions
9.14 Further Readings

Objectives

After studying this unit, you will be able to:

- Define I/O systems
- Describe I/O hardware
- Explain application I/O interface
- Know disk scheduling
- Describe swap space management
- Explain RAID structure

Introduction

The central processing unit is the unseen part of a computer system, and users are only dimly aware of it. But users are very much aware of the input and output associated with the computer. They submit input data to the computer to get processed information, the output. One of the important tasks of the operating system is to control all of the I/O devices, such as issuing commands concerning data transfer or status polling, catching and processing interrupts as well

Notes

as handling different kind of errors. In this unit, we shall discuss how operating system handles the inputs and outputs.

9.1 I/O Systems

Input/output, or I/O, refers to the communication between an information processing system (such as a computer), and the outside world – possibly a human, or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, keyboards and mouses are considered input devices of a computer, while monitors and printers are considered output devices of a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

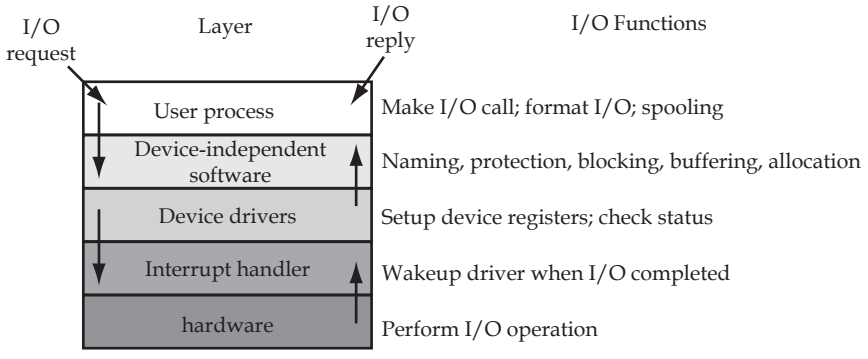


Note The designation of a device as either input or output depends on the perspective. Mouses and keyboards take as input physical movement that the human user outputs and convert it into signals that a computer can understand. The output from these devices is input for the computer. Similarly, printers and monitors take as input signals that a computer outputs. They then convert these signals into representations that human users can see or read. (For a human user the process of reading or seeing these representations is receiving input.)

In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the heart of a computer, and from that point of view any transfer of information from or to that combination, for example to or from a disk drive, is considered I/O. The CPU and its supporting circuitry provide I/O methods that are used in low-level computer programming in the implementation of device drivers.

Higher-level operating system and programming facilities employ separate, more abstract I/O concepts and primitives. For example, most operating systems provide application programs with the concept of files. The C and C++ programming languages, and operating systems in the Unix family, traditionally abstract files and devices as streams, which can be read or written, or sometimes both. The C standard library provides functions for manipulating streams for input and output.

Figure 9.1: I/O System



Notes

9.2 I/O Hardware

I/O devices allow your managed system to gather, store, and transmit data. I/O devices are found in the server unit itself and in expansion units and towers that are attached to the server. I/O devices can be embedded into the unit, or they can be installed into physical slots.

Not all types of I/O devices are supported for all operating systems or on all server models.

Example: Switch Network Interface (SNI) adapters are supported only on certain server models, and are not supported for i5/OS® logical partitions.

9.2.1 Input Device

A hardware device that sends information into the CPU is known as input device. Without any input devices a computer would simply be a display device and not allow users to interact with it, much like a TV. Below is a listing of different types of computer input devices.

Keyboard: One of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.

Figure 9.2: Keyboard



Mouse: An input device that allows an individual to control a mouse pointer in a graphical user interface (GUI). Utilizing a mouse a user has the ability to perform various functions such as opening a program or file and does not require the user to memorize commands, like those used in a text-based environment such as MS-DOS. To the right is a picture of a Microsoft IntelliMouse and is an example of what a mouse may look like.

Figure 9.3: Mouse



Touch Screen: A touch screen is a display which can detect the presence and location of a touch within the display area. The term generally refers to touch or contact to the display of the device by a finger or hand. Touch screens can also sense other passive objects, such as a stylus. However, if the object sensed is active, as with a light pen, the term touch screen is generally not applicable. The ability to interact directly with a display typically indicates the presence of a touch screen.

Notes

Figure 9.4: Touch Screen



Joystick: A joystick is an input device consisting of a stick that pivots on a base and reports its angle or direction to the device it is controlling. Joysticks are often used to control video games, and usually have one or more push-buttons whose state can also be read by the computer. A popular variation of the joystick used on modern video game consoles is the analog stick.

Figure 9.5: Joystick



Scanner: Hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object. A scanner is commonly connected to a computer USB, Firewire, Parallel or SCSI port.

Figure 9.6: Scanner



Microphone: Sometimes abbreviated as mic, a microphone is a hardware peripheral that allows computer users to input audio into their computers.

Figure 9.7: Microphone



Notes

Webcam: A camera connected to a computer or server that allows anyone connected to the Internet to view still pictures or motion video of a user. The majority of webcam web sites are still pictures that are frequently refreshed every few seconds, minutes, hours, or days. However, there are some sites and personal pages that can supply streaming video for users with broadband.

Figure 9.8: Webcam



Digital camera: A type of camera that stores the pictures or video it takes in electronic format instead of to film. There are several features that make digital cameras a popular choice when compared to film cameras. First, the feature often enjoyed the most is the LCD display on the digital camera. This display allows users to view photos or video after the picture or video has been taken, which means if you take a picture and don't like the results, you can delete it; or if you do like the picture, you can easily show it to other people. Another nice feature with digital cameras is the ability to take dozens, sometimes hundreds of different pictures.

Figure 9.9: Digital Camera



9.2.2 Output Device

Any peripheral that receives and/or displays output from a computer is known as output device. Below are some examples of different types of output devices commonly found on a computer.

Prointer: A printer is a peripheral which produces a hard copy (permanent human-readable text and/or graphics) of documents stored in electronic form, usually on physical print media such as paper or transparencies. Many printers are primarily used as local peripherals, and are attached by a printer cable or, in most newer printers, a USB cable to a computer which serves as a document source. Some printers, commonly known as network printers, have built-in network interfaces (typically wireless or Ethernet), and can serve as a hardcopy device for any user on the network. Individual printers are often designed to support both local and network connected users at the same time.

Figure 9.10: Printer



Notes

Plotter: A plotter is a vector graphics printing device to print graphical plots, that connects to a computer. There are two types of main plotters. Those are pen plotters and electrostatic plotters.

Figure 9.11: Plotter



Fax: Fax (short for facsimile, from Latin fac simile, "make similar", i.e. "make a copy") is a telecommunications technology used to transfer copies (facsimiles) of documents, especially using affordable devices operating over the telephone network. The word telefax, short for telefacsimile, for "make a copy at a distance", is also used as a synonym. Although fax is not an acronym, it is often written as "FAX".

Figure 9.12: Fax



Monitors: A visual display unit, often called simply a monitor or display, is a piece of electrical equipment which displays images generated from the video output of devices such as computers, without producing a permanent record. Most new monitors typically consist of a TFT LCD, with older monitors based around a cathode ray tube (CRT). Almost all of the mainstream new monitors being sold on market now are LCD.

Figure 9.13: Monitor



Speakers: A hardware device connected to a computer's sound card that outputs sounds generated by the card.

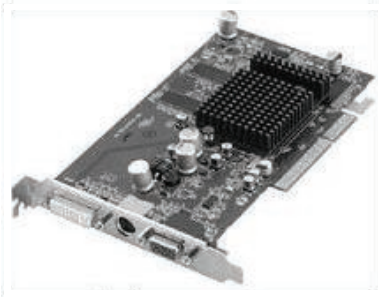
Notes

Figure 9.14: Speaker



Video card: Also known as a graphics card, video card, video board, or a video controller, a video adapter is an internal circuit board that allows a display device, such as a monitor, to display images from the computer.

Figure 9.15: Video Card



Task Discuss about any four I/O hardware devices.

9.3 Application I/O Interface

The I/O and memory interface are the counterparts to the bus control logic. What goes between the bus control logic and interface is simply the conductors in the bus; therefore, the interface must be designed to accept and send signals that are compatible with the bus control logic and its timing. Although there are similarities in I/O interfaces, there are also significant differences.

An I/O interface must be able to:

1. Interpret the address and memory-I/O select signals to determine whether or not it is being referenced and, if so, determine which of its registers is being accessed.
2. Determine whether an input or output is being conducted and accept output data or control information from the bus or place input data or status information on the bus.
3. Input data from or output data to the associated I/O device and convert the data from parallel to the format acceptable to the I/O device, or vice versa.
4. Send a ready signal when data have been accepted from or placed on the data bus, thus informing the processor that a transfer has been completed.
5. Send interrupt requests and, if there is no interrupt priority management in the bus control logic, receive interrupt acknowledgments and send an interrupt type.
6. Receive a reset signal and reinitialize itself and, perhaps, its associated device.

Notes

Figure 9.16 Typical Block Diagram of an I/O Interface

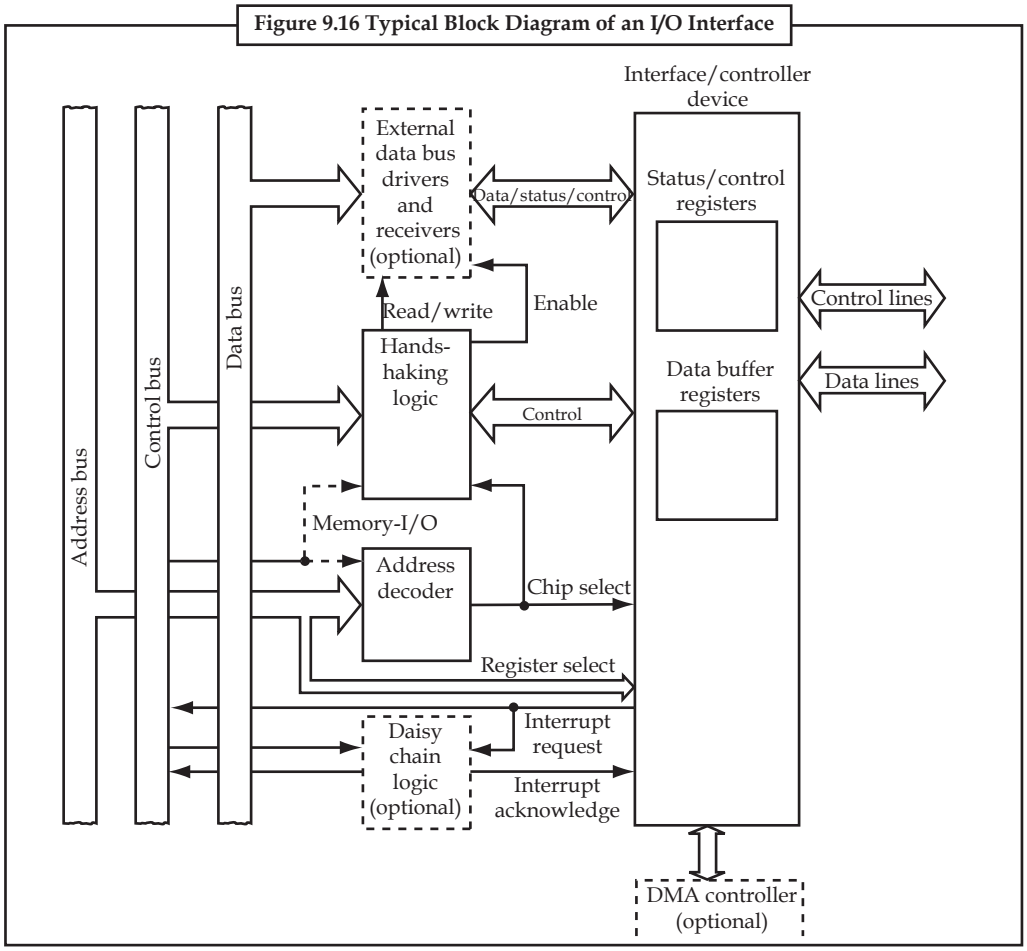


Figure 9.16 contains a block diagram of a typical I/O interface. The function of an I/O interface is essentially to translate the signals between the system bus and the I/O device and provide the buffers needed to satisfy the two sets of timing constraints. Most of the work done by the interface is accomplished by the large block on the right of the figure. Most often this block is implemented by a single IC device, but its functions could be scattered across several devices. Clearly, its functions vary radically, depending on the I/O device with which it is designed to communicate.

An interface can be divided into two parts, a part that interface to the I/O device and a part that interface to the system bus. Although little can be said about the I/O device side of the interface without knowing a lot about the device, the bus sides of all interfaces in a given system are very similar because they connect to the same bus. To support the main interface logic there must be data bus drivers and receivers, logic translating the interface control signals into the proper handshaking signals, and logic for decoding the address that appear on the bus. In an 8086/8088 system, 8286 transceivers could drive the data bus, just as they are used to drive the bus at its bus control logic end. However, the main interface devices may have built-in drivers and receivers that are sufficient for small, single board systems.

The handshaking logic cannot be designed until the control signals needed by the main interface device are known, and these signals may vary from one interface to the next. Typically this logic must accept read/write signals for determining the data direction and output the OE and T signals require by the 8286s. In a maximum mode 8086/8088 system it would receive the IOWC (or AIOWC) and IORC signals from the 8288 bus controller and in a minimum mode system it would receive the RD , WR and M/IO (or IO/M) signals. The interrupt request, ready, and reset

Notes

lines would also pass through this logic. In some cases, the bus control lines may pass through the handshaking logic unaltered (i.e. be connected directly to the main interface device).

The address decoder must receive the address and perhaps a bit indicating wheteher the address is in the I/O address space or the memory address space. In a minimum mode system this bit could be taken from the M/IO (or IO/M) line, but in a maximum mode system the memory-I/O determination is obtained directly from the IOWC and IORC lines. If the decoder determines that its interface is being referenced, then the decoder must send signals to the main device indicating that it has been selected and which register is being accessed. The bits designing the register may be the low-order address bits, but are often generated inside the interface device from the read/write control signal as well as the address signals.



Example: If there are two registers A and B that can be read from and two registers C and D that can be written into, then the read and write signals and bit 0 of the address bus could be used to specify the register as follows:

Write	Read	Address Bit 0	Register Being Accessed
0	1	0	A
0	1	1	B
1	0	0	C
1	0	1	D

If a daisy chain is included in the system instead of an interrupt priority management device, then each interface must contain daisy chain logic; and must include logic to generate the interrupt type. Also, the interface may be associated with a DMA controller.

Many interface are designed to detect at least two kinds of errors. Because the lines connecting an interface to its device are almost subject to noise, parity bits are normally appended to the information bytes as they are transmitted. If even parity is used the parity bit is set so that the total number of 1s, including the parity bit, is even. For odd parity the total number of 1s is odd. As these bytes are received the parity is checked and if it is in error, a certain status bit is set in a status register. Some interfaces are also designed to check error detection redundancy bytes that are placed after blocks of data. The other type of error most interfaces can detect is known as an overrun error. As we have seen when a computer inputs data it brings the data in form a data-in buffer register. If, for some reason, the contents of this register are replaced by new data before they are input by the computer, an overrun error occurs, such an error also happens when data are put in a data-out buffer before the current contents of the register have been output. As with parity errors, overrun errors cause a certain status bit to be set.

9.4 Functions of I/O Interface

An I/O interface is bridge between the processor and I/O devices. It controls the data exchange between the external devices and the main memory; or external devices and processor registers. Therefore, an I/O interface provides an interface internal to the computer which connects it to the processor and main memory and an interface external to the computer connecting it to external device or peripheral. The I/O interface should not only communicate the information from processor to main I/O device, but it should also coordinate these two. In addition, since there are speed differences between processor and I/O devices, the I/O interface should have facilities like buffer and error detection mechanism. Therefore, the major functions or requirements of an I/O interface are:

It should be able to provide control and timing signals

The need of I/O from various I/O devices by the processor is quite unpredictable. In fact it depends on I/O needs of particular programs and normally does not follow any pattern. Since, the

Notes

I/O interface also shares system bus and memory for data input/output, control and timing are needed to coordinate the flow of data from/to external devices to/from processor or memory.



Example: The control of the transfer of data from an external device to the processor might involve the following steps:

1. The processor enquires from the I/O interface to check the status of the attached device. The status can be busy, ready or out of order.
2. The I/O interface returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data by means of a command, which is a binary signal, to the I/O interface.
4. The I/O interface obtains a unit of data (e.g., 8 or 16 bits) from the external device.
5. The data is transferred from the I/O interface to the processor.

It should Communicate with the Processor

The above example clearly specifies the need of communication between the processor and I/O interface. This communication involves the following steps:

1. Commands such as READ SECTOR, WRITE SECTOR, SEEK track number and SCAN record-id sent over the control bus.
2. Data that are exchanged between the processor and I/O interface sent over the data bus.
3. Status: As peripherals are so slow, it is important to know the status of the I/O interface. The status signals are BUSY or READY or in an error condition from I/O interface.
4. Address recognition as each word of memory has an address, so does each I/O device. Thus an I/O interface must recognize one unique address for each peripheral it controls.

It should Communicate with the I/O Device

Communication between I/O interface and I/O device is needed to complete the I/O operation. This communication involves commands, status or data.

It should have a Provision for Data Buffering

Data buffering is quite useful for the purpose of smoothing out the gaps in speed of processor and the I/O devices. The data buffers are registers, which hold the I/O information temporarily. The I/O is performed in short bursts in which data are stored in buffer area while the device can take its own time to accept them. In I/O device to processor transfer, data are first transferred to the buffer and then passed on to the processor from these buffer registers. Thus, the I/O operation does not tie up the bus for slower I/O devices.

Error Detection Mechanism should be in-built

The error detection mechanism may involve checking the mechanical as well as data communication errors. These errors should be reported to the processor. The examples of the kind of mechanical errors that can occur in devices are paper jam in printer, mechanical failure, electrical failure etc. The data communication errors may be checked by using parity bit.

9.5 Kernel I/O Sub-system

Notes

I/O services provides by kernel are:

1. Scheduling
2. Buffering
3. Caching
4. Spooling
5. Device reservation
6. Error handling
7. I/O protection

I/O Scheduling

1. Some I/O request ordering via per-device queue
2. Some OSs try fairness – no one application receives poor service
3. To support asynchronous I/O, it must keep track of many I/O requests at the same time
   (a) Use device-status table

Buffering

Store data in memory while transferring between devices

1. To cope with device speed mismatch

   Example: File received via modem for storage on the hard disk via double buffering (next slide shows the differences in device speeds)

2. To cope with device transfer size mismatch
   (a) Common in computer networking
3. To maintain "copy semantics"
   (a) The version of data written to disk is guaranteed to be the version on the time of the application system call, independent of any subsequent changes in the application's buffer.

Caching

Fast memory holding copy of data

1. Always just a copy
2. Key to performance

Spooling

A buffer hat holds output for a device

1. If device can serve only one request at a time
2. Useful for printers and tape drives

Notes

Device Reservation

Coordination to provide exclusive access to a device

1. System calls for allocation and deallocation
2. Watch out for deadlock

Error Handling

1. OS can recover from disk read, device unavailable, transient write failures
2. Most OS's return an error number or code when I/O request fails
3. System error logs hold problem reports

I/O Protection

User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions

1. All I/O instructions defined to be privileged
2. I/O must be performed via system calls
3. Memory-mapped and I/O port memory locations must be protected too



Task For complete the I/O operation you need two I/O operation and these operations are.

9.6 Disk Scheduling

In order to satisfy an I/O request the disk controller must first move the head to the correct track and sector. Moving the head between cylinders takes a relatively long time so in order to maximise the number of I/O requests which can be satisfied the scheduling policy should try to minimise the movement of the head. On the other hand, minimising head movement by always satisfying the request of the closest location may mean that some requests have to wait a long time. Thus, there is a trade-off between throughput (the average number of requests satisfied in unit time) and response time (the average time between a request arriving and it being satisfied). Various different disk scheduling policies are used:

First Come First Served (FCFS)

The disk controller processes the I/O requests in the order in which they arrive, thus moving backwards and forwards across the surface of the disk to get to the next requested location each time. Since no reordering of request takes place the head may move almost randomly across the surface of the disk. This policy aims to minimise response time with little regard for throughput.

Each time an I/O request has been completed the disk controller selects the waiting request whose sector location is closest to the current position of the head. The movement across the surface of the disk is still apparently random but the time spent in movement is minimised. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it.

Notes

The drive head sweeps across the entire surface of the disk, visiting the outermost cylinders before changing direction and sweeping back to the innermost cylinders. It selects the next waiting requests whose location it will reach on its path backwards and forwards across the disk. Thus, the movement time should be less than FCFS but the policy is clearly fairer than SSTF.

Circular SCAN (C-SCAN)

C-SCAN is similar to SCAN but I/O requests are only satisfied when the drive head is traveling in one direction across the surface of the disk. The head sweeps from the innermost cylinder to the outermost cylinder satisfying the waiting requests in order of their locations. When it reaches the outermost cylinder it sweeps back to the innermost cylinder without satisfying any requests and then starts again.

Look

Similarly to SCAN, the drive sweeps across the surface of the disk, satisfying requests, in alternating directions. However the drive now makes use of the information it has about the locations requested by the waiting requests.



Example: A sweep out towards the outer edge of the disk will be reversed when there are no waiting requests for locations beyond the current cylinder.

Circular LOOK (C-LOOK)

Based on C-SCAN, C-LOOK involves the drive head sweeping across the disk satisfying requests in one direction only. As in LOOK the drive makes use of the location of waiting requests in order to determine how far to continue a sweep, and where to commence the next sweep. Thus it may curtail a sweep towards the outer edge when there are locations requested in cylinders beyond the current position, and commence its next sweep at a cylinder which is not the innermost one, if that is the most central one for which a sector is currently requested.

9.7 Disk Management

The hard disk is the secondary storage device that is used in the computer system. Usually the primary memory is used for the booting up of the computer. But a hard disk drive is necessary in the computer system since it needs to store the operating system that is used to store the information of the devices and the management of the user data.

The management of the IO devices that is the Input Output devices, like the printer and the other peripherals like the keyboard and the etc; all require the usage of the operating system. Hence the information of the all such devices and the management of the system are done by the operating system. The operating system works as an interpreter between the machine and the user.

The operating system is a must for the proper functioning of the computer. The computer is a device that needs to be fed with the instructions that are to be carried out and executed. Hence there needs to be an interpreter who is going to carry out the conversions from the high level language of the user to the low level language of the computer machine.



Task If your system not proper shut down what happen when you restart the system.

Notes

The hard disk drive as secondary memory is therefore needed for the purpose of installing the operating system. If there is no operating system then the question arises where to install the operating system. The operating system obviously cannot be installed in the primary memory however large that may be. The primary memory is also a volatile memory that cannot be used for the permanent storage of the system files of the operating system. The operating system requires the permanent file storage media like the hard disk.

Moreover the hard disk management is an important part of maintaining the computer, since it requires an efficient management of the data or the user information. The information regarding the Master Boot Record is stored in the hard disk drive. This is the information that is required during the start up of the computer. The computer system needs this information for loading the operating system.

The file management and the resources management is also a part of the hard disk management. The hard disk management requires an efficient knowledge of the operating system and its resources and the methods of how these resources can be employed in order to achieve maximum benefit. The operating system contains the resources and the tools that are used to manage the files in the operating system. The partitioning and the installation of the operating system itself may be considered as the hard disk management.

The hard disk management also involves the formatting of the hard disk drive and to check the integrity of the file system. The data redundancy check can also be carried out for the consistency of the hard disk drive. The hard disk drive management is also important in the case of the network where there are many hard disk drives to be managed.

Managing a single hard disk in a single user operating system is quite easy in comparison with the management of the hard disk drives in a multi user operating system where there is more than one user. It is not that much easy since the users are also required to be managed.

9.8 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 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 do 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.

9.8.1 Pseudo-Swap Space

When the system memory is used for swap space then it 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.

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.

When using pseudo swap for swap, the pages are locked; as the amount of pseudo-swap increases, the amount of lockable memory decreases. Pseudo-swap space is set to a maximum of three-quarters of system memory because the system can begin paging once three-quarters of

Notes

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.

9.8.2 Physical Swap Space

There are two kinds of physical swap space: device swap and file-system swap.

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.

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 must be made for each file-system block.

9.8.3 Three Rules of Swap Space Allocation

1. 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.
2. 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, swap requests are initially interleaved between the two swap devices at priority 0.
3. 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.



Task List various types of physical swap space.

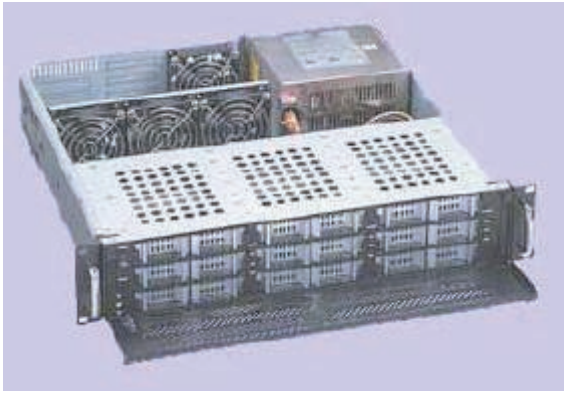
9.9 RAID Structure

RAID stands for Redundant Array of Independent (or Inexpensive) Disks. It involves the configuration (setting up) of two or more drives in combination for fault tolerance and performance. RAID disk drives are used frequently on servers and are increasingly being found in home and office personal computers.

Disks have high failure rates and hence there is the risk of loss of data and lots of downtime for restoring and disk replacement. To improve disk usage many techniques have been implemented. One such technology is RAID (Redundant Array of Inexpensive Disks). Its organisation is based on disk striping (or interleaving), which uses a group of disks as one storage unit. Disk striping

Notes

is a way of increasing the disk transfer rate up to a factor of N, by splitting files across N different disks. Instead of saving all the data from a given file on one disk, it is split across many. Since the N heads can now search independently, the speed of transfer is, in principle, increased manifold. Logical disk data/blocks can be written on two or more separate physical disks which can further transfer their sub-blocks in parallel. The total transfer rate system is directly proportional to the number of disks. The larger the number of physical disks striped together, the larger the total transfer rate of the system. Hence, the overall performance and disk accessing speed is also enhanced. The enhanced version of this scheme is mirroring or shadowing. In this RAID organisation a duplicate copy of each disk is kept. It is costly but a much faster and more reliable approach. The disadvantage with disk striping is that, if one of the N disks becomes damaged, then the data on all N disks is lost. Thus striping needs to be combined with a reliable form of backup in order to be successful.



Another RAID scheme uses some disk space for holding parity blocks. Suppose, three or more disks are used, then one of the disks will act as a parity block, which contains corresponding bit positions in all blocks. In case some error occurs or the disk develops a problems all its data bits can be reconstructed. This technique is known as disk striping with parity or block interleaved parity, which increases speed. But writing or updating any data on a disk requires corresponding recalculations and changes in parity block. To overcome this the parity blocks can be distributed over all disks.

RAID is a method of creating one or more pools of data storage space from several hard drives. It can offer fault tolerance and higher throughput levels than a single hard drive or group of independent hard drives. You can build a RAID configuration with IDE (parallel ATA), SATA (Serial ATA) or SCSI hard disks or, in fact, even drives like the old 3.5" floppy disk drive!

The exact meaning of RAID has been much debated and much argued. The use of "Redundant" is, in itself, a contentious point. That several manufacturers have deviated from accepted RAID terminology, created new levels of disk arrangements, called them RAID, and christened them with a number has not helped. There are even some single disk RAID configurations! Double parity, RAID 1.5, Matrix RAID etc., are examples of proprietary RAID configurations.

Data can be distributed across a RAID "array" using either hardware, software or a combination of the two. Hardware RAID is usually achieved either on-board on some server class motherboards or via an add-on card, using an ISA/PCI slot.

Basic RAID levels are the building blocks of RAID. Compound RAID levels are built by using:

JBOD: JBOD is NOT RAID. JBOD stands for 'Just a Bunch Of Disks'. This accurately describes the underlying physical structure that all RAID structures rely upon. When a hardware RAID controller is used, it normally defaults to JBOD configuration for attached disks. Some disk controller manufacturers incorrectly use the term JBOD to refer to a Concatenated array.

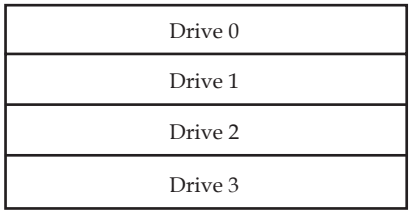
Notes

Concatenated Array: A Concatenated array is NOT RAID, although it is an array. It is a group of disks connected together, end-to-end, for the purpose of creating a larger logical disk. Although it is not RAID, it is included here as it is the result of early attempts to combine multiple disks into a single logical device. There is no redundancy with a Concatenated array. Any performance improvement over a single disk is achieved because the file-system uses multiple disks. This type of array is usually slower than a RAID-0 array of the same number of disks.

The good point of a Concatenated array is that different sized disks can be used in their entirety. The RAID arrays below require that the disks that make up the RAID array be the same size, or that the size of the smallest disk be used for all the disks.

The individual disks in a concatenated array are organized as follows:

Figure 9.17 Concatenated Segments



RAID-0: In RAID Level 0 (also called striping), each segment is written to a different disk, until all drives in the array have been written to.

The I/O performance of a RAID-0 array is significantly better than a single disk. This is true on small I/O requests, as several can be processed simultaneously, and for large requests, as multiple disk drives can become involved in the operation. Spindle-sync will improve the performance for large I/O requests.

This level of RAID is the only one with no redundancy. If one disk in the array fails, data is lost.

The individual segments in a 4-wide RAID-0 array are organized as follows:

Table 9.1: RAID-0 Segments

Drive 0	Drive 1	Drive 2	Drive 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

RAID-1: In RAID Level 1 (also called mirroring), each disk is an exact duplicate of all other disks in the array. When a write is performed, it is sent to all disks in the array. When a read is performed, it is only sent to one disk. This is the least space efficient of the RAID levels.

A RAID-1 array normally contains two disk drives. This will give adequate protection against drive failure. It is possible to use more drives in a RAID-1 array, but the overall reliability will not be significantly effected.

RAID-1 arrays with multiple mirrors are often used to improve performance in situations where the data on the disks is being read from multiple programs or threads at the same time. By being able to read from the multiple mirrors at the same time, the data throughput is increased, thus improving performance. The most common use of RAID-1 with multiple mirrors is to improve performance of databases.

Notes

Spindle-sync will improve the performance of writes. but have virtually no effect on reads. The read performance for RAID-1 will be no worse than the read performance for a single drive. If the RAID controller is intelligent enough to send read requests to alternate disk drives, RAID-1 can significantly improve read performance.

RAID-2: RAID Level 2 is an intellectual curiosity, and has never been widely used. It is more space efficient then RAID-1, but less space efficient then other RAID levels.

Instead of using a simple parity to validate the data (as in RAID-3, RAID-4 and RAID-5), it uses a much more complex algorithm, called a Hamming Code. A Hamming code is larger than a parity, so it takes up more disk space, but, with proper code design, is capable of recovering from multiple drives being lost. RAID-2 is the only simple RAID level that can retain data when multiple drives fail.

The primary problem with this RAID level is that the amount of CPU power required to generate the Hamming Code is much higher then is required to generate parity.

A RAID-2 array has all the penalties of a RAID-4 array, with an even larger write performance penalty. The reason for the larger write performance penalty is that it is not usually possible to update the Hamming Code. In general, all data blocks in the stripe modified by the write, must be read in, and used to generate new Hamming Code data.

Also, on large writes, the CPU time to generate the Hamming Code is much higher that to generate Parity, thus possibly slowing down even large writes.

The individual segments in a 4+2 RAID-2 array are organized as follows:

Table 9.2: RAID-2 Segments

Drive 0	Drive 1	Drive 2	Drive 3	Drive 4	Drive 5
0	1	2	3	Code	Code
4	5	6	7	Code	Code
8	9	10	11	Code	Code
12	13	14	15	Code	Code
16	17	18	19	Code	Code
20	21	22	23	Code	Code

RAID-3: RAID Level 3 is defined as bytewise (or bitwise) striping with parity. Every I/O to the array will access all drives in the array, regardless of the type of access (read/write) or the size of the I/O request.

During a write, RAID-3 stores a portion of each block on each data disk. It also computes the parity for the data, and writes it to the parity drive.

In some implementations, when the data is read back in, the parity is also read, and compared to a newly computed parity, to ensure that there were no errors.

RAID-3 provides a similar level of reliability to RAID-4 and RAID-5, but offers much greater I/O bandwidth on small requests. In addition, there is no performance impact when writing. Unfortunately, it is not possible to have multiple operations being performed on the array at the same time, due to the fact that all drives are involved in every operation.

As all drives are involved in every operation, the use of spindle-sync will significantly improve the performance of the array.

Because a logical block is broken up into several physical blocks, the block size on the disk drive would have to be smaller than the block size of the array. Usually, this causes the disk drive to need to be formatted with a block size smaller than 512 bytes, which decreases the storage capacity of the disk drive slightly, due to the larger number of block headers on the drive.

Notes

RAID-3 also has configuration limitations. The number of data drives in a RAID-3 configuration must be a power of two. The most common configurations have four or eight data drives.

Some disk controllers claim to implement RAID-3, but have a segment size. The concept of segment size is not compatible with RAID-3. If an implementation claims to be RAID-3, and has a segment size, then it is probably RAID-4.

RAID-4: RAID Level 4 is defined as blockwise striping with parity. The parity is always written to the same disk drive. This can create a great deal of contention for the parity drive during write operations.

For reads, and large writes, RAID-4 performance will be similar to a RAID-0 array containing an equal number of data disks.

For small writes, the performance will decrease considerably. To understand the cause for this, a one-block write will be used as an example.

1. A write request for one block is issued by a program.
2. The RAID software determines which disks contain the data, and parity, and which block they are in.
3. The disk controller reads the data block from disk.
4. The disk controller reads the corresponding parity block from disk.
5. The data block just read is XORed with the parity block just read.
6. The data block to be written is XORed with the parity block.
7. The data block and the updated parity block are both written to disk.

It can be seen from the above example that a one block write will result in two blocks being read from disk and two blocks being written to disk. If the data blocks to be read happen to be in a buffer in the RAID controller, the amount of data read from disk could drop to one, or even zero blocks, thus improving the write performance.

The individual segments in a 4+1 RAID-4 array are organized as follows:

Table 9.3: RAID-4 Segments

Drive 0	Drive 1	Drive 2	Drive 3	Drive 4
0	1	2	3	Parity
4	5	6	7	Parity
8	9	10	11	Parity
12	13	14	15	Parity
16	17	18	19	Parity
20	21	22	23	Parity

RAID-5: RAID Level 5 is defined as blockwise striping with parity. It differs from RAID-4, in that the parity data is not always written to the same disk drive.

RAID-5 has all the performance issues and benefits that RAID-4 has, except as follows:

Since there is no dedicated parity drive, there is no single point where contention will be created. This will speed up multiple small writes.

Multiple small reads are slightly faster. This is because data resides on all drives in the array. It is possible to get all drives involved in the read operation.

Notes

The individual segments in a 4+1 RAID-5 array are organized as follows:

Table 9.4: RAID-5 Segments

Drive 0	Drive 1	Drive 2	Drive 3	Drive 4
0	1	2	3	Parity
4	5	6	Parity	7
8	9	Parity	10	11
12	Parity	13	14	15
Parity	16	17	18	19
20	21	22	23	Parity

The above block layout is an example of Linux RAID-5 in left-asymmetric mode.

9.10 Summary

- Input is the signal or data received by the system and output is the signal or data sent from it.
- I/O devices are used by a person (or other system) to communicate with a computer.
- Keyboard is the one of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.
- Mouse is an input device that allows an individual to control a mouse pointer in a graphical user interface (GUI).
- Scanner is a hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object.
- A microphone is a hardware peripheral that allows computer users to input audio into their computers.
- Web Cam is a camera connected to a computer or server that allows anyone connected to the internet to view still pictures or motion video of a user.
- Digital camera is a type of camera that stores the pictures or video it takes in electronic format instead of to film.
- A computer joystick allows an individual to easily navigate an object in a game such as navigating a plane in a flight simulator.
- Monitor is a video display screen and the hard shell that holds it. It is also called a video display terminal (VDT).
- Printer is an external hardware device responsible for taking computer data and generating a hard copy of that data.
- Sound card is a sound card is an expansion card or integrated circuit that provides a computer with the ability to produce sound that can be heard by the user. It is also known as a sound board or an audio card.
- Speaker is a hardware device connected to a computer's sound card that outputs sounds generated by the card.

9.11 Keywords

Notes

Digital camera: A type of camera that stores the pictures or video it takes in electronic format instead of to film.

I/O devices: Hardware, which are used by a person (or other system) to communicate with a computer.

Input: It is the signal or data received by the system.

Joystick: A computer joystick allows an individual to easily navigate an object in a game such as navigating a plane in a flight simulator.

Keyboard: One of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.

Microphone: A microphone is a hardware peripheral that allows computer users to input audio into their computers.

Monitor: Also called a video display terminal (VDT) a monitor is a video display screen and the hard shell that holds it.

Mouse: An input device that allows an individual to control a mouse pointer in a graphical user interface (GUI).

Output: It is the signal or data sent from the system.

Printer: An external hardware device responsible for taking computer data and generating a hard copy of that data.

Scanner: Hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object.

Web Cam: A camera connected to a computer or server that allows anyone connected to the internet to view still pictures or motion video of a user.

9.12 Self Assessment

Fill in the blanks:

1. In computer architecture, the combination of the ..... and ..... is considered the heart of a computer.
2. I/O devices can be installed into ..... of the computer.
3. A hardware device that sends information into the CPU is known as ..... device.
4. Any peripheral ..... from a computer is known as output device.
5. A driver acts like a translator between the ..... and ..... .
6. A ..... is the smallest physical storage unit on the disk.
7. The size of a sector is ..... .
8. Most disks used in personal computers today rotate at a ..... .
9. The operating system works as ..... between the machine and the user.
10. ..... are often used to control video games, and usually have one or more push-buttons whose state can also be read by the computer.
11. The I/O and memory interface are the ..... to the bus control logic.
12. An ..... is bridge between the processor and I/O devices.

Notes

9.13 Review Questions

1. What is device driver? How it communicates with the devices?
2. What is device controller? How it differs from device driver?
3. What is memory-mapped I/O? Describe its role in the I/O system.
4. What is burst-block transfer?
5. Write a short note on computer terminal.
6. How a hard disk is physically composed? Describe it with suitable diagram.
7. What is the function of a system disk controller?
8. Write short notes on:
   (a) Pseudo swap space
   (b) Device swap space
   (c) File system swap space
9. What are the rules for swap space allocation?
10. Explain output unit in detail.

Answers: Self Assessment

1. CPU, main memory
2. physical slots
3. input
4. that receives and/or displays output
5. device, programs
6. sector
7. 512 bytes
8. a constant angular velocity
9. an interpreter
10. Joysticks
11. counterparts
12. I/O interface

9.14 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.

Notes



Online links www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 10: System Protection

CONTENTS

Objectives

Introduction

10.1 System Protection

10.2 Goals of Protection

10.3 Access Matrix and its Implementation

10.4 Access Control

10.5 Access Control Techniques

10.5.1 Passwords

10.5.2 Memory Card

10.5.3 Smart Card

10.5.4 Hand-held Password Generators

10.5.5 Biometrics

10.5.6 Encryption

10.5.7 Token

10.5.8 Encrypted Keys

10.6 Revocation of Access Rights

10.7 Capability-based System

10.8 Summary

10.9 Keywords

10.10 Self Assessment

10.11 Review Questions

10.12 Further Readings

Objectives

After studying this unit, you will be able to:

- Know system protection
- Describe various goals of protection
- Explain access matrix and its implementation
- Define access control
- Describe capability-based system

Introduction

The meaning of access control has changed over the last several years. Originally, access control usually referred to restricting physical access to a facility, building or room to authorized persons.

Notes

This used to be enforced mainly through a physical security guard. Then, with the advent of electronic devices, access control has evolved into the use of physical card access systems of a wide variety including biometric activated devices.

As computers evolved the meaning of access control began to change. Initially "access control lists" evolved specifying the user identities and the privileges granted to them in order to access a network operating system or an application.

Access control further evolved into the authentication, authorization and audit of a user for a session. Access control authentication devices evolved to include id and password, digital certificates, security tokens, smart cards and biometrics.

Access control authorization meanwhile evolved into Role Based Access Control (RBAC). This normally involves "mandatory access control". Mandatory access control is access control policies that are determined by the system and not the application or information owner.

RBAC is commonly found in government, military and other enterprises where the role definitions are well defined, the pace of change is not that fast and the supporting human resource environment is capable of keeping up with changes to an identity re their roles and privileges.

Access control is the process by which users are identified and granted certain privileges to information, systems, or resources. Understanding the basics of access control is fundamental to understanding how to manage proper disclosure of information.

10.1 System Protection

The use of computers to store and modify information can simplify the composition, editing, distribution, and reading of messages and documents. These benefits are not free, however, part of the cost is the aggravation of some of the security problems discussed above and the introduction of some new problems as well. Most of the difficulties arise because a computer and its programs are shared amongst several users.



Example: Consider a computer program that displays portions of a document on a terminal. The user of such a program is very likely not its developer. It is, in general, possible for the developer to have written the program so that it makes a copy of the displayed information accessible to himself (or a third party) without the permission or knowledge of the user who requested the execution of the program. If the developer is not authorised to view this information, security has been violated.

In compensation for the added complexities automation brings to security, an automated system, if properly constructed, can bestow a number of benefits as well.



Example: A computer system can place stricter limits on user discretion. In the paper system, the possessor of a document has complete discretion over its further distribution. An automated system that enforces distribution constraints strictly can prevent the recipient of a message or document from passing it to others. Of course, the recipient can always copy the information by hand or repeat it verbally, but the inability to pass it on directly is a significant barrier.

An automated system can also offer new kinds of access control. Permission to execute certain programs can be granted or denied so that specific operations can be restricted to designated users. Controls can be designed so that some users can execute a program but cannot read or modify it directly. Programs protected in this way might be allowed to access information not directly available to the user, filter it, and pass the results back to the user.

Notes

Information contained in an automated system must be protected from three kinds of threats:

1. Unauthorised disclosure of information,
2. Unauthorised modification of information and
3. Unauthorised withholding of information (usually called denial of service).

To protect the computer systems, you often need to apply some security models. Let us see in the next section about the various security models available.

10.2 Goals of Protection

The goals of protection are to ensure secrecy, privacy, authenticity and integrity of information. Table 10.1 provides descriptions of these goals.

Secrecy is a security concern because it is threatened by entities outside an operating system. An OS addresses it using the authentication service. Privacy is a protection concern. An OS address addresses privacy through the authorization service and the service and resource manager. The authorization service determines privileges of a user and the service and resource manager disallows request that exceed a user's privileges. It is up to the users to ensure privacy of their information using this set up. A user who wishes to share his programs and data with a few other users should set the authorization for his information accordingly. You call it controlled sharing of information. It is based on the need-to-know principle.

Table 10.1: Goals of Computer Security and Protection

Goals	Description
Secrecy	Only authorized users should be able to access information. This goal is also called confidentiality.
Privacy	Information should be used only for the purposes for which it is intended and shared.
Authenticity	It should be possible to verify the source or sender of information, and also verify that the information is preserved in the form in which it was created or sent.
Integrity	It should not be possible to destroy or corrupt information

10.3 Access Matrix and its Implementation

The access matrix model for computer protection is based on abstraction of operating system structures. Because of its simplicity and generality, it allows a variety of implementation techniques, as has been widely used.

There are three principal components in the access matrix model:

1. A set of passive objects,
2. A set of active subjects, which may manipulate the objects and
3. A set of rules governing the manipulation of objects by subjects.

Objects are typically files, terminals, devices, and other entities implemented by an operating system. A subject is a process and a domain (a set of constraints within which the process may access certain objects). It is important to note that every subject is also an object; thus it may be read or otherwise manipulated by another subject. The access matrix is a rectangular array with one row per subject and one column per object. The entry for a particular row and column reflects the mode of access between the corresponding subject and object. The mode of access allowed depends on the type of the object and on the functionality of the system; typical modes are read, write, append, and execute.

Notes

Table 10.2: An Access Matrix

Subject \ Objects	File 1	File 2	File 3
User 1	r, w	R	r, w, x
User 2	r	R	r, w, x
User 3	r, w, x	R, w	r, w, x

All accesses to objects by subjects are subject to some conditions laid down by an enforcement mechanism that refers to the data in the access matrix. This mechanism, called a reference monitor, rejects any accesses (including improper attempts to alter the access matrix data) that are not allowed by the current protection state and rules. References to objects of a given type must be validated by the monitor for that type.

While implementing the access matrix, it has been observed that the access matrix tends to be very sparse if it is implemented as a two-dimensional array. Consequently, implementations that maintain protection of data tend to store them either row wise, keeping with each subject a list of the objects and access modes allowed on it; or column wise, storing with each object a list of those subjects that may access it and the access modes allowed on each. The former approach is called the capability list approach and the latter is called the access control list approach.

In general, access control governs each user's ability to read, execute, change, or delete information associated with a particular computer resource. In effect, access control works at two levels: first, to grant or deny the ability to interact with a resource, and second, to control what kinds of operations or activities may be performed on that resource. Such controls are managed by an access control system. Today, there are numerous methods of access controls implemented or practiced in real-world settings.



Task Previously we call capability list approach now we call that approach.

Mandatory Access Control

In a Mandatory Access Control (MAC) environment, all requests for access to resources are automatically subject to access controls. In such environments, all users and resources are classified and receive one or more security labels (such as "Unclassified," "Secret," and "Top Secret"). When a user requests a resource, the associated security labels are examined and access is permitted only if the user's label is greater than or equal to that of the resource.

Discretionary Access Control

In a Discretionary Access Control (DAC) environment, resource owners and administrators jointly control access to resources. This model allows for much greater flexibility and drastically reduces the administrative burdens of security implementation.

Rule-based Access Control

In general, rule-based access control systems associate explicit access controls with specific system resources, such as files or printers. In such environments, administrators typically establish access rules on a per-resource basis, and the underlying operating system or directory services employ those rules to grant or deny access to users who request access to such resources. Rule-based access controls may use a MAC or DAC scheme, depending on the management role of resource owners.

Notes

Role-based Access Control

Role-based access control (RBAC) enforces access controls depending upon a user's role(s). Roles represent specific organisational duties and are commonly mapped to job titles such as "Administrator," "Developer," or "System Analyst." Obviously, these roles require vastly different network access privileges.

Role definitions and associated access rights must be based upon a thorough understanding of an organisation's security policy. In fact, roles and the access rights that go with them should be directly related to elements of the security policy.

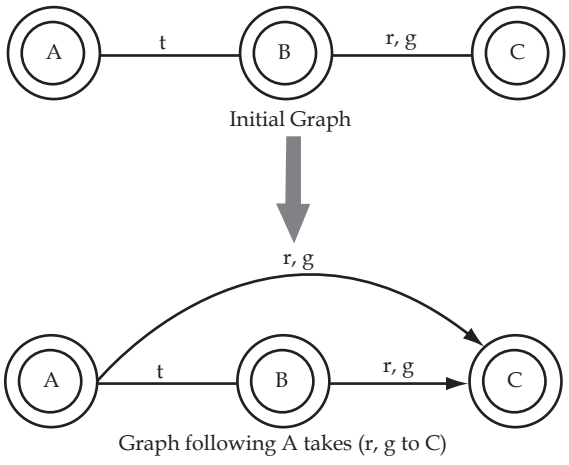
Take-grant Model

The access matrix model, properly interpreted, corresponds very well to a wide variety of actual computer system implementations. The simplicity of the model, its definition of subjects, objects, and access control mechanisms, is very appealing. Consequently, it has served as the basis for a number of other models and development efforts. You now discuss a model based on access matrix.

Take-grant models use graphs to model access control. They have been well researched. Although explained in the terms of graph theory, these models are fundamentally access matrix models. The graph structure can be represented as an adjacency matrix, and labels on the arcs can be coded as different values in the matrix.

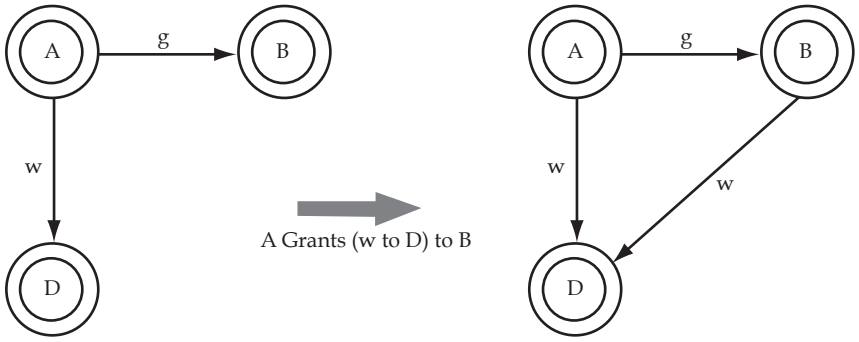
In a take-grant model, the protection state of a system is described by a directed graph that represents the same information found in an access matrix. Nodes in the graph are of two types, one corresponding to subjects and the other to objects. An arc directed from a node A to another node B indicates that the subject (or object) A has some access right(s) to subject (or object) B. The arc is labeled with the set of A's rights to B. The possible access rights are read (r), write (w), take (t), and grant (g). Read and write have the obvious meanings. "Take" access implies that node A can take node B's access rights to any other node.

Example: If there is an arc labeled (r, g) from node B to node C, and if the arc from A to B includes a "t" in its label, then an arc from A to C labeled (r, g) could be added to the graph. Conversely, if the arc from A to B is marked with a "g", B can be granted any access right A possesses. Thus, if A has (w) access to a node D and (g) access to B, an arc from B to D marked (w) can be added to the graph.



Take

Notes



Grant

Because the graph needs only the inclusion of arcs corresponding to non-null entries in the access matrix, it provides a compact way to present the same information given in a relatively sparse access matrix. Capability systems are thus prime candidates for this modeling technique; each arc would then represent a particular capability. Together with the protection graph, the model includes a set of rules for adding and deleting both nodes and arcs to the graph.

Two of these, corresponding to the exercise of "take" and "grant" access rights, have already been described. A "create" rule allows a new node to be added to the graph. If subject A creates a new node Y, both the node Y and an arc AY are added to the graph. The label on AY includes any subset of the possible access rights. A "remove" rule allows an access right to be removed from an arc; if all rights are removed from an arc, the arc is removed as well.

Task Differentiate between DAC and RBAC concept.

10.4 Access Control

Access control is the ability to permit or deny the use of a particular resource by a particular entity. Access control mechanisms can be used in managing physical resources (such as a movie theater, to which only ticketholders should be admitted), logical resources (a bank account, with a limited number of people authorized to make a withdrawal), or digital resources (for example, a private text document on a computer, which only certain users should be able to read).

Today, in the age of digitization, there is a convergence between physical access control and computer access control. Modern access control (more commonly referred to in the industry as "identity management systems") now provide an integrated set of tools to manage what a user can access physically, electronically and virtually as well as providing an audit trail for the lifetime of the user and their interactions with the enterprise.

Modern access control systems rely upon:

1. Integrated enterprise user and identity databases and Lightweight Directory Access Protocol (LDAP) directories.
2. Strong business processes pertaining to the provisioning and de-provisioning of a user.
3. Provisioning software integrated with the business provisioning and de-provisioning process.
4. Site, building and room based access control systems that are LDAP enabled or, able to be integrated into a virtual enterprise LDAP directory.

Notes

5. A global enterprise id for each user to integrate the user's identity between many applications and systems.
6. A strong end to end audit of everywhere the physical person went as well as the systems, application and information systems they accessed.

With many portions of an enterprise now outsourced, the challenges to access control have increased. Today it is becoming common to have contractual agreements with the enterprise's outsource partners that:

1. Automatically provision and de-provision users
2. Build trusted authentication and authorization mechanisms
3. Provide end to end user session audit
4. Integrate with the remote user's physical access e.g. to a call center operating on the enterprise's behalf.

Controlling how network resources are accessed is paramount to protecting private and confidential information from unauthorized users. The types of access control mechanisms available for information technology initiatives today continues to increase at a breakneck pace.

Most access control methodologies are based on the same underlying principles. If you understand the underlying concepts and principles, you can apply this understanding to new products and technologies and shorten the learning curve so you can keep pace with new technology initiatives.

Access control devices properly identify people, and verify their identity through an authentication process so they can be held accountable for their actions. Good access control systems record and timestamp all communications and transactions so that access to systems and information can be audited at later dates.

Reputable access control systems all provide authentication, authorization, and administration. Authentication is a process in which users are challenged for identity credentials so that it is possible to verify that they are who they say they are.

Once a user has been authenticated, authorization determines what resources a user is allowed to access. A user can be authenticated to a network domain, but only be authorized to access one system or file within that domain. Administration refers to the ability to add, delete, and modify user accounts and user account privileges.

10.5 Access Control Techniques

There are different types of access control technologies that can all be used to solve enterprise access solutions. Tokens, smart cards, encrypted keys, and passwords are some of the more popular access control technologies.

10.5.1 Passwords

Passwords are used for access control more than any other type of solution because they are easy to implement and are extremely versatile. On information technology systems, passwords can be used to write-protect documents, files, directories, and to allow access to systems and resources. The downside to using passwords is that they are among the weakest of the access control technologies that can be implemented.

The security of a password scheme is dependent upon the ability to keep passwords secret. Therefore, a discussion of increasing password security should begin with the task of choosing a password. A password should be chosen such that it is easy to remember, yet difficult to guess.

Notes

There are a few approaches to guessing passwords which I shall discuss, along with methods of countering these attacks.

Most operating systems, as well as large applications such as Database Management Systems, are shipped with administrative accounts that have preset passwords. Because these passwords are standard, outside attackers have used them to break into IT systems. It is a simple, but important, measure to change the passwords on administrative accounts as soon as an IT system is received.

A second approach to discovering passwords is to guess them, based on information about the individual who created the password. Using such information as the name of the individual, spouse, pet or street address or other information such as a birth date or birthplace can frequently yield an individual's password. Users should be cautioned against using information that is easily associated with them for a password.

There are several brute force attacks on passwords that involve either the use of an on-line dictionary or an exhaustive attempt at different character combinations. There are several tactics that may be used to prevent a dictionary attack.

They include deliberately misspelling words, combining two or more words together, or including numbers and punctuation in a password. Ensuring that passwords meet a minimum length requirement also helps make them less susceptible to brute force attacks.

To assist users in choosing passwords that are unlikely to be guessed, some operating systems provide randomly generated passwords. While these passwords are often described as pronounceable, they are frequently difficult to remember, especially if a user has more than one of them, and so are prone to being written down. In general, it is better for users to choose their own passwords, but with the considerations outlined above in mind.

Password length and the frequency with which passwords are changed in an organization should be defined by the organization's security policy and procedures and implemented by the organization's IT system administrator(s).

The frequency with which passwords should be changed should depend on the sensitivity of the data. Periodic changing of passwords can prevent the damage done by stolen passwords, and make "brute force" attempts to break into system more difficult.

Too frequent changes, however, can be irritating to users and can lead to security breaches such as users writing down passwords or using too obvious passwords in an attempt to keep track of a large number of changing passwords. This is inevitable when users have access to a large number of machines. Security policy and procedures should strive for consistent, livable rules across an organization.

Some mainframe operating systems and many PC applications use passwords as a means of access control, not just authentication. Instead of using mechanisms such as Access Control Lists (ACLs), access is granted by entering a password. The result is a proliferation of passwords that can significantly reduce the overall security of an IT system. While the use of passwords as a means of access control is common, it is an approach that is less than optimal and not cost-effective.

There are numerous password-cracking utilities out on the Internet — some of which are freeware and some of which are licensed professional products. If a hacker downloads an encrypted password file, or a write-protected document with password protection, they can run the password file or document through a password cracking utility, obtain the password, and then either enter the system using a legitimate user's account or modify the write-protected document by inserting the correct password when prompted. By using a protocol analyzer, hackers can "sniff" the network traffic on the wire and obtain passwords in plaintext rather easily.

However, in spite of the risks in using passwords, they are still commonly used world over with the assumption that taking the trouble to violate password protections would not be worth the

Notes

time and effort. If passwords are used, it is recommended that mixed-case passwords with both numeric and alphabet characters are used, since these types of passwords are more difficult for password cracking tools to crack. Passwords with names and real words in them are easiest to crack. Good password choices look like this:

1bHkL0m8

a9T4j7uU

7VbbsT10

gL4lJT3m

koO521qW

Poor password choices look like this:

Billsmith

Troutfishing

Jessica

NewYorkOffice

Surfdude

While stronger access control systems are clearly available, password models are not going to go away anytime soon. Some organizations routinely run password crackers on end-user accounts to check if end-users are using easy to guess passwords, or more secure password choices. As long as passwords are being used, they should be managed through routine audits, and expired according to a pre-determined schedule.

10.5.2 Memory Card

There is a very wide variety of memory card systems with applications for user identification and authentication. Such systems authenticate a user's identity based on a unique card, i.e., something the user possesses, sometimes in conjunction with a PIN (Personal Identification Number), i.e., something a user knows.

The use of a physical object or token, in this case a card, has prompted memory card systems to be referred to as token systems. Other examples of token systems are optical storage cards and Integrated Circuit (IC) keys.

Memory cards store, but do not process, information. Special reader/writer devices control the writing and reading of data to and from the cards. The most common type of memory card is a magnetic stripe card.

These cards use a film of magnetic material, similar or identical to audio and computer magnetic tape and disk equipment, in which a thin strip, or stripe, of magnetic material affixed to the surface of a card. A magnetic stripe card is inexpensive, easy to produce and has a high storage capacity.

The most common forms of a memory card are the telephone calling card, credit card, and ATM card. The number on a telephone calling card serves as both identification and authentication for the user of a long distance carrier and so must remain secret.

The card can be used directly in phones that read cards or the number may be entered manually in a touch tone phone or verbally to an operator. Possession of the card or knowledge of the number is sufficient to authenticate the user.

Possession of a credit card, specifically the card holder's name, card number and expiration date, is sufficient for both identification and authentication for purchases made over the telephone.

Notes

The inclusion of a signature and occasionally a photograph provide additional security when the card is used for purchases made in person.

The ATM card employs a more sophisticated use of a memory card, involving not only something the user possesses, namely the card, but also something the user knows, viz. the PIN. A lost or stolen card is not sufficient to gain access; the PIN is required as well. This paradigm of use seems best suited to IT authentication applications.

While there are some sophisticated technical attacks that can be made against memory cards, they can provide a marked increase in security over password only systems. It is important that users be cautioned against writing their PIN on the card itself or there will be no increase in security over a simple password system.

Memory cards can and are widely used to perform authentication of users in a variety of circumstances from banking to physical access. It is important that the considerations mentioned above for password selection are followed for PIN selection and that the PIN is never carried with the card to gain the most from this hybrid authentication system.

10.5.3 Smart Card

A smart card is a device typically the size and shape of a credit card and contains one or more integrated chips that perform the functions of a computer with a microprocessor, memory, and input/output. Smart cards may be used to provide increased functionality as well as an increased level of security over memory cards when used for identification and authentication.

Smart Cards are plastic cards that have integrated circuits or storage receptacles embedded in them. Smart cards with integrated circuits that can execute transactions and are often referred to as "active" smart cards.

Cards with memory receptacles that simply store information (such as your bank ATM card) are referred to as "passive." Whether or not a memory card is a type of smart card depends on who you ask and what marketing material you are reading. Used to authenticate users to domains, systems, and networks, smart cards offer two-factor authentication – something a user has, and something a user knows. The card is what the user has, and the Personal Identification Number (PIN) is what the person knows.

A smart card can process, as well as store, data through its microprocessor; therefore, the smart card itself (as opposed to the reader/writer device), can control access to the information stored on the card. This can be especially useful for applications such as user authentication in which security of the information must be maintained. The smart card can actually perform the password or PIN comparisons inside the card.

As an authentication method, the smart card is something the user possesses. With recent advances, a password or PIN (something a user knows) can be added for additional security and a fingerprint or photo (something the user is) for even further security. As contrasted with memory cards, an important and useful feature of a smart card is that it can be manufactured to ensure the security of its own memory, thus reducing the risk of lost or stolen cards.

The smart card can replace conventional password security with something better, a PIN, which is verified by the card versus the computer system, which may not have as sophisticated a means for user identification and authentication.

The card can be programmed to limit the number of login attempts as well as ask biographic questions, or make a biometric check to ensure that only the smart card's owner can use it. In addition, non-repeating challenges can be used to foil a scenario in which an attacker tries to login using a password or PIN he observed from a previous login. In addition, the complexity of smart card manufacturing makes forgery of the card's contents virtually impossible.

Notes

Use of smart devices means the added expense of the card itself, as well as the special reader devices. Careful decisions as to what systems warrant the use of a smart card must be made. The cost of manufacturing smart cards is higher than that of memory cards but the disparity will get less and less as more and more manufacturers switch to this technology. On the other hand, it should be remembered that smart cards, as opposed to memory only cards, can effectively communicate with relatively 'dumb', inexpensive reader devices.

The proper management and administration of smart cards will be a more difficult task than with typical password administration. It is extremely important that responsibilities and procedures for smart card administration be carefully implemented. Smart card issuance can be easily achieved in a distributed fashion, which is well suited to a large organizational environment. However, just as with password systems, care should be taken to implement consistent procedures across all involved systems.

10.5.4 Hand-held Password Generators

Hand-held password generators are a state-of-the-art type of smart token. They provide a hybrid authentication, using both something a user possesses (i.e., the device itself) and something a user knows (e.g., a 4 to 8 digit PIN). The device is the size of a shirt-pocket calculator, and does not require a special reader/writer device. One of the main forms of password generators is a challenge-response calculator.

When using a challenge-response calculator, a user first types his user name into the IT system. The system then presents a random challenge, for example, in the form of a 7-digit number. The user is required to type his PIN into the calculator and then enter the challenge generated by the IT system into the calculator. The generator then provides a corresponding response, which he then types into the IT system. If the response is valid, the login is permitted and the user is granted access to the system.

When a password generator is used for access to a computer system in place of the traditional user name and password combination, an extra level of security is gained. With the challenge response calculator, each user is given a device that has been uniquely keyed; he cannot use someone else's device for access. The host system must have a process or a processor to generate a challenge response pair for each login attempt, based on the initially supplied user name.

Each challenge is different, so observing a successful challenge-response exchange gives no information for a subsequent login. Of course, with this system the user must memorize a PIN.

The hand-held password generator can be a low-cost addition to security, but the process is slightly complicated for the user. He must type two separate entries into the calculator, and then correctly read the response and type it into the computer. This process increases the chance for making a mistake.

Overall, this technology can be a useful addition to security, but users may find some inconvenience. Management, if they decide to use this approach, will have to establish a plan for integrating the technology into their IT systems. There will also be the administrative challenge for keying and issuing the cards, and keeping the user database up-to-date.

10.5.5 Biometrics

Biometric devices authenticate users to access control systems through some sort of personal identifier such as a fingerprint, voiceprint, iris scan, retina scan, facial scan, or signature dynamics. The nice thing about using biometrics is that end-users do not lose or misplace their personal identifier. It's hard to leave your fingers at home. However, biometrics have not caught on as fast as originally anticipated due to the false positives and false negatives that are common when using biometric technologies.

Notes

Biometric authentication systems employ unique physical characteristics (or attributes) of an individual person in order to authenticate the person's identity. Physical attributes employed in biometric authentication systems include fingerprints, hand geometry, hand -written signatures, retina patterns and voice patterns. Biometric authentication systems based upon these physical attributes have been developed for computer login applications.

Biometric authentication systems generally operate in the following manner:

Prior to any authentication attempts, a user is "enrolled" by creating a reference profile (or template) based on the desired physical attribute. The reference profile is usually based on the combination of several measurements. The resulting template is associated with the identity of the user and stored for later use.

When attempting to authenticate themselves, the user enters his login name or, alternatively, the user may provide a card/token containing identification information. The user's physical attribute is then measured.

The previously stored reference profile of the physical attribute is then compared with the measured profile of the attribute taken from the user. The result of the comparison is then used to either accept or reject the user.

Biometric systems can provide an increased level of security for IT systems, but the technology is still less matures than memory or smart cards. Imperfections in biometric authentication devices arise from technical difficulties in measuring and profiling physical attributes as well as from the somewhat variable nature of physical attributes. Many physical attributes change depending on various conditions.



Example: A person's speech pattern may change under stressful conditions or when suffering from a sore throat or cold.

Biometric systems are typically used in conjunction with other authentication means in environments requiring high security.

10.5.6 Encryption

Encryption is a process of coding information which could either be a file or mail message in into cipher text a form unreadable without a decoding key in order to prevent anyone except the intended recipient from reading that data. Decryption is the reverse process of converting encoded data to its original un-encoded form, plaintext.

A key in cryptography is a long sequence of bits used by encryption/decryption algorithms.

The following represents a hypothetical 40-bit key:

00001010 01101001 10011110 00011100 01010101

A given encryption algorithm takes the original message, and a key, and alters the original message mathematically based on the key's bits to create a new encrypted message. Likewise, a decryption algorithm takes an encrypted message and restores it to its original form using one or more keys.

When a user encodes a file, another user cannot decode and read the file without the decryption key. Adding a digital signature, a form of personal authentication, ensures the integrity of the original message.

To encode plaintext, an encryption key is used to impose an encryption algorithm onto the data. To decode cipher, a user must possess the appropriate decryption key. A decryption key consists of a random string of numbers, from 40 through 2,000 bits in length. The key imposes a decryption algorithm onto the data. This decryption algorithm reverses the encryption algorithm, returning

Notes

the data to plaintext. The longer the encryption key is, the more difficult it is to decode. For a 40-bit encryption key, over one trillion possible decryption keys exist.

There are two primary approaches to encryption: symmetric and public-key. Symmetric encryption is the most common type of encryption and uses the same key for encoding and decoding data. This key is known as a session key. Public-key encryption uses two different keys, a public key and a private key. One key encodes the message and the other decodes it. The public key is widely distributed while the private key is secret.

Aside from key length and encryption approach, other factors and variables impact the success of a cryptographic system.



Example: Different cipher modes, in coordination with initialization vectors and salt values, can be used to modify the encryption method. Cipher modes define the method in which data is encrypted. The stream cipher mode encodes data one bit at a time. The block cipher mode encodes data one block at a time. Although block cipher tends to execute more slowly than stream cipher.

10.5.7 Token

A token is a handheld device that has a built-in challenge response scheme that authenticates with an enterprise server. Today's leading tokens typically use time-based challenge and response algorithms that constantly change and expire after a certain length of time, e.g., one minute. Like smart cards, tokens use two-factor authentication. However, unlike smart cards, the two-factor authentication is constantly changing based on timed intervals – therefore, when a password is entered, it cannot be reused, even if someone sniffing the wire detected it in transit.

10.5.8 Encrypted Keys

Encrypted keys are mathematical algorithms that are used to secure confidential information and verify the authenticity of the people sending and receiving the information. Standards for encrypted keys have been created to make sure that security requirements are taken into account, and to allow technologies made by different vendors to work together. The most widely used standard for encrypted keys is called X.509 digital certificates. Using digital certificates allows you to stipulate who can access and view the information you are encrypting with the key.



Task Explain various techniques of access controls.

10.6 Revocation of Access Rights

Revocation of access rights to objects in shared environment is possible.

Following parameter are consider for revocation of access rights.

1. Immediate and delayed
2. Selective and general
3. Partial and total
4. Temporary and permanent

Revocation is easy for access list and complex for capabilities list. The access list searched for the access right to be revoked and they are detected from the list. Revocation may be immediate general. It may be selective, total or partial.

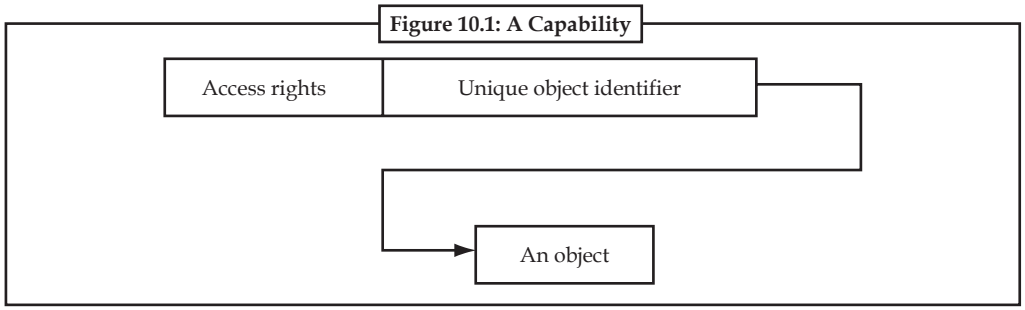
10.7 Capability-based System

Notes

Capability-based systems differ significantly from conventional computer systems. Capabilities provide (1) a single mechanism to address both primary and secondary memory, and (2) a single mechanism to address both hardware and software resources. While solving many difficult problems in complex system design, capability systems introduce new challenges of their own.

Conceptually, a capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system. A capability is implemented as a data structure that contains two items of information: a unique object identifier and access rights, as shown in Figure 10.1.

Figure 10.1: A Capability



The identifier addresses or names a single object in the computer system. An object, in this context, can be any logical or physical entity, such as a segment of memory, an array, a file, a line printer, or a message port. The access rights define the operations that can be performed on that object.

Example: The access rights can permit read-only access to a memory segment or send-and-receive access to a message port.

Each user, program, or procedure in a capability system has access to a list of capabilities. These capabilities identify all of the objects which that user, program, or procedure is permitted to access. To specify an object, the user provides the index of a capability in the list.

Example: To output a record to a file, the user might call the file system as follows:

PUT(file-capability , "this is a record");

The capability specified in the call serves two purposes. First, it identifies the file to be written. Second, it indicates whether the operation to be performed (PUT in this case) is permitted.

A capability thus provides addressing and access rights to an object. Capabilities are the basis for object protection; a program cannot access an object unless its capability list contains a suitably privileged capability for the object. Therefore, the system must prohibit a program from directly modifying the bits in a capability. If a program could modify the bits in a capability, it could forge access to any object in the system by changing the identifier and access rights fields.

Capability system integrity is usually maintained by prohibiting direct program modification of the capability list. The capability list is modified only by the operating system or the hardware. However, programs can obtain new capabilities by executing operating system or hardware operations.

Example: When a program calls an operating system routine to create a new file, the operating system stores a capability for that file in the program's capability list. A capability system also provides other capability operations. Examples include operations to:

1. Move capabilities to different locations in a capability list.

Notes

2. Delete a capability.
3. Restrict the rights in a capability, producing a less-privileged version,
4. Pass a capability as a parameter to a procedure.
5. Transmit a capability to another user in the system.

Thus, a program can execute direct control over the movement of capabilities and can share capabilities, and therefore, objects, with other programs and users.

It is possible for a user to have several capability lists. One list will generally be the master capability list containing capabilities for secondary lists, and so on. This structure is similar to a multi-level directory system, but, while directories address only files, capabilities address objects of many types.

Manage the access control matrix is to store it by rows. These are called capabilities. In the example in Table 10.3, Bob's capabilities would be as shown in Table 10.4.

Table 10.3: Naive Access Control Matrix

	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	rw	r
Alice	x	x	rw	–
Bob	rx	r	r	r

The strengths and weaknesses of capabilities are more or less the opposite of ACLs. Runtime security checking is more efficient, and you can do delegation without much difficulty: Bob could create a certificate saying "Here is my capability, and I hereby delegate to David the right to read file 4 from 9 A.M. to 1 P.M.; signed Bob." On the other hand, changing a file's status can suddenly become more tricky, as it can be difficult to find out which users have access. This can be tiresome when investigating an incident or preparing evidence of a crime.

Table 10.4: A Capability

User	Operating System	Accounts Program	Accounting Data	Audit Trail
Bob	rx	r	r	r

There were a number of experimental implementations in the 1970s, which were rather like file passwords; users would get hard-to-guess bit strings for the various read, write, and other capabilities to which they were entitled. It was found that such an arrangement could give very comprehensive protection. It was not untypical to find that almost all of an operating system could run in user mode, rather than as supervisor, so operating system bugs were not security critical. (In fact, many operating system bugs caused security violations, which made debugging the operating system much easier.)

The IBM AS/400 series systems employed capability-based protection, and enjoyed some commercial success. Now capabilities are making a comeback in the form of public key certificates. For now, think of a public key certificate as a credential signed by some authority, which declares that the holder of a certain cryptographic key is a certain person, a member of some group, or the holder of some privilege.

As an example of where certificate-based capabilities can be useful, consider a hospital. If you implemented a rule stating "a nurse will have access to all the patients who are on her ward, or who have been there in the last 90 days," naively, each access control decision in the patient record system would require several references to administrative systems, to find out which nurses and which patients were on which ward, when. This means that a failure of the administrative systems can now affect patient safety much more directly than was previously the case, which is

Notes

a clearly bad thing. Matters can be much simplified by giving nurses certificates that entitle them to access the files associated with their current ward. Such a system is starting to be fielded at our university hospital.

One point to bear in mind is that as public key certificates are often considered to be "crypto" rather than "access control," their implications for access control policies and architectures are not always thought through. The unit that could have been learned from the capability systems of the 1970s are generally having to be rediscovered (the hard way). In general, the boundary between crypto and access control is a fault line where things can easily go wrong. The experts often come from different backgrounds, and the products from different suppliers.

10.8 Summary

- Access control mechanisms operate at a number of levels in a system, from applications down through the operating system to the hardware.
- Higher-level mechanisms can be more expressive, but also tend to be more vulnerable to attack, for a variety of reasons ranging from intrinsic complexity to implementer skill levels.
- Most attacks involve the opportunistic exploitation of bugs; and software that is very large, very widely used, or both (as with operating systems) is particularly likely to have security bugs found and publicized.
- Operating systems are also vulnerable to environmental changes that undermine the assumptions used in their design.

10.9 Keywords

Access control: It is the process by which users are identified and granted certain privileges to information, systems, or resources.

Access control device: It properly identifies people, and verifies their identity through an authentication process so they can be held accountable for their actions.

Authentication: It is a process by which you verify that someone is who they claim they are.

Authorization: It is finding out if the person, once identified, is permitted to have the resource.

Smart card: It is a device typically the size and shape of a credit card and contains one or more integrated chips that perform the functions of a computer with a microprocessor, memory, and input/output.

10.10 Self Assessment

State whether the following statements are true or false:

1. An automated system can also offer new kinds of access control.
2. Information should not be used only for the purposes for which it is intended and shared.
3. The access matrix model for computer protection is based on abstraction of operating system structures.
4. Role-based access control enforces access controls does not depends upon a user's role(s).
5. Take-grant models use graphs to model access control.

Notes

Fill in the blanks:

6. ..... is the ability to permit or deny the use of a particular resource by a particular entity.
7. Modern access control more commonly referred to in the industry as ..... .
8. LDAP stands for ..... .
9. ..... are plastic cards that have integrated circuits or storage receptacles embedded in them.
10. ..... is a process of coding information which could either be a file or mail message in into cipher text a form unreadable without a decoding key in order to prevent anyone except the intended recipient from reading that data.

10.11 Review Questions

1. What do you mean by system protection?
2. Describe various goals of security and protection.
3. Explain access matrix. Also describe its implementation.
4. What do you mean by access control?
5. Describe various techniques of access controls.
6. Write short note on "Biometrics".
7. Explain encryption concept.
8. What do you mean by smart card?
9. Describe capability based system.
10. Describe the concept of "hand-held password generators".

Answers: Self Assessment

1. True
2. False
3. True
4. False
5. True
6. Access control
7. identity management systems
8. Lightweight Directory Access Protocol
9. Smart Cards
10. Encryption

10.12 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Notes

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 11: System Security

CONTENTS

Objectives

Introduction

11.1 System Security

11.2 Security Problem

11.3 Program Threats

11.4 System and Network Threats

11.5 Cryptography as a Security Tools

11.5.1 Hashing

11.5.2 Pretty Good Privacy (PGP)

11.6 User Authentication

11.7 Implementing Security Defenses

11.8 Types of Intrusion Prevention System

11.9 Implementation Challenges

11.10 Firewall to Protect Systems and Networks

11.11 Summary

11.12 Keywords

11.13 Self Assessment

11.14 Review Questions

11.15 Further Readings

Objectives

After studying this unit, you will be able to:

- Define system security
- Explain system and network threats
- Know cryptography as a security tools
- Describe user authentication
- Explain various types of intrusion prevention system

Introduction

Computer security is traditionally defined by the three attributes of confidentiality, integrity, and availability. Confidentiality is the prevention of unauthorised disclosure of information. Integrity is the prevention of unauthorised modification of information, and availability is the prevention of unauthorised withholding of information or resources. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

Notes

Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user.

Even if a perfectly secure operating system was created, human error (or purposeful human malice) can make it insecure. More importantly, there is no such thing as a completely secure system. No matter how secure the experts might think a particular system is, there exists someone who is clever enough to bypass the security.

It is important to understand that a trade-off always exists between security and the ease of use. It is possible to be too secure but security always extracts a price, typically making it more difficult to use your systems for their intended purposes. Users of a secure system need to continually type in, continually change and memorise complex passwords for every computer operation, or use biometric systems with retina and fingerprint and voice scans. All these measures along with confirmations are likely to bring any useful work to a snail's pace. If the consequences are great enough, then you might have to accept extreme security measures. Security is a relative concept, and gains in security often come only with penalties in performance. To date, most systems designed to include security in the operating system structure have exhibited either slow response times or awkward user interfaces-or both.

11.1 System Security

Computer security can be very complex and may be very confusing to many people. It can even be a controversial subject. Network administrators like to believe that their network is secure and those who break into networks may like to believe that they can break into any network.

Computer security is the prevention and protection of computer assets from unauthorized access, use, alteration, degradation, destruction, and other threats. There are two main subtypes: physical and logical. Physical computer security involves tangible protection devices, such as locks, cables, fences, safes, or vaults.

Logical computer security involves non-physical protection, such as that provided by authentication or encryption schemes. Make a point of noting that the physical versus non-physical (logical) distinction runs through a number of areas in computer science, despite minor differences in definition.

Computer security, in many ways, is about secrecy, not in the sense of being mysterious or clandestine, but because of the fact that you are always dealing with authorization and Authenticity.

The major technical areas of computer security are usually represented by the initials CIA: confidentiality, integrity, and authentication or availability. Confidentiality means that information cannot be access by unauthorized parties.

Confidentiality is also known as secrecy or privacy; breaches of confidentiality range from the embarrassing to the disastrous. Integrity means that information is protected against unauthorized changes that are not detectable to authorized users; many incidents of hacking compromise the integrity of databases and other resources.

Authentication means that users are who they claim to be. Availability means that resources are accessible by authorized parties; "denial of service" attacks, which are sometimes the topic of national news, are attacks against availability. Other important concerns of computer security professionals are access control and non-repudiation.

Maintaining access control means not only that users can access only those resources and services to which they are entitled, but also that they are not denied resources that they legitimately can expect to access. Non-repudiation implies that a person who sends a message cannot deny that he sent it and, conversely, that a person who has received a message cannot deny that he received

Notes

it. In addition to these technical aspects, the conceptual reach of computer security is broad and multifaceted.

Computer security touches draws from disciplines as ethics and risk analysis, and is concerned with topics such as computer crime; the prevention, detection, and remediation of attacks; and identity and anonymity in cyberspace.

While confidentiality, integrity, and authenticity are the most important concerns of a computer security manager, privacy is perhaps the most important aspect of computer security for everyday Internet users. Although users may feel that they have nothing to hide when they are registering with an Internet site or service, privacy on the Internet is about protecting one's personal information, even if the information does not seem sensitive.

Because of the ease with which information in electronic format can be shared among companies, and because small pieces of related information from different sources can be easily linked together to form a composite of, for example, a person's information seeking habits, it is now very important that individuals are able to maintain control over what information is collected about them, how it is used, who may use it, and what purpose it is used for.



Task System security is a major problem in this era. Discuss some important security techniques for operating system.

11.2 Security Problem

System security can mean several things. To have system security I need to protect the system from corruption and I need to protect the data on the system. There are many reasons why these need not be secure.

1. Malicious users may try to hack into the system to destroy it.
2. Power failure might bring the system down.
3. A badly designed system may allow a user to accidentally destroy important data.
4. A system may not be able to function any longer because one user fills up the entire disk with garbage.

Although discussions of security usually concentrate on the first of these possibilities, the latter two can be equally damaging the system in practice. One can protect against power failure by using un-interruptible power supplies (UPS). These are units which detect quickly when the power falls below a certain threshold and switch to a battery. Although the battery does not last forever-the UPS gives a system administrator a chance to halt the system by the proper route.

The problem of malicious users has been heightened in recent years by the growth of international networks. Anyone connected to a network can try to log on to almost any machine. If a machine is very insecure, they may succeed. In other words, you are not only looking at our local environment anymore, I must consider potential threats to system security to come from any source. The final point can be controlled by enforcing quotas on how much disk each user is allowed to use.

You can classify the security attacks into two types as mentioned below:

1. ***Direct:*** This is any direct attack on your specific systems, whether from outside hackers or from disgruntled insiders.
2. ***Indirect:*** This is general random attack, most commonly computer viruses, computer worms, or computer Trojan horses.

Notes

These security attacks make the study of security measures very essential for the following reasons:

1. ***To prevent loss of data:*** You don't want someone hacking into your system and destroying the work done by you or your team members. Even if you have good back-ups, you still have to identify that the data has been damaged (which can occur at a critical moment when a developer has an immediate need for the damaged data), and then restore the data as best you can from your backup systems.
2. ***To prevent corruption of data:*** Sometimes, the data may not completely be lost, but just be partially corrupted. This can be harder to discover, because unlike complete destruction, there is still data. If the data seems reasonable, you could go a long time before catching the problem, and cascade failure could result in serious problems spreading far and wide through your systems before discovery.
3. ***To prevent compromise of data:*** Sometimes it can be just as bad (or even worse) to have data revealed than to have data destroyed. Imagine the consequences of important trade secrets, corporate plans, financial data, etc. falling in the hands of your competitors.
4. ***To prevent theft of data:*** Some kinds of data are subject to theft. An obvious example is the list of credit card numbers belonging to your customers. Just about anything associated with money can be stolen.
5. ***To prevent sabotage:*** A disgruntled employee, a dishonest competitor, or even a stranger could use any combination of the above activities to maliciously harm your business. Because of the thought and intent, this is the most dangerous kind of attack, the kind that has the potential for the greatest harm to your business.

11.3 Program Threats

Any person, act, or object that poses a danger to computer security is called a threat. Any kind of policy, procedure, or action that recognizes, minimizes, or eliminates a threat or risk is called a countermeasure. Any kind of analysis that ties-in specific threats to specific assets with an eye toward determining the costs and/or benefits of protecting that asset is called risk, or risk assessment. Risk is always a calculated assumption made based on past occurrences.

Threat, on the other hand, is constant. Any kind of asset that is not working optimally and is mission-critical or essential to the organization, such as data that are not backed-up, is called a vulnerability, while anything imperfect is called a weakness. Any kind of counter measure that becomes fairly automated and meets the expectations of upper management is called a control, and there are many types of controls in a computer security environment, as well as threats, some of which are given below:

Malicious Threats

Category	Threat	OSI Layer	Definition	Typical Behaviors	Vulnerabilities	Prevention	Detection	Countermeasures
Malicious Software	Virus	Application	Malicious software that attaches itself to other software. For example, a patched software application in which the patch's algorithm is designed to implement the same patch on other applications, thereby replicating.	Replicates within computer system, potentially attaching itself to every software application Behavior categories ➢ Innocuous ➢ Humorous ➢ Data altering ➢ Catastrophic	All computers Common categories ➢ Boot sector ➢ Terminate and Stay Resident (TSR) ➢ Application software ➢ Stealth (or Chameleon) ➢ Mutation engine ➢ Network ➢ Mainframe	Limit connectivity. Limit downloads Use only authorized media for loading data and software Enforce mandatory access controls. Viruses generally cannot run unless host application is running	Changes in file sizes or date/time stamps Computer is slow starting or slow running Unexpected or frequent system failures Change of system date/time Low computer memory or increased bad blocks on disks	Contain, identify and recover Antivirus scanners- look for known viruses Antivirus monitors- look for virus related application behaviors Attempt to determine source of infection and issue alert

Notes

	Worm	Application Network	Malicious software which is a stand alone application	Often designed to propagate through a network, rather than just a single computer	Multitasking computers, especially those employing open network standards	Limit connectivity, employ firewalls Worms can run even without a host application	Computer is slow starting or slow running Unexpected or frequent system failures	Contain, identify and recover Attempt to determine source of infection and issue alert
	Trojan Horse	Application	A Worm which pretends to be a useful program or a Virus which is purposely attached to a useful program prior to distribution	Same as Virus or Worm, but also sometimes used to send information back to or make information available to perpetrator	Unlike Worms, which self propagate, Trojan Horses require user cooperation Untrained users are vulnerable	User cooperation allows Trojan Horses to bypass automated controls User training is best prevention	Same as Virus and Worm	Same as Virus and Worm Alert must be issued, not only to other system admins, but to all network users
	Time Bomb	Application	A Virus or Worm designed to activate at a certain date/time	Same as Virus or Worm, but widespread throughout organization upon trigger date	Same as Virus and Worm Time Bombs are usually found before the trigger date	Run associated anti-viral software immediately as available	Correlate user problem reports to find patterns indicating possible Time Bomb	Contain, identify and recover Attempt to determine source of infection and issue alert
	Logic Bomb	Application	A Virus or Worm designed to activate under certain conditions	Same as Virus or Worm	Same as Virus and Worm	Same as Virus and Worm	Correlate user problem reports indicating possible Logic Bomb	Contain, identify and recover Determine source and issue alert
	Rabbit	Application Network	A Worm designed to replicate to the point of exhausting computer resources	Rabbit consumes all CPU cycles, disk space or network resources, etc.	Multitasking computers, especially those on a network	Limit connectivity, employ firewalls	Computer is slow starting or running Frequent system failures	Contain, identify and recover Determine source and issue alert
	Bacterium	Application	A Virus designed to attach itself to the OS in particular (rather than any application in general) and exhaust computer resources, especially CPU cycles	Operating System consumes more and more CPU cycles, resulting eventually in noticeable delay in user transactions	Older versions of operating systems are more vulnerable than newer versions since hackers have had more time to write Bacterium	Limit write privileges and opportunities to OS files System administrators should work from non-admin accounts whenever possible	Changes in OS file sizes, date/time stamps Computer is slow in running Unexpected or frequent system failures	Antivirus scanners: look for known viruses Antivirus monitors: look for virus related system behaviors.
Spoofing	Spoofing	Network Data Link	Getting one computer on a network to pretend to have the identity of another computer, usually one with special access privileges, so as to obtain access to the other computers on the network	Spoofing computer often doesn't have access to user level commands so attempts to use automation level services, such as email or message handlers, are employed	Automation services designed for network interoperability are especially vulnerable, especially those adhering to open standards	Limit system privileges of automation services to minimum necessary Upgrade via security patches as they become available	Monitor transaction logs of automation services, scanning for unusual behaviors If automating this process do so off-line to avoid "tunneling" attacks	Disconnect automation services until patched or monitor automation access points, such as network sockets, scanning for next spoof, in attempt to trace back to perpetrator

Contd...

Notes

	Masquerade	Network	Accessing a computer by pretending to have an authorized user identity	Masquerading user often employs network or administrator command functions to access even more of the system, e.g., by attempting to download password, routing tables	Placing false or modified login prompts on a computer is a common way to obtain user IDs, as are Snooping, Scanning and Scavenging	Limit user access to network or administrator command functions Implement multiple levels of administrators, with different privileges for each	Correlate user identification with shift times or increased frequency of access Correlate user command logs with administrator command functions	Change user password or use standard administrator functions to determine access point, then trace back to perpetrator
Scanning	Sequential Scanning	Transport Network	Sequentially testing passwords/authentication codes until one is successful	Multiple users attempting network or administrator command functions, indicating multiple Masquerades	Since most login prompts have a time delay built in to foil automated scanning, accessing the encoded password table and testing it offline is a common technique	Enforce organizational password policies. Make even system administrator access to password files cumbersome	Correlate user identification with shift times Correlate user problem reports relevant to possible Masquerades	Change entire password file or use baiting tactics to trace back to perpetrator
	Dictionary Scanning	Application	Scanning through a dictionary of commonly used passwords/ authentication codes until one is successful	Multiple users attempting network or administrator command functions, indicating multiple Masquerades	Use of common words and names as passwords or authentication codes (so called "Joe Accounts")	Enforce organizational password policies	Correlate user identification with shift times Correlate user problem reports relevant to possible Masquerades	Change entire password file or use baiting tactics to trace back to perpetrator
Snooping (Eavesdropping)	Digital Snooping	Network	Electronic monitoring of digital networks to uncover passwords or other data	Users or even system administrators found online at unusual or off-shift hours Changes in behavior of network transport layer	Example of how COMSEC affects COMPUSEC Links can be more vulnerable to snooping than nodes	Employ data encryption Limit physical access to network nodes and links	Correlate user identification with shift times Correlate user problem reports. Monitor network performance	Change encryption schemes or employ network monitoring tools to attempt trace back to perpetrator
	Shoulder Surfing	Physical	Direct visual observation of monitor displays to obtain access	Authorized user found online at unusual or off-shift hours, indicating a possible Masquerade Authorized user attempting administrator command functions	"Sticky" notes used to record account and password information Password entry screens that do not mask typed text "Loitering" opportunities	Limit physical access to computer areas Require frequent password changes by users	Correlate user identification with shift times or increased frequency of access Correlate user command logs with administrator command functions	Change user password or use standard administrator functions to determine access point, then trace back to perpetrator
Scavenging	Dumpster Diving	All	Accessing discarded trash to obtain passwords and other data	Multiple users attempting network or administrator command functions, indicating multiple Masquerades	"Sticky" notes used to record account and password information System administrator printouts of user logs	Destroy discarded hardcopy	Correlate user identification with shift times Correlate user problem reports relevant to possible Masquerades	Change entire password file or use baiting tactics to trace back to perpetrator

Contd...

Notes

	Browsing	Application Network	Usually automated scanning of large quantities of unprotected data (discarded media or online "finger" type commands) to obtain clues as to how to achieve access	Authorized user found online at unusual or off-shift hours, indicating a possible Masquerade Authorized user attempting administrator command functions	"Finger" type services provide information to any and all users. The information is usually assumed safe but can give clues to passwords (e.g., spouse's name)	Destroy discarded media When on open source networks especially, disable "finger" type services	Correlate user identification with shift times or increased frequency of access Correlate user command logs with administrator command functions	Change user password or use standard administrator functions to determine access point, then trace back to perpetrator
Spamming	Spamming	Application Network	Overloading a system with incoming message or other traffic to cause system crashes	Repeated system crashes, eventually traced to overfull buffer or swap space	Open source networks especially vulnerable	Require authentication fields in message traffic	Monitor disk partitions, network sockets, etc. for overfull conditions	Analyze message headers to attempt trace back to perpetrator
Tunneling	Tunneling	Network	Any digital attack that attempts to get "under" a security system by accessing very low level system functions (e.g., device drivers, OS kernels)	Bizarre system behaviors such as unexpected disk accesses, unexplained device failures, halted security software, etc.	Tunneling attacks often occur by creating system emergencies to cause system reloading or initialization	Design security and audit capabilities into even the lowest level software, such as device drivers, shared libraries, etc.	Changes in date/time stamps for low level system files or changes in sector/block counts for device drivers	Patch or replace compromised drivers to prevent access Monitor suspected access points to attempt trace back to perpetrator

Unintentional Threats

Category	Threat	OSI Layer	Definition	Typical Behaviors	Vulnerabilities	Prevention	Detection	Countermeasures
Malfunction	Equipment Malfunction	All	Hardware operates in abnormal, unintended mode	Immediate loss of data due to abnormal shutdown Continuing loss of capability until equipment is repaired	Vital peripheral equipment is often more vulnerable than the computers themselves	Replication of entire system including all data and recent transactions	Hardware diagnostic systems	On-site replication of hardware components for quick recovery
	Software Malfunction	Application	Software behavior is in conflict with intended behavior	Immediate loss of data due to abnormal end Repeated system failure when re-fed "faulty" data	Software developed using ad hoc rather than defined formal processes	Comprehensive testing procedures and software designed for graceful degradation	Software diagnostic tools	Backup software and robust operating systems facilitate quick recovery
Human Error	Trap Door (Back door)	Application	System access for developers inadvertently left available after software delivery	Unauthorized system access enables viewing, alteration or destruction of data or software	Software developed outside defined organizational policies and formal methods	Enforce defined development policies Limit network and physical access	Audit trails of system usage, especially user identification logs	Close Trap Door or monitor ongoing access to trace back to perpetrator
	User/ Operator Error	All	Inadvertent alteration, manipulation or destruction of programs, data files or hardware	Incorrect data entered into system or incorrect behavior of system	Poor user documentation or training	Enforcement of training policies and separation of programmer/operator duties	Audit trails of system transactions	Backup copies of software and data On-site replication of hardware

Notes

Physical Threats

Category	Threat	OSI Layer	Definition	Typical Behaviors	Vulnerabilities	Prevention	Detection	Countermeasures
Physical Environment	Fire Damage	N/A	Physical destruction of equipment due to fire or smoke damage	Physical destruction of systems and supporting equipment	Systems located near potential fire hazards, e.g., fuel storage tanks	Off-site system replication, while costly, provides backup capability	On-site smoke alarms	Halon gas or FM200 fire extinguishers mitigate electrical and water damage
	Water Damage	N/A	Physical destruction of equipment due to water (including sprinkler) damage	Physical destruction of systems and supporting equipment	Systems located below ground or near sprinkler systems	Off-site system replication	Water detection devices	Computer rooms equipped with emergency drainage capabilities
	Power Loss	N/A	Computers or vital supporting equipment fail due to lack of power	Immediate loss of data due to abnormal shutdown, even after power returns Continuing loss of capability until power returns	Sites fed by above ground power lines are particularly vulnerable Power loss to computer room air conditioners can also be an issue	Dual or separate feeder lines for computers and supporting equipment	Power level alert monitors	Uninterruptible Power Supplies (UPS) Full scale standby power facilities where economically feasible
	Civil Disorder Vandalism	N/A	Physical destruction during operations other than war	Physical destruction of systems and supporting equipment	Sites located in some overseas environments, especially urban environments	Low profile facilities (no overt disclosure of high value nature of site)	Physical intrusion detection devices	Physical access restrictions and riot contingency policies
	Battle Damage	N/A	Physical destruction during military action	Physical destruction of systems and supporting equipment	Site located in theater	Off-site system replication OPSEC and low profile to prevent hostile targeting	Network monitoring systems	Hardened sites

11.4 System and 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.

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

Notes

network security threat is the infamous ping of death where an attacker sends ICMP packets 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.

Denial of Service Attack

On a modern time-sharing computer, any user takes some time and disk space, which is then not available to other users. By "denying service to authorized users", mean gobbling unreasonably large amounts of computer time or disk space, for example:

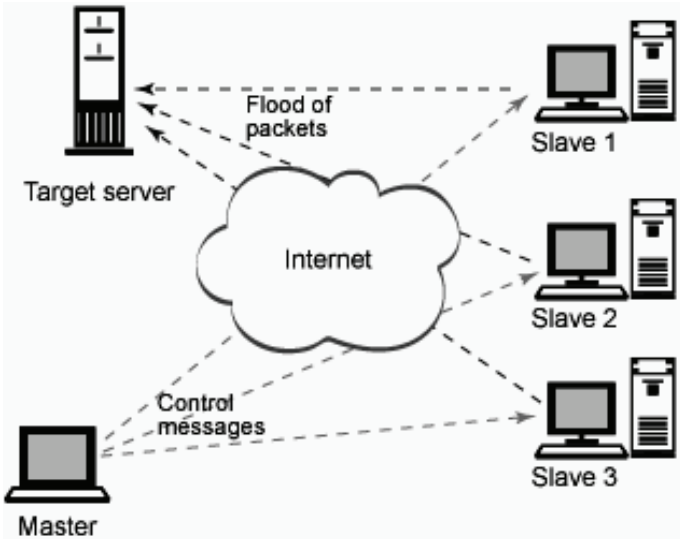
1. By sending large amounts of junk e-mail in one day, a so-called "mail bomb", Email bombing refers to sending a large number of emails to the victim resulting in the victim's email account (in case of an individual) or mail servers (in case of a company or an email service provider) crashing. In one case, a foreigner who had been residing in Shimla, India, for almost thirty years wanted to avail of a scheme introduced by the Shimla Housing Board to buy land at lower rates. When he made an application it was rejected on the grounds that the 169 schemes was available only for citizens of India. He decided to take his revenge. Consequently he sent thousands of mails to the Shimla Housing Board and repeatedly kept sending e-mails till their servers crashed.
2. By having the computer execute a malicious program that puts the processing unit into an infinite loop.
3. By flooding an Internet server with bogus requests for webpages, thereby denying legitimate users an opportunity to download a page and also possibly crashing the server. This is called a Denial of Service (DoS) attack. This involves flooding a computer resource with more requests than it can handle. This causes the resource (e.g. a web server) to crash thereby denying authorized users the service offered by the resource. Another variation to a typical denial of service attack is known as a Distributed Denial of Service (DDoS) attack wherein the perpetrators are many and are geographically widespread. It is very difficult to control such attacks. The attack is initiated by sending excessive demands to the victim's computer(s), exceeding the limit that the victim's servers can support and making the servers crash. Denial-of-service attacks have had an impressive history having, in the past, brought down websites like Amazon, CNN, Yahoo and eBay.

DoS (Denial-of-Service) attacks are probably the nastiest, and most difficult to address. These are the nastiest, because they're very easy to launch, difficult (sometimes impossible) to track, and it isn't easy to refuse the requests of the attacker, without also refusing legitimate requests for service.

Such attacks were fairly common in late 1996 and early 1997, but are now becoming less popular.

Notes

Figure 11.1: Denial of Service Attack



11.5 Cryptography as a Security Tools

Internet provides essential communication between tens of 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.

When your computer sends the information out, it scrambles it by using some key. This scrambled information would be gibberish to anyone who didn't have the correct key to unscramble it at the other end.

When the information reaches its destination, it gets unscrambled by using the key. This lets the person or website read the information correctly at the other end.

Websites that use an encrypted connection use something called SSL (Secure Sockets Layer) to secure the information going back and forth. This is how websites like Amazon or your bank can ensure your private information like passwords and credit card numbers are safe from prying eyes.

Cryptography can play many different roles in user authentication. Cryptographic authentication systems provide authentication capabilities through the use of cryptographic keys known or possessed only by authorized entities.

Cryptography also supports authentication through its widespread use in other authentication systems.

Example: Password systems often employ cryptography to encrypt stored password files, card/token system often employ cryptography to protect sensitive stored information, and hand-held password generators often employ cryptography to generate random, dynamic passwords.

Notes

Cryptography is frequently used in distributed applications to convey identification and authentication information from one system to another over a network. Cryptographic authentication systems authenticate a user based on the knowledge or possession of a cryptographic key. Cryptographic authentication systems can be based on either private key cryptosystems or public key cryptosystems.

Private key cryptosystems use the same key for the functions of both encryption and decryption. Cryptographic authentication systems based upon private key cryptosystems rely upon a shared key between the user attempting access and the authentication system.

Public key cryptosystems separate the functions of encryption and decryption, typically using a separate key to control each function. Cryptographic authentication systems based upon public key cryptosystems rely upon a key known only to the user attempting access.

Today's cryptography is more than encryption and decryption. Authentication is as fundamentally a part of our lives as privacy. You use authentication throughout our everyday lives – when you sign your name to some document for instance – and, as you move to a world where our decisions and agreements are communicated electronically, you need to have electronic techniques for providing authentication.

Cryptography provides mechanisms for such procedures. A digital signature binds a document to the possessor of a particular key, while a digital timestamp binds a document to its creation at a particular time. These cryptographic mechanisms can be used to control access to a shared disk drive, a high security installation, or a pay-per-view TV channel.

The field of cryptography encompasses other uses as well. With just a few basic cryptographic tools, it is possible to build elaborate schemes and protocols that allow us to pay using electronic money, to prove you know certain information without revealing the information itself and to share a secret quantity in such a way that a subset of the shares can reconstruct the secret.

While modern cryptography is growing increasingly diverse, cryptography is fundamentally based on problems that are difficult to solve. A problem may be difficult because its solution requires some secret knowledge, such as decrypting an encrypted message or signing some digital document. The problem may also be hard because it is intrinsically difficult to complete, such as finding a message that produces a given hash value.



Task Discuss work process of secure socket layers.

Computer encryption is based on the science of cryptography, which has been used throughout history. Before the digital age, the biggest users of cryptography were governments, particularly for military purposes.

Encryption is the transformation of data into a form that is as close to impossible as possible to read without the appropriate knowledge. Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who have access to the encrypted data. Decryption is the reverse of encryption; it is the transformation of encrypted data back into an intelligible form.

Encryption and decryption generally require the use of some secret information, referred to as a key. For some encryption mechanisms, the same key is used for both encryption and decryption; for other mechanisms, the keys used for encryption and decryption are different.

The existence of coded messages has been verified as far back as the Roman Empire. But most forms of cryptography in use these days rely on computers, simply because a human-based code is too easy for a computer to crack.

There are several tools of cryptography and these are:

Notes

11.5.1 Hashing

Hash functions, also called message digests and one-way encryption, are algorithms that, in some sense, use no key. Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered.

Hash algorithms are typically used to provide a digital fingerprint of a file's contents, often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Hash functions, then, provide a measure of the integrity of a file.

Broadly speaking, a cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable.

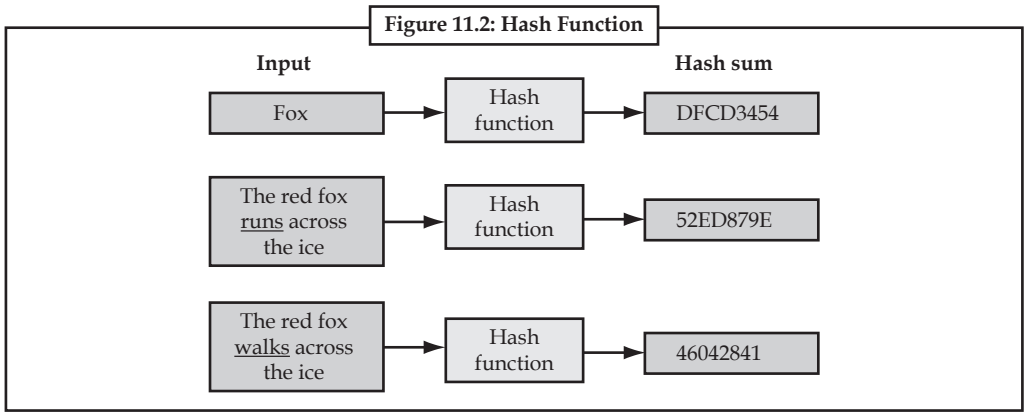
A cryptographic hash function is considered insecure if either of the following is computationally feasible:

1. Finding a (previously unseen) message that matches a given digest.
2. Finding "collisions", wherein two different messages have the same message digest.

An attacker who can do either of these things might, for example, use them to substitute an unauthorized message for an authorized one.

Ideally, it should not even be feasible to find two messages whose digests are substantially similar; nor would one want an attacker to be able to learn anything useful about a message given only its digest. Of course the attacker learns at least one piece of information, the digest itself, which for instance gives the attacker the ability to recognise the same message should it occur again.

Figure 11.2: Hash Function



11.5.2 Pretty Good Privacy (PGP)

It is one of today's most widely used public key cryptography programs. Developed by Philip Zimmermann in the early 1990s and long the subject of controversy, PGP is available as a plug-in for many e-mail clients, such as Claris Emailer, Microsoft Outlook/Outlook Express, and Qualcomm Eudora.

PGP can be used to sign or encrypt e-mail messages with the mere click of the mouse. Depending upon the version of PGP, the software uses SHA or MD5 for calculating the message hash; CAST, Triple-DES, or IDEA for encryption; and RSA or DSS/Diffie-Hellman for key exchange and digital signatures.

Notes

When PGP is first installed, the user has to create a key-pair. One key, the public key, can be advertised and widely circulated. The private key is protected by use of a passphrase. The passphrase has to be entered every time the user accesses their private key.

Box 11.1: A PGP Signed Message

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Hi Carol.
What was that pithy Groucho Marx quote?
/kess
-----BEGIN PGP SIGNATURE-----
Version: PGP for Personal Privacy 5.0
Charset: noconv
iQA/AwUBNFUdO5WOcz5SFtuEEQJx/ACaAgR97+vvDU6XWELV/GANjAAgBtUAnjG3
Sdfw2JgmZIOLNjFe7jP0Y8/M
=jUAU
-----END PGP SIGNATURE-----
```

Box 11.1 shows a PGP signed message. This message will not be kept secret from an eavesdropper, but a recipient can be assured that the message has not been altered from what the sender transmitted. In this instance, the sender signs the message using their own private key. The receiver uses the sender's public key to verify the signature; the public key is taken from the receiver's keyring based on the sender's e-mail address.



Note The signature process does not work unless the sender's public key is on the receiver's keyring.

Box 11.2: A PGP Encrypted Message

```
-----BEGIN PGP MESSAGE-----
Version: PGP for Personal Privacy 5.0
MessageID: DAdVB3wzpBr3YRunZwYvhK5gBKBXOb/m
qANQR1DBwU4D/TlT68XXuiUQCADfj2o4b4aFYBcWumA7hR1Wvz9rbv2BR6WbEUsy
ZBIEFtjyqCd96qF38sp9IQiJIKlNaZfx2GLRWikPZwchUXxB+AA5+lqsG/ELBvRa
c9XefaYpbbAZ6z6LkOQ+eE0XASe7aEEPfdxvZZT37dVyiyxuBBRYNLN8Bphdr2zv
z/9Ak4/OLnLiJRk05/2UNE5Z0a+3lcvITMmfGajvRhkXqocavPOKiin3hv7+Vx88
uLLem2/fQHZhGcQvkqZVqXx8SmNw5gzuvwjV1WHj9muDGBY0MkjiZIRI7azWnoU9
3KCnmpR60VO4rDRAS5uGl9fioSvze+q8XqxubaNsgdKkoD+tB/4u4c4tznLfw1L2
YBS+dzFDw5desMFSo7JkecAS4NB9jAu9K+f7PTAsesCBNETDd49BTOFFTWWavAfE
gLYcPrcn4s3EriUgvL3OzPR4P1chNu6sa3ZJkTBbriDoA3VpnqG3hxqfNyOlqAka
mJJuQ53Ob9ThaFH8YcE/VqUFdw+bQtrAJ6NpjIxi/x0FfOInhC/bBw7pDLXBFNaX
HdlLQRPQdrmnWskKznOSarxq4GjpRTQo4hpCRJJ5aU7tZO9HPTZXFG6iRIT0wa47
AR5nvkEKoIAjW5HaDKiJriuWLdtN4OXecWvxFsjR32ebz76U8aLpAK87GZEyTzBx
dV+lH0hwyT/y1cZQ/E5USePP4oKWF4uqquPee1OPeFMBo4CvuGyhZXD/18Ft/53Y
WIebvdiCqsOoabK3jEfdGExce63zDI0=
=MpRf
-----END PGP MESSAGE-----
```

Notes

Box 11.2 shows a PGP encrypted message (PGP compresses the file, where practical, prior to encryption because encrypted files lose their randomness and, therefore, cannot be compressed). In this case, public key methods are used to exchange the session key for the actual message encryption using secret-key cryptography. In this case, the receiver's e-mail address is the pointer to the public key in the sender's keyring; in fact, the same message can be sent to multiple recipients and the message will not be significantly longer since all that needs to be added is the session key encrypted by each receiver's private key. When the message is received, the recipient must use their private key to extract the session secret key to successfully decrypt the message (Box 11.3).

Box 11.3: The Decrypted Message

Hi Gary,

"Outside of a dog, a book is man's best friend.

Inside of a dog, it's too dark to read."

Carol

It is worth noting that PGP was one of the first so-called "hybrid cryptosystems" that combined aspects of SKC and PKC. When Zimmermann was first designing PGP in the late-1980s, he wanted to use RSA to encrypt the entire message. The PCs of the days, however, suffered significant performance degradation when executing RSA so he hit upon the idea of using SKC to encrypt the message and PKC to encrypt the SKC key.

PGP went into a state of flux in 2002. Zimmermann sold PGP to Network Associates, Inc. (NAI) in 1997 and himself resigned from NAI in early 2001. In March 2002, NAI announced that they were dropping support for the commercial version of PGP having failed to find a buyer for the product willing to pay what NAI wanted. In August 2002, PGP was purchased from NAI by PGP Corp. Meanwhile, there are many freeware versions of PGP available.

11.6 User Authentication

A user authentication method includes the steps of: inputting, by a user, a predetermined password having a plurality of digits; examining whether an input password includes an actual password that is predetermined by using less digits than the input password; authenticating the input password if the input password includes the actual password; and refusing to authenticate the input password if the input password does not include the actual password.

The user authentication method using the password is very useful for reinforcing the security by applying a simple processing, not necessarily consuming high costs and much time. Further, even when the password may be exposed to others, it is still safe. Also, although a password may be used in many cases in common, the security still can be reinforced by differentiating the input password. Most of all, the user can remember the actual password very easily, and yet get the same effect with changing the password.

In a wired, switched network, the policy that controls what traffic an authenticated user can send and receive is typically based on the port through which the user is connected rather than on the user's identity. This works when only one user is connected via a given port. Also, where physical barriers (locked doors, cardkeys etc.) are used to control access, it can be assumed that a user who has physical access to a port is authorized to connect on that port.

When wireless access enters the picture, the identity of the user becomes crucial. Since multiple users can connect through a single wireless access point, the assumption of one user per port is no longer valid, and port-based access policies do not work. All sorts of users – visitors,

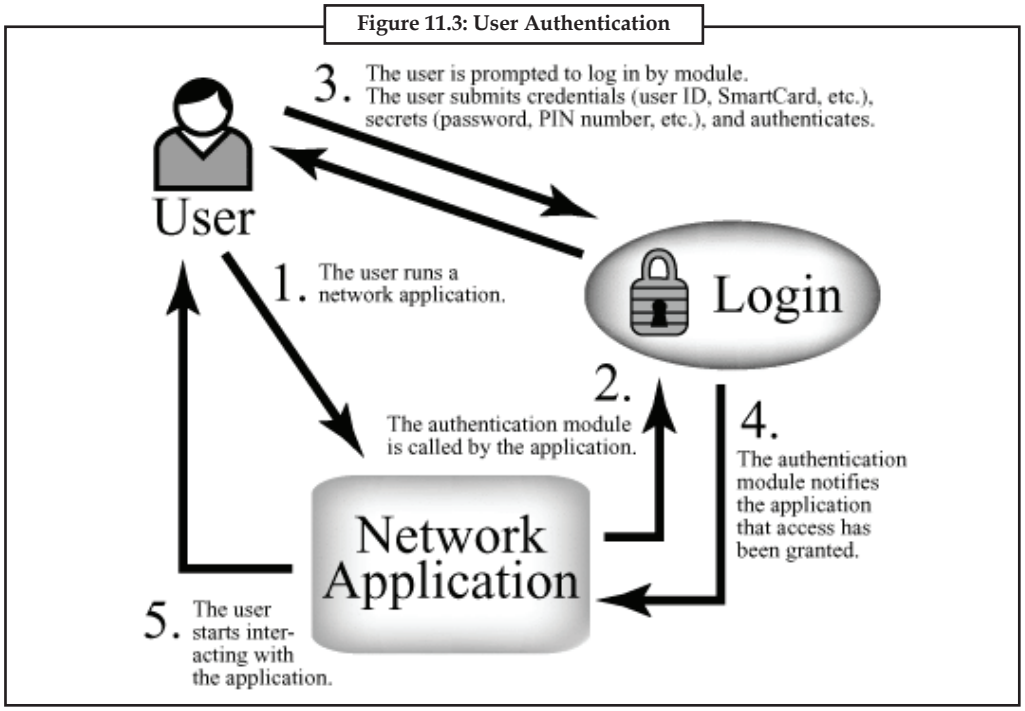
Notes

temporary workers, system administrators, the CFO – may all happen to access the network via the same access point, sharing the same port. A single set of access rights for that port would be too permissive for some users and too restrictive for others. Therefore, the system must be able to distinguish between the users on a port, and apply policy based on each user's identity.

Further, given the range of wireless access point signals, physical barriers become meaningless; given the mobility of wireless devices, users are no longer constrained to connect only through specific ports. In a wireless network, therefore, it is important both to determine who the user is when he attempts to connect and to track the user throughout his entire session on the network.

The system must be able to track the user if he or she physically moves (from desk to conference room, for example, roaming to a different access point and thus appearing on a different port) in order to enforce the appropriate policy for that user.

Figure 11.3: User Authentication



Task Authentication means permission. What about cryptography?

11.7 Implementing Security Defenses

An Intrusion Prevention System is a network security device that monitors network and/or system activities for malicious or unwanted behavior and can react, in real-time, to block or prevent those activities.

Network-based IPS, for example, will operate in-line to monitor all network traffic for malicious code or attacks. When an attack is detected, it can drop the offending packets while still allowing all other traffic to pass. Intrusion prevention technology is considered by some to be an extension of Intrusion Detection System (IDS) technology. The term "Intrusion Prevention System" was coined by Andrew Plato who was a technical writer and consultant for *NetworkICE.

Intrusion Prevention Systems (IPSs) evolved in the late 1990s to resolve ambiguities in passive network monitoring by placing detection systems in-line. Early IPS were IDS that were able

Notes

to implement prevention commands to firewalls and access control changes to routers. This technique fell short operationally for it created a race condition between the IDS and the exploit as it passed through the control mechanism.

Inline IPS can be seen as an improvement upon firewall technologies (snort inline is integrated into one), IPS can make access control decisions based on application content, rather than IP address or ports as traditional firewalls had done.

However, in order to improve performance and accuracy of classification mapping, most IPS use destination port in their signature format. As IPS systems were originally a literal extension of intrusion detection systems, they continue to be related.

Intrusion prevention systems may also serve secondarily at the host level to deny potentially malicious activity. There are advantages and disadvantages to host-based IPS compared with network-based IPS. In many cases, the technologies are thought to be complementary.

An Intrusion Prevention system must also be a very good Intrusion Detection system to enable a low rate of false positives. Some IPS systems can also prevent yet to be discovered attacks, such as those caused by a Buffer overflow.

The role of an IPS in a network is often confused with access control and application-layer firewalls. There are some notable differences in these technologies. While all share similarities, how they approach network or system security is fundamentally different.

An IPS is typically designed to operate completely invisibly on a network. IPS products do not typically claim an IP address on the protected network but may respond directly to any traffic in a variety of ways. (Common IPS responses include dropping packets, resetting connections, generating alerts, and even quarantining intruders.) While some IPS products have the ability to implement firewall rules, this is often a mere convenience and not a core function of the product.

Moreover, IPS technology offers deeper insight into network operations providing information on overly active hosts, bad logons, inappropriate content and many other network and application layer functions.

Application firewalls are a very different type of technology. An application firewall uses proxies to perform firewall access control for network and application-layer traffic. Some application-layer firewalls have the ability to do some IPS-like functions, such as enforcing RFC specifications on network traffic. Also, some application layer firewalls have also integrated IPS-style signatures into their products to provide real-time analysis and blocking of traffic.

Application firewalls do have IP addresses on their ports and are directly addressable. Moreover, they use full proxy features to decode and reassemble packets. Not all IPS perform full proxy-like processing. Also, application-layer firewalls tend to focus on firewall capabilities, with IPS capabilities as add-on. While there are numerous similarities between the two technologies, they are not identical and are not interchangeable.

Unified Threat Management (UTM), or sometimes called "Next Generation Firewalls" are also a different breed of products entirely. UTM products bring together multiple security capabilities on to a single platform.

A typical UTM platform will provide firewall, VPN, anti-virus, web filtering, intrusion prevention and anti-spam capabilities. Some UTM appliances are derived from IPS products such as 3Com's X-series products.

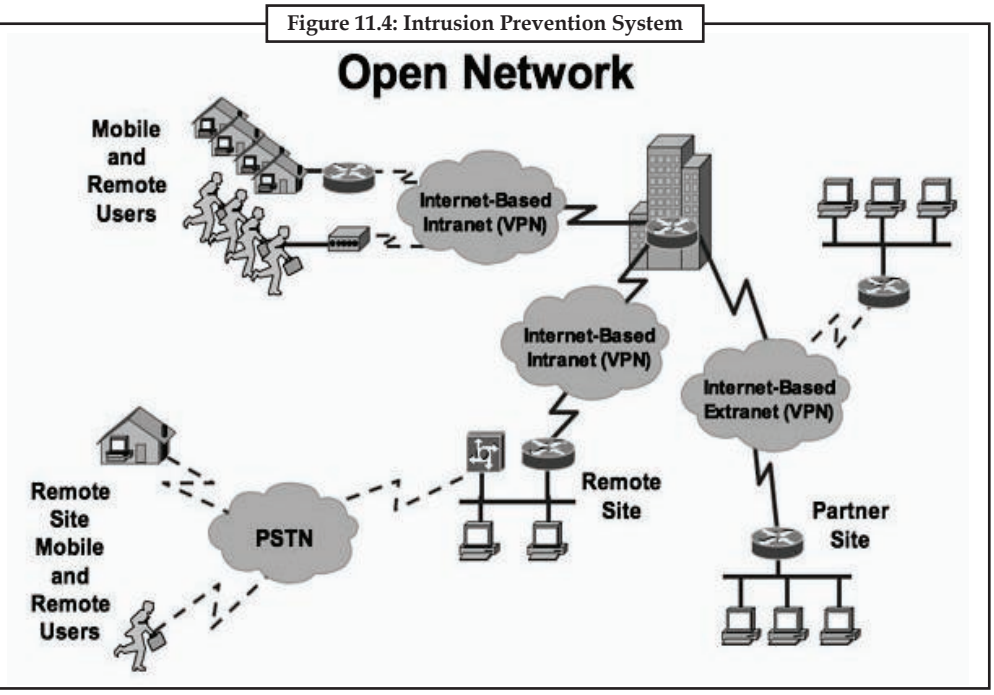
Others are derived from a combination with firewall products, such as Juniper's SSG or Cisco's Adaptive Security Appliances (ASA). And still others were derived from the ground up as a UTM appliance such as Fortinet or Astero. The main feature of a UTM is that it includes multiple security features on one appliance. IPS is merely one feature.

Notes

Access Control is also an entirely different security concept. Access control refers to general rules allowing hosts, users or applications access to specific parts of a network. Typically, access control helps organizations segment networks and limit access.

While an IPS has the ability to block access to users, hosts or applications, it does so only when malicious code has been discovered. As such, IPS does not necessarily serve as an access control device. While it has some access control abilities, firewalls and network access control (NAC) technologies are better suited to provide these features.

Figure 11.4: Intrusion Prevention System



11.8 Types of Intrusion Prevention System

Host-based

A Host-based IPS (HIPS) is one where the intrusion-prevention application is resident on that specific IP address, usually on a single computer. HIPS compliments traditional finger-print-based and heuristic antivirus detection methods, since it does not need continuous updates to stay ahead of new malware. As ill-intended code needs to modify the system or other software residing on the machine to achieve its evil aims, a truly comprehensive HIPS system will notice some of the resulting changes and prevent the action by default or notify the user for permission.

Extensive use of system resources can be a drawback of existing HIPS, which integrate firewall, system-level action control and sandboxing into a coordinated detection net, on top of a traditional AV product.

This extensive protection scheme may be warranted for a laptop computer frequently operating in untrusted environments (e.g. on cafe or airport Wi-Fi networks), but the heavy defenses may take their toll on battery life and noticeably impair the generic responsiveness of the computer as the HIPS protective component and the traditional AV product check each file on a PC to see if it is malware against a huge blacklist.

Notes

Alternatively if HIPS is combined with an AV product utilising whitelisting technology then there is far less use of system resources as many applications on the PC are trusted (whitelisted). HIPS as an application then becomes a real alternative to traditional antivirus products.

Network-based

A network-based IPS is one where the IPS application/hardware and any actions taken to prevent an intrusion on a specific network host(s) is done from a host with another IP address on the network (This could be on a front-end firewall appliance.).

Network Intrusion Prevention Systems (NIPSs) are purpose-built hardware/software platforms that are designed to analyze, detect and report on security related events. NIPS are designed to inspect traffic and based on their configuration or security policy, they can drop malicious traffic.

Content-based

A Content-based IPS (CBIPS) inspects the content of network packets for unique sequences, called signatures, to detect and hopefully prevent known types of attack such as worm infections and hacks.

Protocol Analysis

A key development in IDS/IPS technologies was the use of protocol analyzers. Protocol analyzers can natively decode application-layer network protocols, like HTTP or FTP. Once the protocols are fully decoded, the IPS analysis engine can evaluate different parts of the protocol for anomalous behavior or exploits. For example, the existence of a large binary file in the User-Agent field of an HTTP request would be very unusual and likely an intrusion. A protocol analyzer could detect this anomalous behavior and instruct the IPS engine to drop the offending packets.

Not all IPS/IDS engines are full protocol analyzers. Some products rely on simple pattern recognition techniques to look for known attack patterns. While this can be sufficient in many cases, it creates an overall weakness in the detection capabilities. Since many vulnerabilities have dozens or even hundreds of exploit variants, pattern recognition-based IPS/IDS engines can be evaded. For example, some pattern recognition engines require hundreds of different signatures (patterns) to protect against a single vulnerability. This is because they must have a different pattern for each exploit variant. Protocol analysis-based products can often block exploits with a single signature that monitors for the specific vulnerability in the network communications.

Rate-based

Rate-based IPS (RBIPS) are primarily intended to prevent Denial of Service and Distributed Denial of Service attacks. They work by monitoring and learning normal network behaviors. Through real-time traffic monitoring and comparison with stored statistics, RBIPS can identify abnormal rates for certain types of traffic e.g. TCP, UDP or ARP packets, connections per second, packets per connection, packets to specific ports etc. Attacks are detected when thresholds are exceeded. The thresholds are dynamically adjusted based on time of day, day of the week etc., drawing on stored traffic statistics.

Unusual but legitimate network traffic patterns may create false alarms. The system's effectiveness is related to the granularity of the RBIPS rulebase and the quality of the stored statistics.

Once an attack is detected, various prevention techniques may be used such as rate-limiting specific attack-related traffic types, source or connection tracking and source-address, port or protocol filtering (black-listing) or validation (white-listing).

Notes



Task How will you implements security defenses on your system? Discuss.

11.9 Implementation Challenges

There are a number of challenges to the implementation of an IPS device that do not have to be faced when deploying passive-mode IDS products. These challenges all stem from the fact that the IPS device is designed to work in-line, presenting a potential choke point and single point of failure.

If a passive IDS fails, the worst that can happen is that some attempted attacks may go undetected. If an in-line device fails, however, it can seriously impact the performance of the network.

Perhaps latency rises to unacceptable values, or perhaps the device fails closed, in which case you have a self-inflicted Denial of Service condition on your hands. On the bright side, there will be no attacks getting through! But that is of little consolation if none of your customers can reach your e-commerce site.

Even if the IPS device does not fail altogether, it still has the potential to act as a bottleneck, increasing latency and reducing throughput as it struggles to keep up with up to a Gigabit or more of network traffic.

Devices using off-the-shelf hardware will certainly struggle to keep up with a heavily loaded Gigabit network, especially if there is a substantial signature set loaded, and this could be a major concern for both the network administrator – who could see his carefully crafted network response times go through the roof when a poorly designed IPS device is placed in-line – as well as the security administrator, who will have to fight tooth and nail to have the network administrator allow him to place this unknown quantity amongst his high performance routers and switches.

As an integral element of the network fabric, the Network IPS device must perform much like a network switch. It must meet stringent network performance and reliability requirements as a prerequisite to deployment, since very few customers are willing to sacrifice network performance and reliability for security. A NIPS that slows down traffic, stops good traffic, or crashes the network is of little use.

Dropped packets are also an issue, since if even one of those dropped packets is one of those used in the exploit data stream it is possible that the entire exploit could be missed. Most high-end IPS vendors will get around this problem by using custom hardware, populated with advanced FPGAs and ASICs – indeed, it is necessary to design the product to operate as much as a switch as an intrusion detection and prevention device.

It is very difficult for any security administrator to be able to characterize the traffic on his network with a high degree of accuracy. What is the average bandwidth? What are the peaks? Is the traffic mainly one protocol or a mix? What is the average packet size and level of new connections established every second – both critical parameters that can have detrimental effects on some IDS/IPS engines? If your IPS hardware is operating "on the edge", all of these are questions that need to be answered as accurately as possible in order to prevent performance degradation.

Another potential problem is the good old false positive. The bane of the security administrator's life (apart from the script kiddie, of course!), the false positive rears its ugly head when an exploit signature is not crafted carefully enough, such that legitimate traffic can cause it to fire accidentally. Whilst merely annoying in a passive IDS device, consuming time and effort on the part of the security administrator, the results can be far more serious and far reaching in an in-line IPS appliance.

Notes

Once again, the result is a self-inflicted Denial of Service condition, as the IPS device first drops the "offending" packet, and then potentially blocks the entire data flow from the suspected hacker.

If the traffic that triggered the false positive alert was part of a customer order, you can bet that the customer will not wait around for long as his entire session is torn down and all subsequent attempts to reconnect to your e-commerce site (if he decides to bother retrying at all, that is) are blocked by the well-meaning IPS.

Another potential problem with any Gigabit IPS/IDS product is, by its very nature and capabilities, the amount of alert data it is likely to generate. On such a busy network, how many alerts will be generated in one working day? Or even one hour? Even with relatively low alert rates of ten per second, you are talking about 36,000 alerts every hour. That is 864,000 alerts each and every day.

The ability to tune the signature set accurately is essential in order to keep the number of alerts to an absolute minimum. Once the alerts have been raised, however, it then becomes essential to be able to process them effectively. Advanced alert handling and forensic analysis capabilities including detailed exploit information and the ability to examine packet contents and data streams can make or break a Gigabit IDS/IPS product.

Of course, one point in favour of IPS when compared with IDS is that because it is designed to prevent the attacks rather than just detect and log them, the burden of examining and investigating the alerts – and especially the problem of rectifying damage done by successful exploits – is reduced considerably.

11.10 Firewall to Protect Systems and Networks

A firewall is a dedicated appliance, or software running on another computer, which inspects network traffic passing through it, and denies or permits passage based on a set of rules.

Firewalls can be implemented in both hardware and 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 examines each message and blocks those that do not meet the specified security criteria.

Basically, a firewall is a barrier to keep destructive forces away from your property. In fact, that's why its called a firewall. Its job is similar to a physical firewall that keeps a fire from spreading from one area to the next.

A firewall is simply a program or hardware device that filters the information coming through the Internet connection into your private network or computer system. If an incoming packet of information is flagged by the filters, it is not allowed through. Let's say that you work at a company with 500 employees. The company will therefore have hundreds of computers that all have network cards connecting them together.

In addition, the company will have one or more connections to the Internet through something like T1 or T3 lines. Without a firewall in place, all of those hundreds of computers are directly accessible to anyone on the Internet. A person who knows what he or she is doing can probe those computers, try to make FTP connections to them, try to make telnet connections to them and so on. If one employee makes a mistake and leaves a security hole, hackers can get to the machine and exploit the hole.

With a firewall in place, the landscape is much different. A company will place a firewall at every connection to the Internet (for example, at every T1 line coming into the company). The firewall

Notes

can implement security rules. For example, one of the security rules inside the company might be:

Out of the 500 computers inside this company, only one of them is permitted to receive public FTP traffic. Allow FTP connections only to that one computer and prevent them on all others.

A company can set up rules like this for FTP servers, Web servers, Telnet servers and so on. In addition, the company can control how employees connect to Web sites, whether files are allowed to leave the company over the network and so on. A firewall gives a company tremendous control over how people use the network.

Firewalls use one or more of three methods to control traffic flowing in and out of the network:

1. ***Packet filtering:*** Packets (small chunks of data) are analyzed against a set of filters. Packets that make it through the filters are sent to the requesting system and all others are discarded.
2. ***Proxy service:*** Information from the Internet is retrieved by the firewall and then sent to the requesting system and vice versa.
3. ***Stateful inspection:*** A newer method that doesn't examine the contents of each packet but instead compares certain key parts of the packet to a database of trusted information.

Information traveling from inside the firewall to the outside is monitored for specific defining characteristics, then incoming information is compared to these characteristics. If the comparison yields a reasonable match, the information is allowed through.

Otherwise it is discarded. There are many creative ways that unscrupulous people use to access or abuse unprotected computers:

1. ***Remote login:*** When someone is able to connect to your computer and control it in some form. This can range from being able to view or access your files to actually running programs on your computer.
2. ***Application backdoors:*** Some programs have special features that allow for remote access. Others contain bugs that provide a backdoor, or hidden access, that provides some level of control of the program.
3. ***SMTP session hijacking:*** SMTP is the most common method of sending e-mail over the Internet. By gaining access to a list of e-mail addresses, a person can send unsolicited junk e-mail (spam) to thousands of users. This is done quite often by redirecting the e-mail through the SMTP server of an unsuspecting host, making the actual sender of the spam difficult to trace.
4. ***Operating system bugs:*** Like applications, some operating systems have backdoors. Others provide remote access with insufficient security controls or have bugs that an experienced hacker can take advantage of.
5. ***Denial of service:*** You have probably heard this phrase used in news reports on the attacks on major Web sites. This type of attack is nearly impossible to counter. What happens is that the hacker sends a request to the server to connect to it. When the server responds with an acknowledgement and tries to establish a session, it cannot find the system that made the request. By inundating a server with these unanswerable session requests, a hacker causes the server to slow to a crawl or eventually crash.
6. ***E-mail bombs:*** An e-mail bomb is usually a personal attack. Someone sends you the same e-mail hundreds or thousands of times until your e-mail system cannot accept any more messages.
7. ***Macros:*** To simplify complicated procedures, many applications allow you to create a script of commands that the application can run. This script is known as a macro. Hackers

Notes

have taken advantage of this to create their own macros that, depending on the application, can destroy your data or crash your computer.

8. ***Viruses:*** Probably the most well-known threat is computer viruses. A virus is a small program that can copy itself to other computers. This way it can spread quickly from one system to the next. Viruses range from harmless messages to erasing all of your data.

9. ***Spam:*** Typically harmless but always annoying, spam is the electronic equivalent of junk mail. Spam can be dangerous though. Quite often it contains links to Web sites. Be careful of clicking on these because you may accidentally accept a cookie that provides a backdoor to your computer.

10. ***Redirect bombs:*** Hackers can use ICMP to change (redirect) the path information takes by sending it to a different router. This is one of the ways that a denial of service attack is set up.

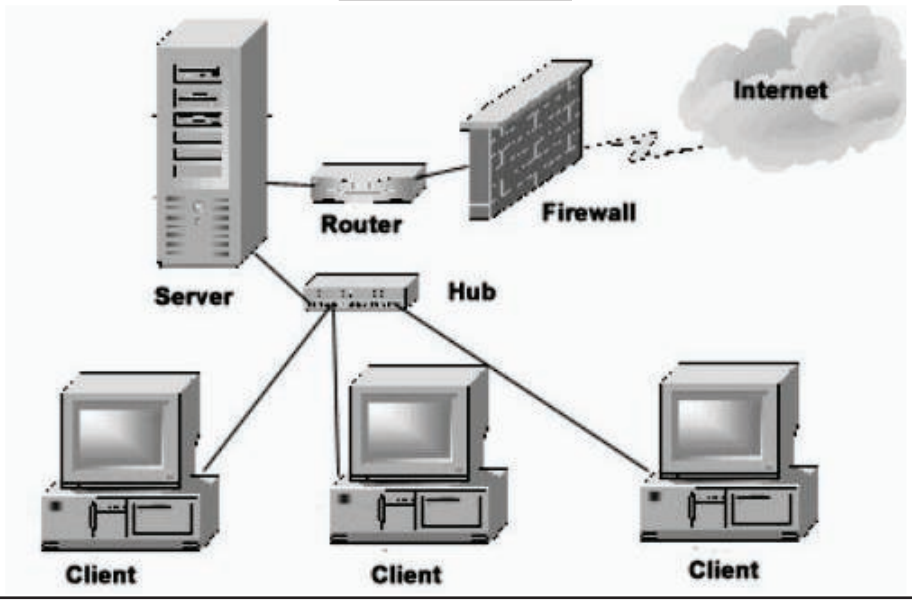
11. ***Source routing:*** In most cases, the path a packet travels over the Internet (or any other network) is determined by the routers along that path. But the source providing the packet can arbitrarily specify the route that the packet should travel. Hackers sometimes take advantage of this to make information appear to come from a trusted source or even from inside the network! Most firewall products disable source routing by default.

Task Discuss the role of firewall in the system security.

Some of the items in the list above are hard, if not impossible, to filter using a firewall. While some firewalls offer virus protection, it is worth the investment to install anti-virus software on each computer. And, even though it is annoying, some spam is going to get through your firewall as long as you accept e-mail.

The level of security you establish will determine how many of these threats can be stopped by your firewall. The highest level of security would be to simply block everything. Obviously that defeats the purpose of having an Internet connection. But a common rule of thumb is to block everything, then begin to select what types of traffic you will allow.

Figure 11.5: Firewall



Notes

You can also restrict traffic that travels through the firewall so that only certain types of information, such as e-mail, can get through. This is a good rule for businesses that have an experienced network administrator that understands what the needs are and knows exactly what traffic to allow through.

For most of us, it is probably better to work with the defaults provided by the firewall developer unless there is a specific reason to change it. One of the best things about a firewall from a security standpoint is that it stops anyone on the outside from logging onto a computer in your private network.

While this is a big deal for businesses, most home networks will probably not be threatened in this manner. Still, putting a firewall in place provides some peace of mind.

11.11 Summary

- A firewall is a software program or device that monitors, and sometimes controls, all transmissions between an organization's internal network and the Internet.
- However large the network, a firewall is typically deployed on the network's edge to prevent inappropriate access to data behind the firewall.
- The firewall ensures that all communication in both directions conforms to an organization's security policy.
- A denial-of-service attack (DoS attack) or distributed denial-of-service attack (DDoS attack) is an attempt to make a computer resource unavailable to its intended users.
- Perpetrators of DoS attacks typically target sites or services hosted on high-profile web servers such as banks, credit card payment gateways, and even DNS root servers.
- It is very difficult to control such attacks. DoS (Denial-of-Service) attacks are probably the nastiest, and most difficult to address.

11.12 Keywords

Computer security: It is more like providing means to protect a single PC against outside intrusion.

Decryption: It is the reverse process of converting encoded data to its original un-encoded form, plaintext.

Encryption: It is a process of coding information which could either be a file or mail message in into cipher text a form unreadable without a decoding key in order to prevent anyone except the intended recipient from reading that data.

Hash function: It is one-way encryption that uses no key.

Intrusion detection system: It gathers and analyzes information from various areas within a computer or a network to identify possible security breaches, which include both intrusions and misuse.

Network intrusion detection system: It is an independent platform which identifies intrusions by examining network traffic and monitors multiple hosts.

Securing network infrastructure: It is like securing possible entry points of attacks on a country by deploying appropriate defense.

Notes

11.13 Self Assessment

State whether the following statements are true or false:

1. Physical computer security involves tangible protection devices, such as locks, cables, fences, safes, or vaults.
2. Confidentiality is also known as secrecy or privacy.
3. Resource attacks are the first category of network security threats.
4. DoS (Denial-of-Service) attacks are probably the nastiest, and most difficult to address.
5. Websites that use an encrypted connection use something called SSL (Secure Sockets Layer) to secure the information going back and forth.

Fill in the blanks:

6. ..... authentication systems authenticate a user based on the knowledge or possession of a cryptographic key.
7. Encryption and ..... generally require the use of some secret information, referred to as a key.
8. ..... can be used to sign or encrypt e-mail messages with the mere click of the mouse.
9. Hash functions, also called ..... and one-way encryption
10. Intrusion Prevention Systems (IPSs) evolved in the late ..... .

11.14 Review Questions

1. What do you mean by system security?
2. Explain security problem.
3. Describe program threats.
4. Write short note on "denial of service" attack.
5. What do you mean by hashing?
6. Describe "Pretty Good Privacy" concept.
7. Explain user authentication process in detail.
8. How will you implement security defenses? Explain
9. Describe firewall concept.
10. Describe various types of intrusion prevention system.

Answers: Self Assessment

1.	True	2.	True	3.	False	4.	True
5.	True	6.	Cryptographic	7.	decryption	8.	PGP
9.	message digests	10.	1990s				

Notes

11.15 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Published by Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 12: Security Solution

Notes

CONTENTS

Objectives
Introduction
12.1 Encryption
12.2 How Encryption Works?
12.3 Symmetric or Secret-key Encryption
12.4 Public-key Encryption
12.5 Digital Encryption Standards (DES)
12.6 Triple—DES
12.7 RSA System
12.8 Comparison between Symmetric and Public Key Encryption
12.8.1 Symmetric Key Encryption
12.8.2 Public Key Cryptography (Encryption)
12.9 Digital Signature
12.9.1 Signing Process
12.9.2 Advantages of Digital Signature
12.10 Digital Certificate
12.11 Certificate Authority
12.12 Enterprise Authentication using Digital Certificates
12.13 Summary
12.14 Keywords
12.15 Self Assessment
12.16 Review Questions
12.17 Further Readings

Objectives

After studying this unit, you will be able to:

- Know the concept of encryption
- Describe secret and public key encryption
- Explain DES
- Describe digital signature
- Describe digital certificate concept

Notes

Introduction

Encryption is essentially the process of encoding – or hiding – the information you send across the internet in a way that it can only be read by the person or website it is meant for. There are various ways this is handled on the net.

Encryption uses a "key" - a certain sequence of numbers that is unique and only "known" by your computer and the one you're sending information to.

When your computer sends the information out, it scrambles it by using this key as a basis. This scrambled information would be gibberish to anyone who didn't have the correct key to unscramble it at the other end.

When the information reaches its destination, it gets unscrambled by using the key. This lets the person or website read the information correctly at the other end.

Websites that use an encrypted connection use something called SSL (Secure Sockets Layer) to secure the information going back and forth. This is how websites like Amazon or your bank can ensure your private information like passwords and credit card numbers are safe from prying eyes.

There are different strengths of encryption codes. 40 bit encryption is the simplest, but it is relatively easy to crack. Most secure websites use 128 bit encryption, which is practically impossible to decode. You might even see 256 bit encryption is some very high-security cases.

12.1 Encryption

Encryption is a process of coding information which could either be a file or mail message in into cipher text a form unreadable without a decoding key in order to prevent anyone except the intended recipient from reading that data. Decryption is the reverse process of converting encoded data to its original un-encoded form, plaintext.

A key in cryptography is a long sequence of bits used by encryption/decryption algorithms. For example, the following represents a hypothetical 40-bit key:

00001010 01101001 10011110 00011100 01010101

A given encryption algorithm takes the original message, and a key, and alters the original message mathematically based on the key's bits to create a new encrypted message. Likewise, a decryption algorithm takes an encrypted message and restores it to its original form using one or more keys.

When a user encodes a file, another user cannot decode and read the file without the decryption key. Adding a digital signature, a form of personal authentication, ensures the integrity of the original message.

To encode plaintext, an encryption key is used to impose an encryption algorithm onto the data. To decode cipher, a user must possess the appropriate decryption key. A decryption key consists of a random string of numbers, from 40 through 2,000 bits in length. The key imposes a decryption algorithm onto the data. This decryption algorithm reverses the encryption algorithm, returning the data to plaintext. The longer the encryption key is, the more difficult it is to decode. For a 40-bit encryption key, over one trillion possible decryption keys exist.

There are two primary approaches to encryption: symmetric and public-key. Symmetric encryption is the most common type of encryption and uses the same key for encoding and decoding data. This key is known as a session key. Public-key encryption uses two different keys, a public key and a private key. One key encodes the message and the other decodes it. The public key is widely distributed while the private key is secret.

Notes

Aside from key length and encryption approach, other factors and variables impact the success of a cryptographic system. For example, different cipher modes, in coordination with initialization vectors and salt values, can be used to modify the encryption method. Cipher modes define the method in which data is encrypted. The stream cipher mode encodes data one bit at a time. The block cipher mode encodes data one block at a time. Although block cipher tends to execute more slowly than stream cipher.

12.2 How Encryption Works?

Encryption or encoding information helps prevent it by unauthorized user. Both the sender and receiver have to know what set of rules (called cipher) was used to transform original information in to its cipher text (code) form – cipher text.

Simple cipher might to be add an arbitrary number of characters to all the character in the message.



Example: Say "Udupa" — is the original name

"Irida" — is the cipher text (arbitrary no. chosen is "!2")

1 2 3 4 5 6 7 8 9 10 11 12

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The decrypt (decode) "Irido", First, start counting letter "I" & replace the letter "I" in the coded text with the letter which comes after the count 12. So, "I" is replaced by "U", similarly for other letters to get back the original name "udupa" It is clear from the above example that both the sender and recipient has to know the arbitrary number chosen in order to encrypt & decrypt the original message.

Basically encryption has two parts:

1. Algorithm-a cryptographic algorithm is mathematical function.
2. Key-string of digit.

In the above example counting forward (to decrypt) & backward (encrypt) is the algorithm part. Key used is 12.

Cryptographic algorithm combines the plain text or other intelligible information with a string of digit called key's to produce unintelligible cipher text. But some encryption algorithms does not use a key.

Encryption on key-based system offers two important advantages.

1. It is difficult to come up with new-algorithm each time to communicate privately with new correspondent. By using a key, same algorithm can be used with many people with different key for each correspondent.
2. It is easy to change the key in case of any mal-practice rather than going for a new algorithm.

The number of keys each algorithm can support depends on the number of bits in the key. Ex-8 bit key allows only 256 possible numeric combinations, each key is called a key of 2^8. Hence more the digits (bit – length) more the possible keys and more difficult to crack an encrypted message. For example, to unlock a physical number zero and nine, at one stage the lock-gets unlocked. If it is a three digit decimal number, the p[ossible combinations vary from 000-999. Similarly if a 1000 bit (binary) key wre used on a computer which is capable of guessing one million keys every second could still take many centuries to discover the right key hence the security of the encryption algorithm correlates with the length of the key. Trying each possible key to find the right one to get back original message is called **Brute—force method.**

Notes

12.3 Symmetric or Secret-key Encryption

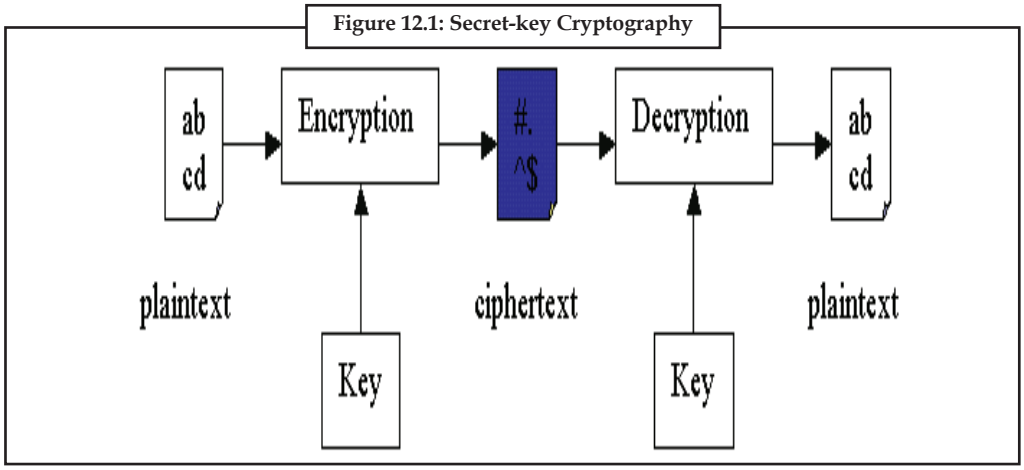
With secret key cryptography, a single key is used for both encryption and decryption. As shown in Figure 12.1, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or rule-set) to decrypt the message and 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.

Figure 12.1: Secret-key Cryptography



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.

Notes

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.

12.4 Public-key Encryption

Public-key cryptography has been said to be the most significant new development in cryptography in the last 300-400 years. Modern PKC was first described publicly by Stanford University professor Martin Hellman and graduate student Whitfield Diffie in 1976. Their paper described a two-key crypto system in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key.

Public-key cryptography, also known as asymmetric cryptography, is a form of cryptography in which a user has a pair of cryptographic keys—a public key and a private key. The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be practically derived from the public key. A message encrypted with the public key can be decrypted only with the corresponding private key.

The two main branches of public key cryptography are:

1. ***Public key encryption —*** a message encrypted with a recipient's public key cannot be decrypted by anyone except the recipient possessing the corresponding private key. This is used to ensure confidentiality.
2. ***Digital signatures —*** a message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender signed it and that the message has not been tampered with. This is used to ensure authenticity.

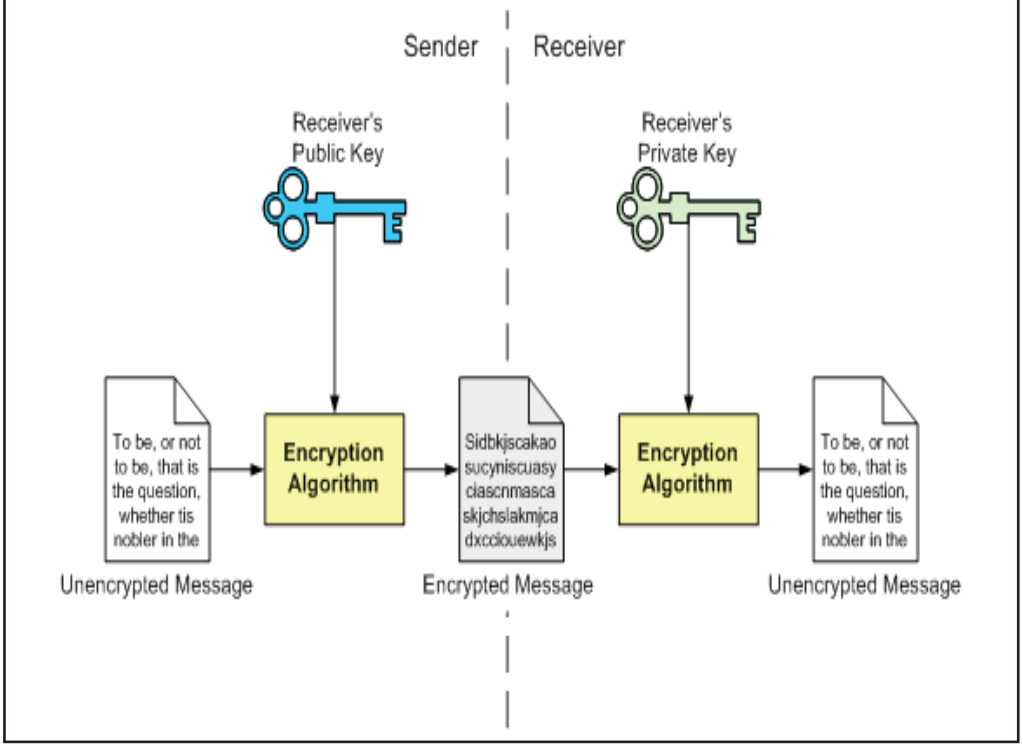
An analogy for public-key encryption is that of a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender.

A central problem for public-key cryptography is proving that a public key is authentic, and has not been tampered with or replaced by a malicious third party. The usual approach to this problem is to use a public-key infrastructure (PKI), in which one or more third parties, known as certificate authorities, certify ownership of key pairs. Another approach, used by PGP, is the "web of trust" method to ensure authenticity of key pairs.

Notes

Figure 12.2: Public-key Cryptography



Public key techniques are much more computationally intensive than purely symmetric algorithms. The judicious use of these techniques enables a wide variety of applications. In practice, public key cryptography is used in combination with secret-key methods for efficiency reasons. For encryption, the sender encrypts the message with a secret-key algorithm using a randomly generated key, and that random key is then encrypted with the recipient's public key. For digital signatures, the sender hashes the message (using a cryptographic hash function) and then signs the resulting "hash value". Before verifying the signature, the recipient also computes the hash of the message, and compares this hash value with the signed hash value to check that the message has not been tampered with.

Task One key encodes the message and the other decodes it. Then why we use encryption techniques to encode and decode the message.

12.5 Digital Encryption Standards (DES)

The data encryption standard (DES) specifies a FIPS (Federal Information Processing Standards) approved cryptography algorithm. Encrypting data converts it to an unintelligible form called cipher. Decrypting cipher converts the data back to its original form called plaintext. The algorithm described in this standards specifies both enciphering and deciphering operations which are based on a binary number called a key.

A key consists of 64 binary digits ("O"s or "1"s)of which 56 bits are randomly generated and used directly by the algorithm. The other 8 bits, which are not used by the algorithm, are used for error detection. The 8 error detecting bits are set to make the parity of each 8-bit byte of the key odd, i.e. there is an odd number of "1"s in each 8-bit byte. Authorized users of encrypted computer data must have the key that was used to encipher the data in order to decrypt it.

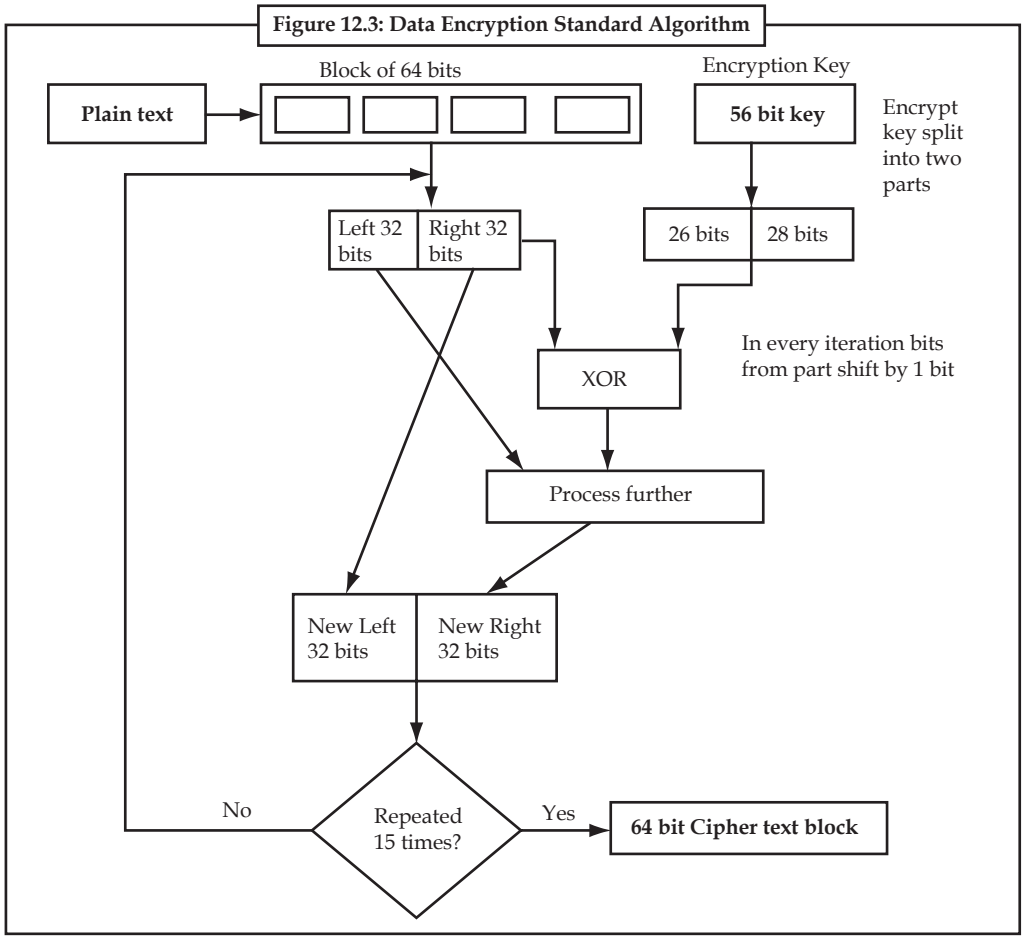
Notes

The encryption algorithm specified in this standard is commonly known among those using the standard. The unique key chosen for use in a particular application makes the results of encrypting data using the algorithm unique. Selection of a different key causes the cipher that is produced for any given set of inputs to be different. The cryptographic security of the data depends on the security provided for the key used to encipher and decipher the data.

Data can be recovered from cipher only by using exactly the same key used to encipher it. Unauthorized recipients of the cipher whop know the algorithm but do not have the correct key cannot drive the original data algorithmically. However, anyone who does have the key and the algorithm can easily decipher the cipher and obtain the original data. A standard algorithm based on a secure key thus provides a basis for exchanging encrypted computer data by issuing the key used to encipher it to those unauthorized to have the data.

Data that is considered sensitive by the responsible authority, data that, has a high value, or data that represents a high value should be cryptographically protected if it is vulnerable to unauthorized disclosure or undetected modification during transmission or while in storage. A risk analysis should be performed under the direction of a responsible authority to determine potential threats. The costs of providing cryptographic protection using this standards as well as alternative methods of providing this protection and their respective costs should be projected. A responsible authority then should make a decision, based on these analyses, whether or not to use cryptographic protection and this standard.

Figure 12.3: Data Encryption Standard Algorithm



DES divides the message into 64 bit block of plain text and a 56-bit key is used on the blocks to encrypt the block. The same key is used for encryption and decryption. The algorithms available as a software product and also as hardware chip. A simplified DES algorithm is presented below, in Figure 12.3.

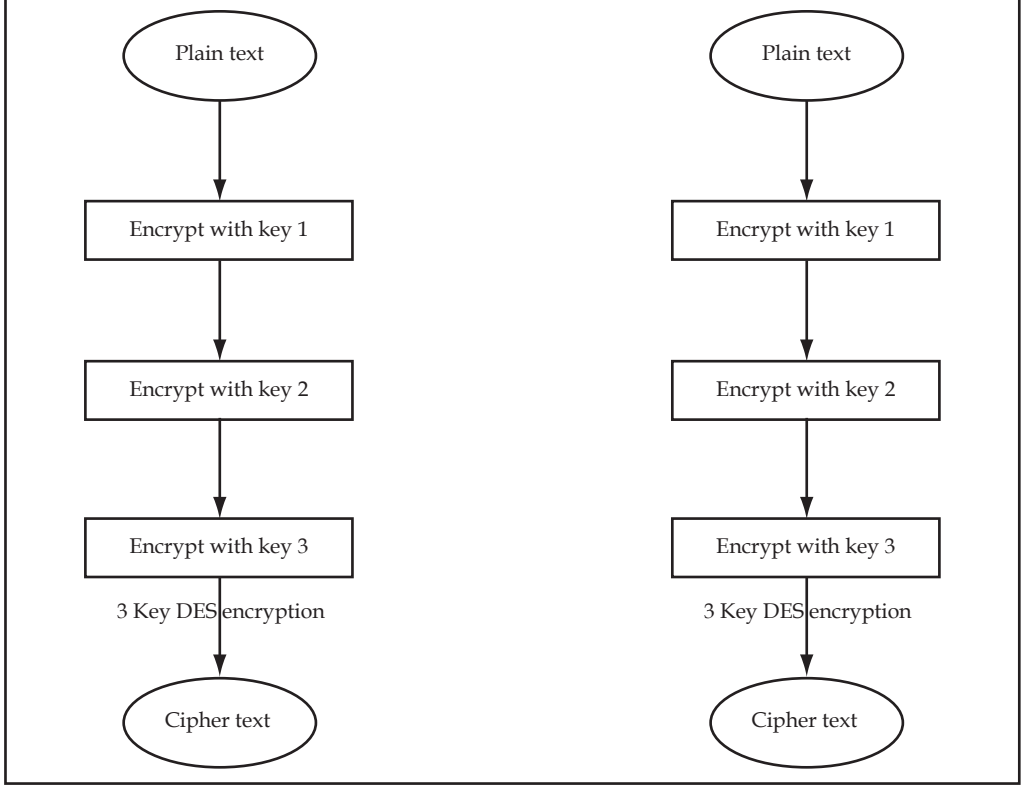
Notes

The DES algorithm is well known. However, since the key is only 56 bit long, there is some concern about security of data. A modification to DES is triple-DES and this improves the security aspects considerably.

12.6 Triple – DES

To strengthen DES, the DES encryption and decryption are done three times using either two keys or three keys as shown in Figure 12.4.

Figure 12.4: 2 and 3 key DES Encryption Algorithm



12.7 RSA System

RSA is an Internet encryption and authentication system that uses an algorithm developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. The RSA algorithm is the most commonly used encryption and authentication algorithm and is included as part of the Web browsers from Microsoft and Netscape. It's also part of Lotus Notes, Intuit's Quicken, and many other products. The encryption system is owned by RSA Security. The company licenses the algorithm technologies and also sells development kits. The technologies are part of existing or proposed Web, Internet, and computing standards.

How the RSA System Works?

The mathematical details of the algorithm used in obtaining the public and private keys are available at the RSA Web site. Briefly, the algorithm involves multiplying two large prime numbers (a prime number is a number divisible only by that number and 1) and through additional operations deriving a set of two numbers that constitutes the public key and another set that is the private key. Once the keys have been developed, the original prime numbers are

Notes

no longer important and can be discarded. Both the public and the private keys are needed for encryption /decryption but only the owner of a private key ever needs to know it. Using the RSA system, the private key never needs to be sent across the Internet.

The private key is used to decrypt text that has been encrypted with the public key. Thus, if I send you a message, I can find out your public key (but not your private key) from a central administrator and encrypt a message to you using your public key. When you receive it, you decrypt it with your private key. In addition to encrypting messages (which ensures privacy), you can authenticate yourself to me (so I know that it is really you who sent the message) by using your private key to encrypt a digital certificate. When I receive it, I can use your public key to decrypt it. A table might help us remember this.

To do this	Use whose	Key
Send an encrypted message	Use the receiver's	Public key
Send an encrypted signature	Use the sender's	Private key
Decrypt and encrypted message	Use the receiver's	private key
Decrypt an encrypted signature (and authenticate the sender)	Use the sender's	Public key

12.8 Comparison between Symmetric and Public Key Encryption

12.8.1 Symmetric Key Encryption

Symmetric cryptography involves a single, secret key, which both the message-sender and the message-recipient must have. It is used by the sender to encrypt the message, and by the recipient to decrypt it.

Symmetric cryptography provides a means of satisfying the requirement of message content security, because the content cannot be read without the secret key. There remains a risk exposure, however, because neither party can be sure that the other party has not exposed the secret key to a third party (whether accidentally or intentionally).

Symmetric cryptography can also be used to address the integrity and authentication requirements. The sender creates a summary of the message, or 'message authentication code (MAC)' encrypts it with the secret key, and sends that with the message. The recipient then re-create the MAC, decrypts the MAC that was sent, and compares the two. If they are identical, then the message that was received must have been identical with that which was sent.

A major difficulty with symmetric schemes is that the secret key has to be possessed by both parties, and hence has to be transmitted from whomever creates it to the other party. Moreover, if the key is compromised, all of the message transmission security measures are undermined. The steps taken to provide a secure mechanism for creating and passing on the secret key are referred to as 'key management'.

The technique does not adequately address the non-repudiation requirement, because both parties have the same secret key. Hence the other, and a claim by either party not to have sent a message is credible, because the other may have compromised the key expose each to the risk of fraudulent falsification of a message.

12.8.2 Public Key Cryptography (Encryption)

Whereas symmetric cryptography has existed, at least in primitive forms, for 2,000 years asymmetric approaches were only invented in the mid-1970s.

Public key cryptography involves two related keys, referred to as a 'key-pair', one of which only the owner knows (the 'private key') and the other which anyone can know (the 'public key').

Notes

The advantages of asymmetric encryption are:

Only one party needs to know the private key; and knowledge of the public key by a third party does not compromise the security of message transmissions.

The crack a mere 40 or 56 bit asymmetric key would be trivially simple, because there are far fewer of keys available (or, expressed more technically, the 'key-space' is relatively 'sparse'). It is currently conventional to regard a 1024-bit asymmetric key-length as being necessary to provide security. Because of the much greater key-length, encryption and decryption require mush core processing power, or, for a given processor, significantly more processing time. Messages are sent in large volumes; so the rsulting delays are of considerable consequence.



Task Discuss the use of Triple-DES for security purpose.

12.9 Digital Signature

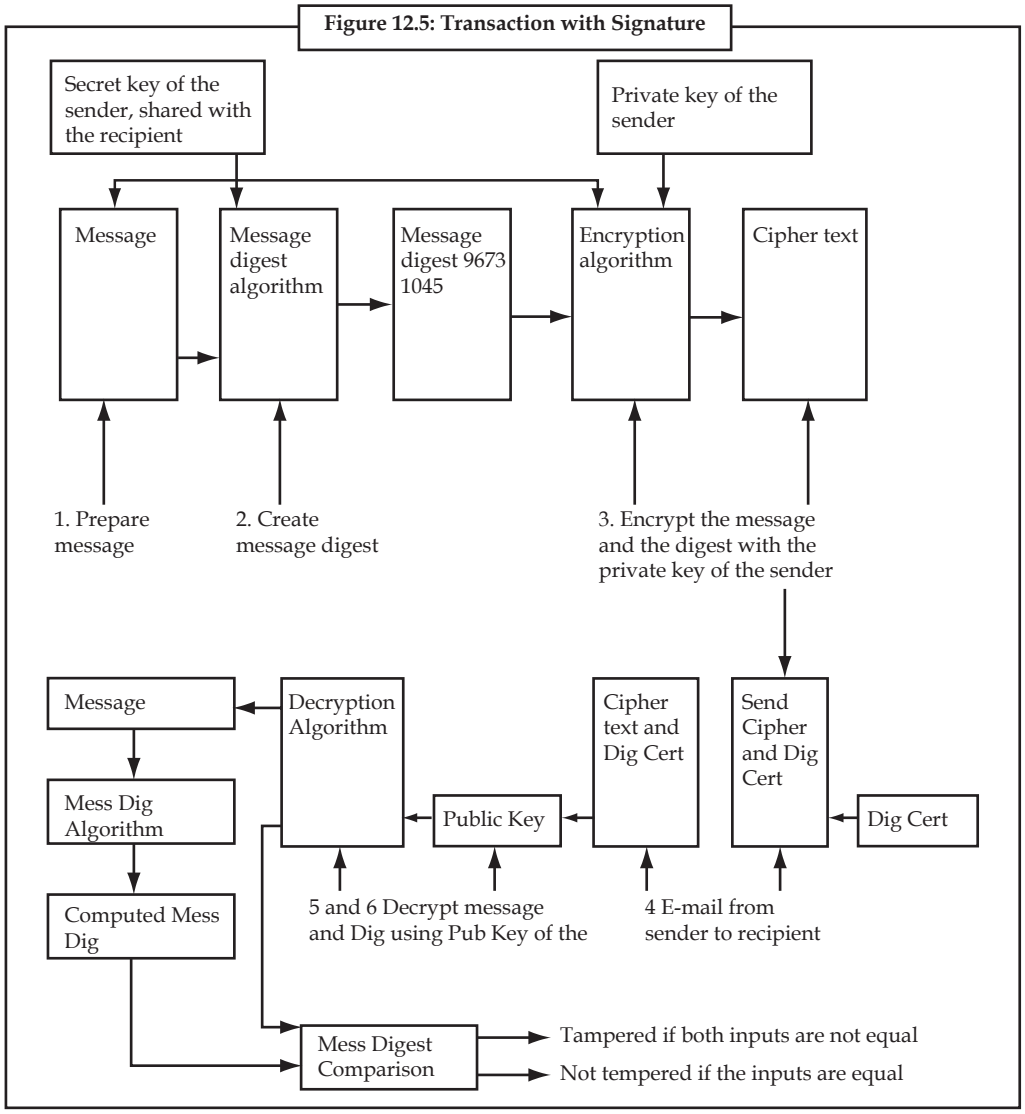
Like the conventional signature, the digital signature assures all concerned that the contents of the electronic messages are authentic, are really sent by the sender on the date and time recorded. All these functions can be performed using the public-key encryption techniques and the message digest techniques. As the message exchange and electronic commerce applications grow, the importance of digital signatures will increase. Let us take the case that you ordered a few dresses from an electronic store. The seller wants to make sure that your order is genuine and has come from you, the information in the order is accurate and has been modified on the network, and finally you will not disown the order.

12.9.1 Signing Process

1. Prepare the message. All the mail and messaging software including messaging programs like Microsoft exchange have all the needed software for handling digital signatures.
2. Create a message digest for the message using the secret key, which the sender is sharing with the recipient.
3. Encrypt the message and the digest with the private key of the sender. At this stage the document is signed as the message is authenticated with the private key of the sender. If required, send also the digital certificate of the sender, as it contains the public key of the sender. The sender should not encrypt this digital certificate, so as top facilitate easy retrieval of the sender's public key by the recipient.
4. Send the cipher text and the digital certificate to the recipient
5. The recipient retrieves the public key of the sender using his/her private key.
6. The recipient decrypts the cipher text
7. Recipient runs the message digest algorithm on the message, using the secret key shared with the sender.
8. Compare the computed message digest with the received message digest. If they are the same, then the message reached intact. Otherwise the message was tampered.

Notes

Figure 12.5: Transaction with Signature



Each message produces a random message digest using the conversion formula. Private key is used to encrypt that digest to obtain digital signature. Or in other words encrypts message digest (private key is used for encryption) called digital signature.

$$(\text{hash function} \rightarrow \text{message digest}) \xrightarrow[\text{with sender privae key}]{\text{encrypt}} \text{Digital signature}$$

Verification of Digital Signature

Say person X is sending the message to person Y.

Steps

1. To send the message (X sends to Y)
   (a) Develop message digest for each message. Using hash function.

Notes

(b) Encrypt the digest using "X"-private key [digital signature]

(c) Combine the plain text (X's-message) with signature, and send it to person 'Y' through Internet.

2. To receive message (Y receive)

(a) Decrypt the 'digital signature' with 'X' public key

(b) Calculate the message digest using hash function. [person Y uses the same hash-functions as that of person X, which was agreed upon before hand]

(c) Compare the each message digest, calculated and decrypted.

(d) If both message digest's are same (one which is sent by X, and the one which is generated by Y) –then it is authentic – if not signature or message has been tempered.

12.9.2 Advantages of Digital Signature

Unauthorized person's can access to the public key of person 'X", but cannot have his (X) hash function, which makes the digital signature authentic.

Disadvantages: As the body of the message is sent as plain text, privacy is not maintained. To overcome this difficulty when privacy is important one could use symmetric algorithm for plain text.

12.10 Digital Certificate

Digital certificates, or certs, simplify the task of establishing whether a public key truly belongs to the purported owner. A certificate is a form of credential. Examples might to your driver's license, your passport, or your birth certificate. Each of these has some information on it identifying you and some authorization stating that someone else has confirmed your identity. Some certificates, such as your passport, are important enough confirmation or your identity that you would not want to lose them, lest someone use them to impersonate you.

A digital certificate is data that functions much like a physical certificate. A digital certificate is information included with a person's public key that helps others verify that a key is genuine or valid. Digital certificates are used to thwart attempts to substitute one person's key for another.

A digital certificate consists of three things:

A public key certificate information ("Identity" information about the user, such as name, user ID and so on). One or more digital signature (of the CA)

The purpose of the digital signature on a certificate is to state that the certificate information has been attested to by some other person or entity. The digital signature does not attest the authenticity of the certificate as a whole, it vouches only that the information, which the certifying authority has signed, goes along or is bound to the public key listed in the certificate.

Basic aim to conduct secure and safe electronic transaction. Asymmetric cryptography allows a merchants distribute his (merchants) public key to all his correspondents, may be e-mail, or server, while keeping the private key secure (confined to himself only). But these key pairs can be generated by any one, third person may generate a pair of key and send that public key to the merchants correspondent, claiming that it has come from the merchant. This allows the third person or party to forge the message in the name of merchant. This is where a "**certificate authority**" comes into existence.

12.11 Certificate Authority

Notes

The certifying authority is an digital entity that binds the identity of a person to his public key. The certifying authority certifies that a person is the holder of a valid key pair and that person's identity has been authenticated by the certifying authority or its agents. The certifying authority thus performs functions that are quasi-governmental and by their very nature require a high amount of trust and security.

A certifying authority creates the digital certificate and digitally signs it using its own private key. When any third person wishes to verify the authenticity of a subscriber's certificate, he uses the CA's public key. The certifying authority thus validates the certificate and establishes a trust model for the third party into a transaction with the subscriber.

Digital certificate is defined as a method to verify (ex. Public Key's) electronically for authenticity.

A certificate authority will accept merchant public key, along with some proof of the identity of the merchant who sends it. Others (correspondents) can request fro verification of merchant's public key from the certificate authority.

Contents of ONES Digital Certificate

It includes:

1. Holder's name, organization, address.
2. The name of certificate authority
3. Public key of the holders for cryptographic use.
4. Time limit, these certificate are issued for 6 months to a year long
5. Class of certificate
6. Digital certificate identification number

Class: Based on degree of verification

Class 1:	Easiest to obtain, it involves the fewest checks on the user's back-round. (only the name of e-mail address are verified
Class 2:	I includes user's driver's license. Social security number & date of birth along with the other (class 1)
Class 3:	In addition to class 2 checks, user's credit card check is added.
Class 4:	In addition to class 3 checks, user's position within the organization is added.

Higher the class, higher the degree of verification and hence higher the fee payable to commercial or government certificate authorities. Certificate Revocation List (CRL) is maintained by certificate authority. So that the user know which certificate are no longer valid. The CRL doesn't include expired certificate, because each certificate has a built in expiration. Certificate lost may be revoked.

One encryption system is not ideal for all situations. One can use more than one encryption method. Table below shows few algorithms for encryption used by PGP (Pretty Good Privacy).

Function	Algorithms used	Process
Message encryption	IDEA, RSA	Use IDEA with one time session key generated by sender to encrypt message Encrypt session key with RSA using recipient's public key
Digital signature	MD5, RSA	Generate hash code of message with MD5 Encrypt message digest with RSA using sender's private key.

Notes

12.12 Enterprise Authentication using Digital Certificates

When one connects to a secure web server such as https://www.Amazom.com and request that server to authenticate itself, it has to so complex process involving public keys, private keys and a digital certificate (also known as electronic credentials or digital IDS). They allow verification of the claim that a given public key does in fact belong to a given individual or entity. In other words the digital certificate tells you that an independent third party has agreed that the server belongs to the company it claims to belong to. A valid certificate means that you can have confidence that you are sending information to the right place.

12.13 Summary

Encryption is essentially the process of encoding – or hiding – the information you send across the internet in a way that it can only be read by the person or website it is meant for. There are various ways this is handled on the net.

There are two primary approaches to encryption: symmetric and public-key. Symmetric encryption is the most common type of encryption and uses the same key for encoding and decoding data. This key is known as a session key. Public-key encryption uses two different keys, a public key and a private key. One key encodes the message and the other decodes it. The public key is widely distributed while the private key is secret.

12.14 Keywords

Decryption: Decryption is the reverse process of converting encoded data to its original un-encoded form, plaintext.

Digital signatures: A message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender signed it and that the message has not been tampered with. This is used to ensure authenticity.

Encryption: Encryption is a process of coding information which could either be a file or mail message in into cipher text a form unreadable without a decoding key in order to prevent anyone except the intended recipient from reading that data.

12.15 Self Assessment

Choose the appropriate answers:

1. SSL stands for
   (a) Secure Service Layer
   (b) Secure Socket Layer
   (c) Source Service Layer
   (d) Secure Service Link
2. MAC stands for
   (a) Message Authentication Code
   (b) Message Authentication Course
   (c) Message Authorization Code
   (d) Message Activity Code

Notes

Fill in the blanks:

3. A digital certificate is data that functions much like a ..... .
4. A certifying authority creates the digital certificate and digitally signs it using its own ..... .
5. ..... is defined as a method to verify (ex. Public Key's) electronically for authenticity.
6. A key in cryptography is a long sequence of bits used by ..... algorithms.
7. One key encodes the message and the other ..... it.
8. Encryption information helps prevent it by ..... .
9. The number of keys each algorithm can support depends on the number of bits in the ..... .
10. ..... schemes are generally categorized as being either stream ciphers or block ciphers.

12.16 Review Questions

1. Explain the concept of digital encryption standards
2. Write short note on RSA system
3. Briefly explain digital signature in details
4. What do you mean by digital certificate?
5. What do you understand by secret key cryptography? Explain with the help of suitable diagram.
6. Distinguish between DES and Triple-DES techniques.
7. "Symmetric cryptography can also be used to address the integrity and authentication requirements." Explain
8. Explain the process of verification of digital signature.
9. "Stream ciphers operate on a single bit at a time and implement some form of feedback mechanism so that the key is constantly changing." Discuss
10. Write short note on "Cipher Block Chaining".

Answers: Self Assessment

1.	(b)	2.	(a)	3.	physical certificate
4.	private key	5.	Digital certificate	6.	encryption/decryption
7.	decodes	8.	unauthorized user	9.	key
10.	Secret key cryptography				

Notes

12.17 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Published by Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 13: Case Study: Linux

Notes

CONTENTS

Objectives

Introduction

13.1 Design Principles

13.2 Kernel Modules

13.2.1 Linux Kernel Modules

13.2.2 Life Cycle of Linux Kernel Module

13.2.3 Unloading Modules

13.3 Process Management

13.4 Process Scheduling

13.5 Memory Management

13.5.1 Demand Paging

13.5.2 Swapping

13.5.3 Shared Virtual Memory

13.5.4 Physical and Virtual Addressing Modes

13.5.5 Access Control

13.5.6 Caches

13.5.7 Linux Page Tables

13.5.8 Page Allocation and Deallocation

13.5.9 Memory Mapping

13.5.10 Demand Paging

13.5.11 The Linux Page Cache

13.5.12 Swapping Out and Discarding Pages

13.5.13 Reducing the Size of the Page and Buffer Caches

13.5.14 Swapping Out System V Shared Memory Pages

13.5.15 Swapping Pages In

13.6 File Systems

13.7 Input & Output

13.8 Inter-process Communication

13.8.1 Signals

13.8.2 Pipes

13.8.3 System V IPC Mechanisms

13.8.4 Message Queues

Contd....

Notes

13.8.5 Semaphores
13.8.6 Shared Memory
13.9 Network Structure
13.9.1 An Overview of TCP/IP Networking
13.9.2 The Linux TCP/IP Networking Layers
13.9.3 The BSD Socket Interface
13.9.4 The INET Socket Layer
13.9.5 The IP Layer
13.9.6 The Address Resolution Protocol (ARP)
13.9.7 IP Routing
13.10 Security
13.11 Summary
13.12 Keywords
13.13 Review Questions
13.14 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe design principles and kernel modules
- Know process management and process scheduling
- Explain memory management and file systems
- Know Input & Output

Introduction

The objective of this unit is to introduce to the Linux operating system. Linux (often pronounced LIH-nuhks with a short "i") is a Unix-like operating system that was designed to provide personal computer users a free or very low-cost operating system comparable to traditional and usually more expensive Unix systems. Linux has a reputation as a very efficient and fast-performing system. Linux's kernel (the central part of the operating system) was developed by Linus Torvalds at the University of Helsinki in Finland. To complete the operating system, Torvalds and other team members made use of system components developed by members of the Free Software Foundation for the GNU Project.

Linux is a remarkably complete operating system, including a graphical user interface, an X Window System, TCP/IP, the Emacs editor, and other components usually found in a comprehensive Unix system. Although copyrights are held by various creators of Linux's components, Linux is distributed using the Free Software Foundation's copyleft stipulations that mean any modified version that is redistributed must in turn be freely available.

Unlike Windows and other proprietary systems, Linux is publicly open and extendible by contributors. Because it conforms to the Portable Operating System Interface standard user and programming interfaces, developers can write programs that can be ported to other operating systems. Linux comes in versions for all the major microprocessor platforms including the Intel, PowerPC, Sparc, and Alpha platforms. It's also available on IBM's S/390. Linux is distributed commercially by a number of companies.

Notes

Linux is sometimes suggested as a possible publicly-developed alternative to the desktop predominance of Microsoft Windows. Although Linux is popular among users already familiar with Unix, it remains far behind Windows in numbers of users. However, its use in the business enterprise is growing.

13.1 Design Principles

Linux is a modular Unix-like operating system. It derives much of its basic design from principles established in Unix during the 1970s and 1980s. Linux uses a monolithic kernel, the Linux kernel, which handles process control, networking, and peripheral and file system access. Device drivers are integrated directly with the kernel.

Much of Linux's higher-level functionality is provided by separate projects which interface with the kernel. The GNU userland is an important part of most Linux systems, providing the shell and Unix tools which carry out many basic operating system tasks. Atop these tools graphical user interfaces can be placed, usually running via the X Window System.

User Interface

Linux is coupled to a text-based Command Line Interface (CLI), though this is usually hidden on desktop computers by a Graphical User Interface (GUI). On small devices, input may be handled through controls on the device itself, and direct input to Linux might be hidden entirely.

The X Window System (X) is the predominant graphical subsystem used in Linux. X provides network transparency, enabling graphical output to be displayed on machines other than that which a program runs on. X runs locally for desktop machines.

Early GUIs for Linux were based on a stand-alone X window manager such as FVWM, Enlightenment, or Window Maker, and a suite of diverse applications running under it. The window manager provides a means to control the placement and appearance of individual application windows, and interacts with the X window system. Because the X window managers only manage the placement of windows, their decoration, and some inter-process communication, the look and feel of individual applications may vary widely, especially if they use different graphical user interface toolkits.

This model contrasts with that of platforms such as Mac OS, where a single toolkit provides support for GUI widgets and window decorations, manages window placement, and otherwise provides a consistent look and feel to the user. For this reason, the use of window managers by themselves declined with the rise of Linux desktop environments. They combine a window manager with a suite of standard applications that adhere to human interface guidelines. While a window manager is analogous to the Aqua user interface for Mac OS X, a desktop environment is analogous to Aqua with all of the default Mac OS X graphical applications and configuration utilities. KDE, which was announced in 1996, along with GNOME and Xfce which were both announced in 1997, are the most popular desktop environments.

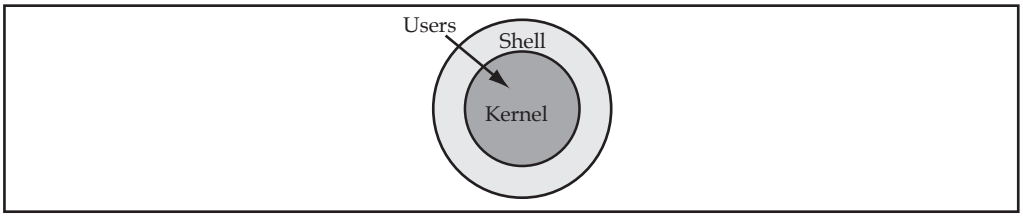
Linux systems usually provide a CLI of some sort through a shell, the traditional way of interacting with Unix systems. Even on modern desktop machines, some form of CLI is almost always accessible. Linux distributions specialized for servers may use the CLI as their only interface, and Linux machines can run without a monitor attached. Such "headless systems" may be controlled by command line via a protocol such as SSH or telnet.

Most low-level Linux components, including the GNU Userland, use the CLI exclusively. The CLI is particularly suited for automation of repetitive or delayed tasks, and provides very simple inter-process communication. Graphical terminal emulator programs can be used to access the CLI from a Linux desktop.

Notes

Shell Penal

The command interpreter is the interface between the user and the operating system, hence its name "shell".



The shell therefore acts as an intermediary between the operating system and the user using command lines entered by the latter. Its role consists of reading the command line, interpreting its meaning, executing the command, and then returning the result via the outputs.

The shell is an executable file responsible for interpreting commands, transmitting them to the system, and returning the result. There are several shells, the most common being sh (called the "Bourne shell"), bash ("Bourne again shell"), csh ("C Shell"), Tcsh ("Tenex C shell"), ksh ("Korn shell"),and zsh ("Zero shell"). Their name generally matches the name of the executable.

Each user has a default shell, which will be launched when a command prompt is opened. The default shell is specified in the configuration file /etc/passwd in the last field of the line corresponding to the user. It is possible to change the shell during a session simply by executing the corresponding executable file, for example:

/bin/bash

Command Prompt Window (Prompt)

The shell is initialized by reading its overall configuration (in a file of the directory /etc/), followed by reading the user's own configuration (in a hidden file the name of which starts with a dot, located in the basic user directory, i.e. /home/user_name/.configuration_file). Then, a command prompt window or prompt is displayed as follows:

machine:/directory/current$

By default, for most shells, the prompt consists of the name of the machine, followed by a colon (:), the current directory, then a character indicating the type of user connected:

1. "$" specifies a normal user
2. "#" specifies the administrator, called "root"

Command Line Concept

A command line is a character string representing a command corresponding to an executable system file or shell command along with optional arguments (parameters):

ls -al /home/jf/

In the above command, ls is the name of the command, -al and /home/jean-francois/ are arguments. Arguments beginning with - are called options. Generally, for each command there are a certain number of options which can be detailed by entering one of the following commands:

command --help

command -?

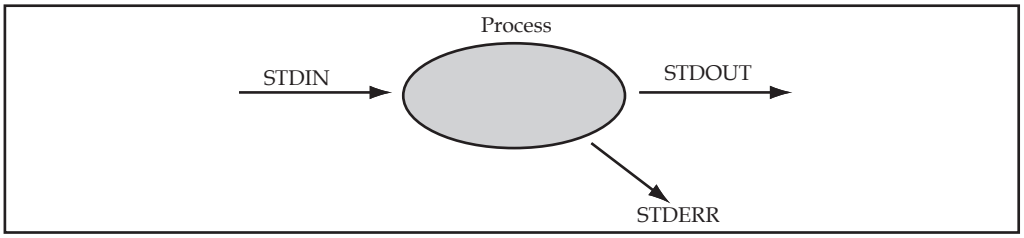
man command

Notes

Standard Input-output

Once a command is run, a process is created. This process then opens three flows:

1. ***stdin***, called the standard input, where the process will read the input data. By default stdin refers to the keyboard; STDIN is identified by the number 0;
2. ***stdout***, called standard output, where the process will write the output data. By default, stdin refers to the screen; STDOUT is identified by the number 1;
3. ***stderr***, called standard error, where the process will write error messages. By default, stderr refers to the screen. STDERR is identified by the number 2;



By default, whenever a program is run data is read from the keyboard and the program sends its output and errors to the screen. However, it is also possible to read data from any input device, even a file, and send the output to a display device, a file, etc.

Redirections

Like any Unix type system, Linux has mechanisms which make it possible to redirect the standard input-output to files.

So, using the ">" character makes it possible to redirect the standard output of a command on the left to the file located on the right:

ls -al /home/jf/ > toto.txt

echo "Toto" > /etc/myconfigurationfile

The following command is equivalent to a copy of the files:

cat toto > toto2

The purpose of the ">" redirection is to create a new file. So, if a file with the same name already exists it will be deleted. The following command simply creates an empty file:

> file

Using the double character ">>" makes it possible to add the standard output to the file, i.e. add the output after the file without deleting it.

In the same way, the "<" character indicates a redirection of the standard input. The following command sends the content of the toto.txt file to the input of the command cat, the only purpose of which is to display the content on the standard output (example not useful, but instructive):

cat < toto.txt

Finally, using the "<<" redirection makes it possible to read on the standard input, until the string located to the right is found. In the following example, the standard input will be read until the word STOP is found, and then the result will be displayed:

cat << STOP

Notes

Communication Pipes

Pipes are a communication mechanism specific to all UNIX systems. A pipe, symbolised by a vertical bar ("|" character), makes it possible to assign the standard output of one command to the standard input of another, like a pipe enabling communication between the standard input of one command with the standard output of another one.

In the following example, the standard output of the command ls -al is sent to the program sort, which is responsible for sorting the results in alphabetical order:

ls -al | sort

This makes it possible to connect a certain number of commands through successive pipes. In the example below, the command displays all the files in the current directory, selects the lines containing the word "zip" (using the grep command), and counts the total number of lines:

ls -l | grep zip | wc -l

13.2 Kernel Modules

The Linux kernel is a Unix-like operating system kernel. It is the namesake of the Linux family of operating systems. Released under the GNU General Public License (GPL) and developed by contributors worldwide, Linux is one of the most prominent examples of free software/open source.

The Linux Kernel was initially conceived and assembled by Linus Torvalds in 1991. Early on, the Minix community contributed code and ideas to the Linux kernel. At the time, the GNU Project had created many of the components required for a free software operating system, but its own kernel, GNU Hurd, was incomplete and unavailable. The BSD operating system had not yet freed itself from legal encumbrances. This meant that despite the limited functionality of the early versions, Linux rapidly accumulated developers and users who adopted code from those projects for use with the new operating system. Today the Linux kernel has received contributions from thousands of programmers.

Linux operates in two modes - the Kernel mode (kernel space) and the User mode (user space). The kernel works in the highest level (also called supervisor mode) where it has all the authority, while the applications work in the lowest level where direct access to hardware and memory are prohibited. Keeping in line with the traditional Unix philosophy, Linux transfers the execution from user space to the kernel space through system calls and the hardware interrupts. The Kernel code executing the system call works in the context of the process, which invokes the system call. As it operates on behalf of the calling process, it can access the data in the processes address space. The kernel code that handles interrupts, works to the processes and related to any particular process.

13.2.1 Linux Kernel Modules

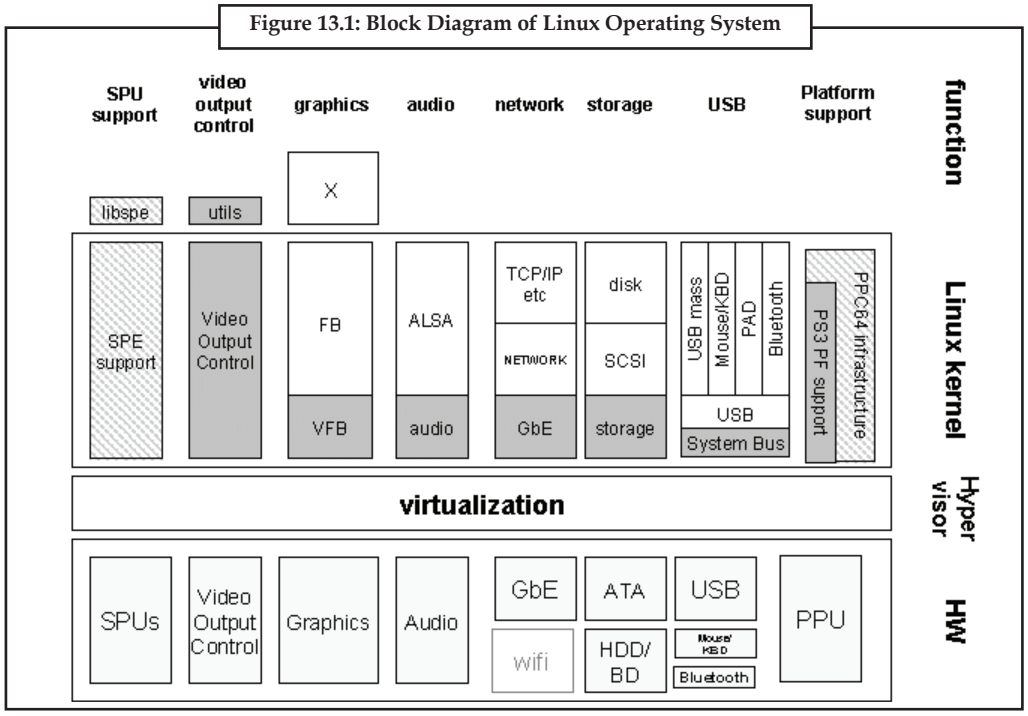
The Linux kernel is a monolithic kernel i.e. it is one single large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative to this is the micro kernel structure where the functional pieces of the kernel are broken out into units with strict communication mechanism between them. This makes adding new components into the kernel, via the configuration process, rather time consuming. The best and the robust alternative is the ability to dynamically load and unload the components of the operating system using Linux Kernel Modules.

The Linux kernel modules are piece of codes, which can be dynamically linked to the kernel (according to the need), even after the system booting. They can be unlinked from the kernel and removed when they are no longer needed. Mostly the Linux kernel modules are used for

Notes

device drivers or pseudo-device drivers such as network drivers or file system. When a Linux kernel module is loaded, it becomes a part of the Linux kernel as the normal kernel code and functionality and it posses the same rights and responsibilities as the kernel code.

Figure 13.1: Block Diagram of Linux Operating System



13.2.2 Life Cycle of Linux Kernel Module

The life cycle of a module starts with the init_module(). The task of init_module is to prepare the module for later invocation. The module is registered to the kernel and attaches its data-structures and functionality to the kernel. The kernel-defined external functions are also resolved. The life cycle of the module ends with cleanup_module(). It `unregisters' the module functionality from the kernel.

Simple Module Program

Let us now program a simple module to review its life cycle. The init_module is called when the module is inserted into the kernel where as the cleanup_module is called just before removing it from the kernel. In the following program, the init_module and the cleanup_module functions are demonstrated.

```
/* Simple Linux kernel module Feb'2001
#include
#include
#if CONFIG_MODVERSIONS==1
#define MODVERSINS
#include
#endif
/ initialise the module /
int init_module()
```

Notes

```
{
printk("init_module invokedn");
printk("the message is printed from the kernel spacen");
/ if the non zero value is returned, then it means that the init_module failed
and the kernel module can't be loaded /
return 0;
}
/ cleanup / end of module life cycle */
void cleanup_module()
{
printk("cleanup_module invokedn");
printk("module is now going to be
unloaded from the kerneln");
}
```

Compile the above program using the following:

```
#gcc -Wall -DMODULE -D__KERNEL__ -DLINUX -O -c simpelModule.c
Run the compiled module using the following:
#insmod simpleModule.o
```

Remember, you have to run the above command from the Linux shell at raw console (not from the console in Xwindows environment) at root login.

Now check the status of the module using

```
#lsmod
```

Then remove the module using

```
#rmmod simpleModule
```

If you have not seen any of the module-initiated console printing (implemented using `printk') about the status of the module, use the following command to see the kernel messages. dmesg | less In the above, commands insmod, lsmod and rmmod are used to load and unload modules to the Linux kernel. The details are discussed in the following section.

Loading modules insmod loads the `loadable kernel modules' in the running kernel. insmod tries to link a module into the running kernel by resolving all the symbols from the kernel's exported `symbol table'.

Let's now discuss the demand loading of the module by the kernel, dynamically. When the Linux kernel discovers the need for a module, the kernel requests to the kernel daemon (kerneld) to load the appropriate module. To illustrate this with an example, lets mount a NTFS partition in the Linux system. If the NTFS filesystem support is not statically implemented in the kernel (but compiled as a module), the kernel daemon will search for the appropriate module and load it from the repository. Then the partition is mounted for the use.

Lets go deep into the action of the kernel daemon (kerneld). The kerneld is the normal user process having exclusive superuser privileges. At the time of booting, kerneld opens the IPC channel to the kernel and uses it for transferring messages (request for loading modules), to and from the kernel. While loading the module, the kerneld calls modprobe and insmod to load the required module. The insmod utility should able to access the requested module. The demand loadable kernel modules are usually located at/lib/module/directory as the object files linked as relocatable images.

Notes

Let us revisit the working of the insmod to get a clear picture of the module loading operation. The insmod depends on some critical system calls to load the module to the kernel. It uses the sys_create_module to allocate kernel memory to hold module. It uses get_kernel_syms system call to get the kernel symbol table in order to link the module. It then calls the sys_init_module system call to copy the relocatable object code of the module to the kernel space. And soon after this, insmod calls the initialization function of the concerned module i.e. init_module. All of these system calls can be found in kernel/module.c.

13.2.3 Unloading Modules

The modules can be unloaded using rmmod command. While removing modules, rmmod ensures the restriction that the modules are not in use and they are not referred by any other module or the part of the kernel. The demand loaded modules (i.e. the modules loaded by kerneld) are automatically removed from the system by `kerneld' when they are no longer used. Every time it's idle, timer expires and kerneld makes a system call requesting for all the demand loaded kernels, which are not busy to be removed. The modules, whose visited flags are cleared and marked as AUTOCLEAN, are `unloaded'.

Assuming that the module can be unloaded, the cleanup_module function of the concerned module is called to freeup the kernel resources it has allocated. After the successful execution of the cleanup_module, the module datastructure is marked DELETED and it is unlinked from the kernel and unlisted from the list of kernel modules. The reference list of the modules on which it (module removed) is dependent is modified and dependency is released. The kernel memory allocated to the concerned module is deallocated and returned to the kernel memory spool.

Version Dependency of modules Version dependency of the module is one of the trickiest parts of the Linux Kernel Module programming. Typically, the modules are required to be compiled for each version of the kernel. Each module defines a symbol called kernel_version, which insmod matches against the version number of the current kernel. The kernel 2.2.x/2.4.x define the symbol in. Hence if the module is made up of multiple source files, should be included in only one of the source files.

Though typically, modules should be recompiled for each kernel version, it is not always possible to recompile module when it is run on as a commercial module distributed in binary form. Kernel developers have provided a flexible way to deal with the version problem. The idea is that a module is incompatible with a different kernel version only if the software interface offered by the kernel is changed. The software interface is represented by the function prototype and the exact definition of all the data structures involved in the function call. The CRC algorithm can be used to map all the information about software interface to the single 32bit number. The issue of version dependency is handled by using the name of the each symbol exported.

13.3 Process Management

Any application that runs on a Linux system is assigned a process ID or PID. This is a numerical representation of the instance of the application on the system. In most situations this information is only relevant to the system administrator who may have to debug or terminate processes by referencing the PID. Process Management is the series of tasks a System Administrator completes to monitor, manage, and maintain instances of running applications.

Multitasking

Process Management beings with an understanding concept of Multitasking. Linux is what is referred to as a preemptive multitasking operating system. Preemptive multitasking systems rely on a scheduler. The function of the scheduler is to control the process that is currently using the CPU. In contrast, symmetric multitasking systems such as Windows 3.1 relied on each running process to voluntary relinquish control of the processor. If an application in this system hung or

Notes

stalled, the entire computer system stalled. By making use of an additional component to pre-empt each process when its "turn" is up, stalled programs do not affect the overall flow of the operating system.

Each "turn" is called a time slice, and each time slice is only a fraction of a second long. It is this rapid switching from process to process that allows a computer to "appear' to be doing two things at once, in much the same way a movie "appears" to be a continuous picture.

Types of Processes

There are generally two types of processes that run on Linux. Interactive processes are those processes that are invoked by a user and can interact with the user. VI is an example of an interactive process. Interactive processes can be classified into foreground and background processes. The foreground process is the process that you are currently interacting with, and is using the terminal as its stdin (standard input) and stdout (standard output). A background process is not interacting with the user and can be in one of two states - paused or running.

The following exercise will illustrate foreground and background processes.

1. Logon as root.
2. Run [cd \]
3. Run [vi]
4. Press [ctrl + z]. This will pause vi
5. Type [jobs]
6. Notice vi is running in the background
7. Type [fg %1]. This will bring the first background process to the foreground.
8. Close vi.

The second general type of process that runs on Linux is a system process or Daemon (day-mon). Daemon is the term used to refer to process' that are running on the computer and provide services but do not interact with the console. Most server software is implemented as a daemon. Apache, Samba, and inn are all examples of daemons.

Any process can become a daemon as long as it is run in the background, and does not interact with the user. A simple example of this can be achieved using the [ls -R] command. This will list all subdirectories on the computer, and is similar to the [dir /s] command on Windows. This command can be set to run in the background by typing [ls -R &], and although technically you have control over the shell prompt, you will be able to do little work as the screen displays the output of the process that you have running in the background. You will also notice that the standard pause (ctrl+z) and kill (ctrl+c) commands do little to help you.

Input, Output, and Error Redirection

In order to work with processes on Linux, you must understand the role of redirection. A process that is set to run in the background will continue to display information on the console until it is told otherwise. Changing where a program sends its output is known as redirection.

Standard Output, Standard Input, and Standard Error are the three items that can be redirected. These items are represented by the files /dev/stdin, /dev/stdout, and /dev/stderr. All interactive process' are programmed to write to these three files for errors, output and input. By default all three of these files are symlinked to through the /proc directory to the terminal the user is currently using. In the case of an interactive user, the flow is as follows.

```
/dev/stdout -> /proc/self/fd/1 -> /dev/tty1
```

Notes

The /proc directory is a virtual directory that contains information used to construct the users interactive environment and is thus different for each user. In this case you can see that it provides a layer of abstraction and acts as a link between the stdout file, and the file representing the users console device.

Redirection involves using special characters to tell the shell to send input, output, and errors to files other than the defaults. The following table lists these symbols.

Symbol, Meaning

```
> Redirect stdout
>> Redirect and append stdout
2> Redirect stderr
2>> Redirect and append stderr
< Use in place of stdin
```

In order for a background process to truly run 100% in the background, you must suppress display of all errors. The following example uses redirection to produce a file listing of the entire file system as a background process.

Managing Running Processes

The [ps] command is the command used to manage running processes and can be used for many things including viewing the status of your computer and getting a quick idea of how well the computer is performing.

Here are some common ps commands.

Command, Usefulness

```
ps View current interactive processes on this terminal.
ps -a All current processes on this terminal, by all users.
ps -x All processes not assigned to a terminal (daemons).
ps -aux Output all process running and include resource utilization information.
```

The man page for ps contains extensive documentation on how to modify and interpret the output of ps.

Top is a utility that can be used to display a live dataset of the currently running processes. Activate it by typing [top].

Killing Stalled Processes

Processes that have stalled or frozen can sometimes cause problems. One of the jobs of a Linux administrator is to identify and resolve stalled processes. The clues that a process has stalled can range from an unresponsive GUI to a noted decrease in system performance. Either way, once you have identified that a processes has stalled you can terminate that process using the [kill] command. The syntax is fairly simple. You kill process by referencing its process ID or PID. This information can be seen in the output of just about any iteration of the ps command. To kill a process you pass a signal to that process. The default signal tells the process to perform a clean shutdown. Here are a few examples.

To kill a single process:

```
ps
PID TT STAT TIME COMMAND
9408 p9 S 0:00 ue temp2.xdh
```

Notes

```
9450 pa S 0:01 -Tcsh (Tcsh)
9501 pa T 0:00 less csh.1
9503 pa R 0:00 ps
kill 9501
```

This kills process number 9501. Notice that the ps command which is entered to check on the process ID's has the latest process number.

To kill a process that refuses to die:

```
kill -9 9352
```

This makes a certain kill of process number 9352.

To kill a background job:

```
jobs
[1] + Running xterm -g 70x55
kill %1
[1] Done xterm -g 70x55
```

This kills job number 1 (one); the only job that is currently running.

To kill more than one process:

kill 8939 9543

This kills processes 8939 and 9543.

It is important to note that the kill command does not perform only negative actions. It can also be used to restart processes, and to keep processes running, even after a logout.

Understanding the init Processes

[init] is the most important process in Linux. All processes are derived from the init process and can trace their roots back to init. [init] always has a PID of 1 and is owned by root. The [init] process is used to start other processes and must be running for the system to operate. In a later article we will examine the boot process. Here [init] will be discussed in much more detail.

Parent Processes

Every process has a parent process, with the exception of [init], whose parent process ID (PPID) is 0. It is important to understand the effects of killing a process, especially if that process has spawned child processes.

```
Sidebar…for all you parent process's out there
```

When you kill a parent process, without first killing the child procesi, then you have created orphans. Orphans will generally cause performance decreases as they are taking up resources, but not doing anything. Additionally, they usually do not self terminate, as that task can often be left up the parent. To ensure that you do not create orphans, make sure that no child processes exist when you kill the parent. This can be determined by reviewing the output of the -l switch of the ps command. Compare the PID and PPID columns.

13.4 Process Scheduling

All processes run partially in user mode and partially in system mode. How these modes are supported by the underlying hardware differs but generally there is a secure mechanism for

Notes

getting from user mode into system mode and back again. User mode has far less privileges than system mode. Each time a process makes a system call it swaps from user mode to system mode and continues executing. At this point the kernel is executing on behalf of the process. In Linux, processes do not preempt the current, running process, they cannot stop it from running so that they can run. Each process decides to relinquish the CPU that it is running on when it has to wait for some system event. For example, a process may have to wait for a character to be read from a file. This waiting happens within the system call, in system mode; the process used a library function to open and read the file and it, in turn made system calls to read bytes from the open file. In this case the waiting process will be suspended and another, more deserving process will be chosen to run.

Processes are always making system calls and so may often need to wait. Even so, if a process executes until it waits then it still might use a disproportionate amount of CPU time and so Linux uses pre-emptive scheduling. In this scheme, each process is allowed to run for a small amount of time, 200ms, and, when this time has expired another process is selected to run and the original process is made to wait for a little while until it can run again. This small amount of time is known as a time-slice.

It is the scheduler that must select the most deserving process to run out of all of the runnable processes in the system.

A runnable process is one which is waiting only for a CPU to run on. Linux uses a reasonably simple priority based scheduling algorithm to choose between the current processes in the system. When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes task_struct data structure. It then restores the state of the new process (again this is processor specific) to run and gives control of the system to that process. For the scheduler to fairly allocate CPU time between the runnable processes in the system it keeps information in the task_struct for each process:

Policy: This is the scheduling policy that will be applied to this process. There are two types of Linux process, normal and real time. Real time processes have a higher priority than all of the other processes. If there is a real time process ready to run, it will always run first. Real time processes may have two types of policy, round robin and first in first out. In round robin scheduling, each runnable real time process is run in turn and in first in, first out scheduling each runnable process is run in the order that it is in on the run queue and that order is never changed.

Priority: This is the priority that the scheduler will give to this process. It is also the amount of time (in jiffies) that this process will run for when it is allowed to run. You can alter the priority of a process by means of system calls and the renice command.

rt_priority: Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.

Counter: This is the amount of time (in jiffies) that this process is allowed to run for. It is set to priority when the process is first run and is decremented each clock tick.

The scheduler is run from several places within the kernel. It is run after putting the current process onto a wait queue and it may also be run at the end of a system call, just before a process is returned to process mode from system mode. One reason that it might need to run is because the system timer has just set the current processes counter to zero. Each time the scheduler is run it does the following:

Kernel work: The scheduler runs the bottom half handlers and processes the scheduler task queue.

Notes

Current process: The current process must be processed before another process can be selected to run.

1. If the scheduling policy of the current processes is round robin then it is put onto the back of the run queue.
2. If the task is INTERRUPTIBLE and it has received a signal since the last time it was scheduled then its state becomes RUNNING.
3. If the current process has timed out, then its state becomes RUNNING.
4. If the current process is RUNNING then it will remain in that state.

Processes that were neither RUNNING nor INTERRUPTIBLE are removed from the run queue. This means that they will not be considered for running when the scheduler looks for the most deserving process to run.

Process selection: The scheduler looks through the processes on the run queue looking for the most deserving process to run. If there are any real time processes (those with a real time scheduling policy) then those will get a higher weighting than ordinary processes. The weight for a normal process is its counter but for a real time process it is counter plus 1000. This means that if there are any runnable real time processes in the system then these will always be run before any normal runnable processes. The current process, which has consumed some of its time-slice (its counter has been decremented) is at a disadvantage if there are other processes with equal priority in the system; that is as it should be. If several processes have the same priority, the one nearest the front of the run queue is chosen. The current process will get put onto the back of the run queue. In a balanced system with many processes of the same priority, each one will run in turn. This is known as Round Robin scheduling. However, as processes wait for resources, their run order tends to get moved around.

Swap processes: If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running. When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes task_struct data structure. Then, all of the machine state for the new process must be loaded. This is a system dependent operation, no CPUs do this in quite the same way but there is usually some hardware assistance for this act.

This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents.

If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated. Again, this action is architecture specific. Processors like the Alpha AXP, which use Translation Look-aside Tables or cached Page Table Entries, must flush those cached table entries that belonged to the previous process.

Scheduling in Multiprocessor Systems

Systems with multiple CPUs are reasonably rare in the Linux world but a lot of work has already gone into making Linux an SMP (Symmetric Multi-Processing) operating system. That is, one that is capable of evenly balancing work between the CPUs in the system. Nowhere is this balancing of work more apparent than in the scheduler.

Notes

In a multiprocessor system, hopefully, all of the processors are busily running processes. Each will run the scheduler separately as its current process exhausts its time-slice or has to wait for a system resource. The first thing to notice about an SMP system is that there is not just one idle process in the system. In a single processor system the idle process is the first task in the task vector, in an SMP system there is one idle process per CPU, and you could have more than one idle CPU. Additionally there is one current process per CPU, so SMP systems must keep track of the current and idle processes for each processor.

In an SMP system each process's task_struct contains the number of the processor that it is currently running on (processor) and its processor number of the last processor that it ran on (last_processor). There is no reason why a process should not run on a different CPU each time it is selected to run but Linux can restrict a process to one or more processors in the system using the processor_mask. If bit N is set, then this process can run on processor N. When the scheduler is choosing a new process to run it will not consider one that does not have the appropriate bit set for the current processor's number in its processor_mask. The scheduler also gives a slight advantage to a process that last ran on the current processor because there is often a performance overhead when moving a process to a different processor.

13.5 Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

Large Address Spaces: The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system.

Protection: Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping: Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation: The memory management subsystem allows each running process in the system a fair share of the physical memory of the system.

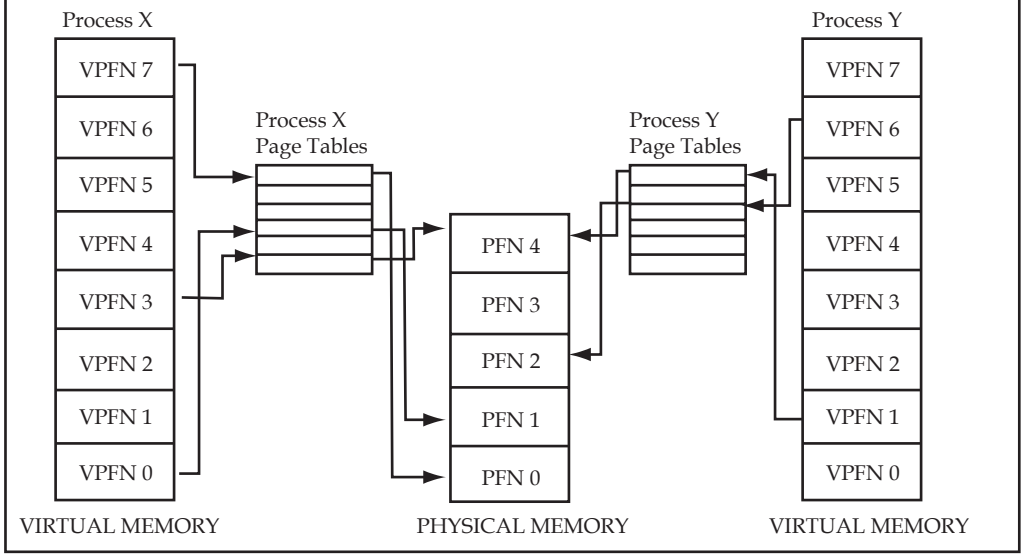
Shared Virtual Memory: Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes.

Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix TM System V shared memory IPC.

Before considering the methods that Linux uses to support virtual memory it is useful to consider an abstract model that is not cluttered by too much detail.

Notes

Figure 13.2: Abstract Model of Virtual to Physical Address Mapping



As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called pages. These pages are all the same size, they need not be but if they were not, the system would be very hard to administer. Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the page frame number (PFN).

In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses page tables.

In the Figure 13.2 shows the virtual address spaces of two processes, process X and process Y, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process X's virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process Y's virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

1. Valid flag. This indicates if this page table entry is valid,
2. The physical page frame number that this entry is describing,
3. Access control information. This describes how the page may be used. Can it be written to? Does it contain executable code?

Notes

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at Figures 13.2 and assuming a page size of 0x2000 bytes (which is decimal 8192) and an address of 0x2194 in process Y's virtual address space then the processor would translate that address into offset 0x194 into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However the processor delivers it, this is known as a page fault and the operating system is notified of the faulting virtual address and the reason for the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process Y's virtual page frame number 1 is mapped to physical page frame number 4 which starts at 0x8000 (4 x 0x2000). Adding in the 0x194 byte offset gives us a final physical address of 0x8194.

By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order. For example, in Figure 13.2 process X's virtual page frame number 0 is mapped to physical page frame number 1 whereas virtual page frame number 7 is mapped to physical page frame number 0 even though it is higher in virtual memory than virtual page frame number 0. This demonstrates an interesting byproduct of virtual memory; the pages of virtual memory do not have to be present in physical memory in any particular order.

13.5.1 Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. If the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

When a process attempts to access a virtual address that is not currently in memory the processor cannot find a page table entry for the virtual page referenced. For example, in Figure 13.2 there is no entry in process X's page table for virtual page frame number 2 and so if process X attempts to read from an address within virtual page frame number 2 the processor cannot translate the address into a physical one. At this point the processor notifies the operating system that a page fault has occurred.

If the faulting virtual address is invalid this means that the process has attempted to access a virtual address that it should not have. Maybe the application has gone wrong in some way, for example writing to random addresses in memory. In this case the operating system will terminate it, protecting the other processes in the system from this rogue process.

Notes

If the faulting virtual address was valid but the page that it refers to is not currently in memory, the operating system must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual page frame number is added to the processes page table. The process is then restarted at the machine instruction where the memory fault occurred. This time the virtual memory access is made, the processor can make the virtual to physical address translation and so the process continues to run.

Linux uses demand paging to load executable images into a processes virtual memory. Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as memory mapping. However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk. As the image executes, it generates page faults and Linux uses the processes memory map in order to determine which parts of the image to bring into memory for execution.

13.5.2 Swapping

If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a dirty page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

If the algorithm used to decide which pages to discard or swap (the swap algorithm is not efficient then a condition known as thrashing occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure 13.2 is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the working set. An efficient swap scheme would make sure that all processes have their working set in physical memory.

Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.

13.5.3 Shared Virtual Memory

Virtual memory makes it easy for several processes to share memory. All memory access are made via page tables and each process has its own separate page table. For two processes sharing a physical page of memory, its physical page frame number must appear in a page table entry in both of their page tables.

Figure 13.2 shows two processes that each share physical page frame number 4. For process X this is virtual page frame number 4 whereas for process Y this is virtual page frame number 6.

Notes

This illustrates an interesting point about sharing pages: the shared physical page does not have to exist at the same place in virtual memory for any or all of the processes sharing it.

13.5.4 Physical and Virtual Addressing Modes

It does not make much sense for the operating system itself to run in virtual memory. This would be a nightmare situation where the operating system must maintain page tables for itself. Most multi-purpose processors support the notion of a physical address mode as well as a virtual address mode. Physical addressing mode requires no page tables and the processor does not attempt to perform any address translations in this mode. The Linux kernel is linked to run in physical address space.

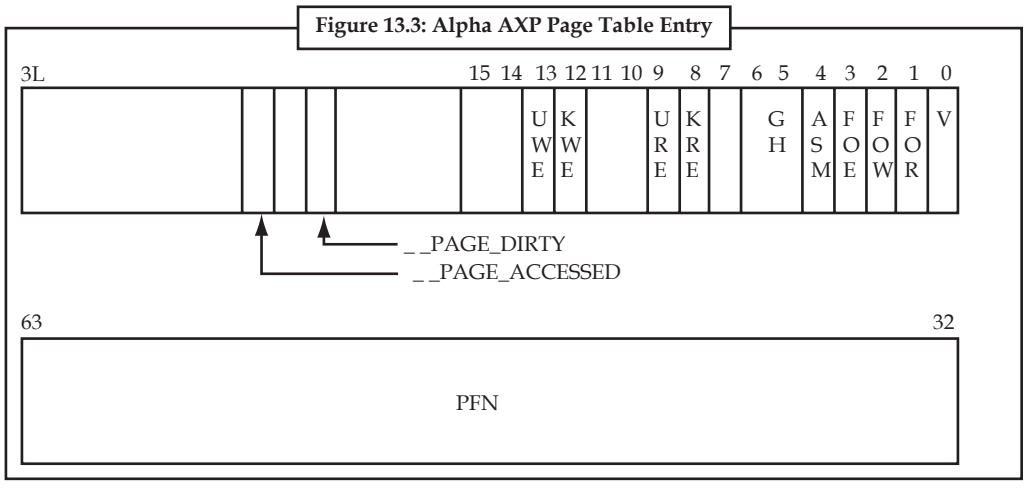
The Alpha AXP processor does not have a special physical addressing mode. Instead, it divides up the memory space into several areas and designates two of them as physically mapped addresses. This kernel address space is known as KSEG address space and it encompasses all addresses upwards from 0xfffffc0000000000. In order to execute from code linked in KSEG (by definition, kernel code) or access data there, the code must be executing in kernel mode. The Linux kernel on Alpha is linked to execute from address 0xfffffc0000310000.

13.5.5 Access Control

The page table entries also contain access control information. As the processor is already using the page table entry to map a processes virtual address to a physical one, it can easily use the access control information to check that the process is not accessing memory in a way that it should not.

There are many reasons why you would want to restrict access to areas of memory. Some memory, such as that containing executable code, is naturally read only memory; the operating system should not allow a process to write data over its executable code. By contrast, pages containing data can be written to but attempts to execute that memory as instructions should fail. Most processors have at least two modes of execution: kernel and user. You would not want kernel code executing by a user or kernel data structures to be accessible except when the processor is running in kernel mode.

Figure 13.3: Alpha AXP Page Table Entry



The access control information is held in the PTE and is processor specific; Figure 13.3 shows the PTE for Alpha AXP. The bit fields have the following meanings:

V: Valid, if set this PTE is valid,

Notes

FOE: "Fault on Execute", Whenever an attempt to execute instructions in this page occurs, the processor reports a page fault and passes control to the operating system,

FOW: "Fault on Write", as above but page fault on an attempt to write to this page,

FOR: "Fault on Read", as above but page fault on an attempt to read from this page,

ASM: Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer,

KRE: Code running in kernel mode can read this page,

URE: Code running in user mode can read this page,

GH: Granularity hint used when mapping an entire block with a single Translation Buffer entry rather than many,

KWE: Code running in kernel mode can write to this page,

UWE: Code running in user mode can write to this page,

page frame number: For PTEs with the V bit set, this field contains the physical Page Frame Number (page frame number) for this PTE. For invalid PTEs, if this field is not zero, it contains information about where the page is in the swap file.

The following two bits are defined and used by Linux:

_PAGE_DIRTY: if set, the page needs to be written out to the swap file,

_PAGE_ACCESSED: Used by Linux to mark a page as having been accessed.

13.5.6 Caches

If you were to implement a system using the above theoretical model then it would work, but not particularly efficiently. Both operating system and processor designers try hard to extract more performance from the system. Apart from making the processors, memory and so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches:

Buffer Cache

The buffer cache contains data buffers that are used by the block device drivers.

These buffers are of fixed sizes (for example 512 bytes) and contain blocks of information that have either been read from a block device or are being written to it. A block device is one that can only be accessed by reading and writing fixed sized blocks of data. All hard disks are block devices.

The buffer cache is indexed via the device identifier and the desired block number and is used to quickly find a block of data. Block devices are only ever accessed via the buffer cache. If data can be found in the buffer cache then it does not need to be read from the physical block device, for example a hard disk, and access to it is much faster.

Page Cache

This is used to speed up access to images and data on disk.

It is used to cache the logical contents of a file a page at a time and is accessed via the file and offset within the file. As pages are read into memory from disk, they are cached in the page cache.

Notes

Swap Cache

Only modified (or dirty) pages are saved in the swap file.

So long as these pages are not modified after they have been written to the swap file then the next time the page is swapped out there is no need to write it to the swap file as the page is already in the swap file. Instead the page can simply be discarded. In a heavily swapping system this saves many unnecessary and costly disk operations.

Hardware Caches

One commonly implemented hardware cache is in the processor; a cache of Page Table Entries. In this case, the processor does not always read the page table directly but instead caches translations for pages as it needs them. These are the Translation Look-aside Buffers and contain cached copies of the page table entries from one or more processes in the system.

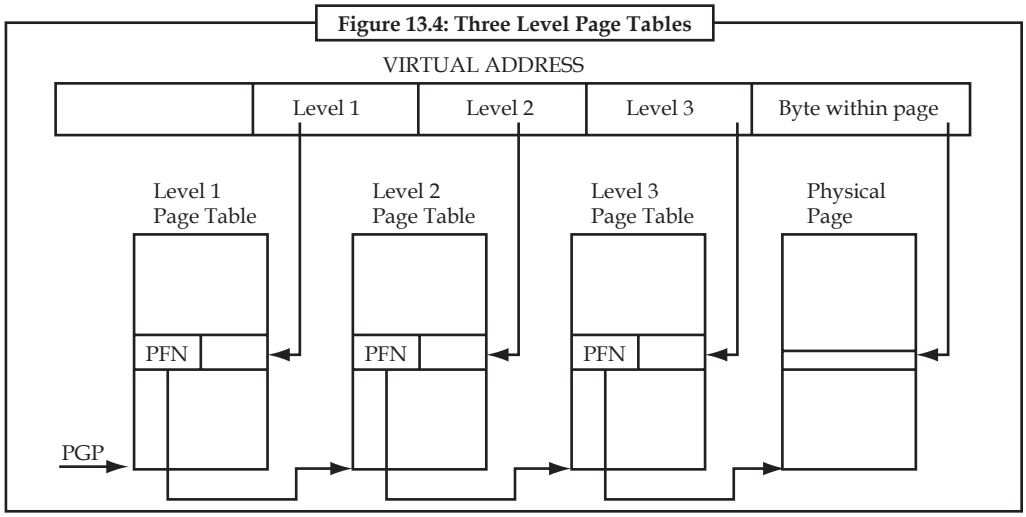
When the reference to the virtual address is made, the processor will attempt to find a matching TLB entry. If it finds one, it can directly translate the virtual address into a physical one and perform the correct operation on the data. If the processor cannot find a matching TLB entry then it must get the operating system to help. It does this by signaling the operating system that a TLB miss has occurred. A system specific mechanism is used to deliver that exception to the operating system code that can fix things up. The operating system generates a new TLB entry for the address mapping. When the exception has been cleared, the processor will make another attempt to translate the virtual address. This time it will work because there is now a valid entry in the TLB for that address.

The drawback of using caches, hardware or otherwise, is that in order to save effort Linux must use more time and space maintaining these caches and, if the caches become corrupted, the system will crash.

13.5.7 Linux Page Tables

Linux assumes that there are three levels of page tables. Each Page Table accessed contains the page frame number of the next level of Page Table. Figure 13.4 shows how a virtual address can be broken into a number of fields; each field providing an offset into a particular Page Table. To translate a virtual address into a physical one, the processor must take the contents of each level field, convert it into an offset into the physical page containing the Page Table and read the page frame number of the next level of Page Table. This is repeated three times until the page frame number of the physical page containing the virtual address is found. Now the final field in the virtual address, the byte offset, is used to find the data inside the page.

Figure 13.4: Three Level Page Tables



Notes

Each platform that Linux runs on must provide translation macros that allow the kernel to traverse the page tables for a particular process. This way, the kernel does not need to know the format of the page table entries or how they are arranged.

This is so successful that Linux uses the same page table manipulation code for the Alpha processor, which has three levels of page tables, and for Intel x86 processors, which have two levels of page tables.

13.5.8 Page Allocation and Deallocation

There are many demands on the physical pages in the system. For example, when an image is loaded into memory the operating system needs to allocate pages. These will be freed when the image has finished executing and is unloaded. Another use for physical pages is to hold kernel specific data structures such as the page tables themselves. The mechanisms and data structures used for page allocation and deallocation are perhaps the most critical in maintaining the efficiency of the virtual memory subsystem.

All of the physical pages in the system are described by the mem_map data structure which is a list of mem_map_t structures which is initialized at boot time. Each mem_map_t describes a single physical page in the system. Important fields (so far as memory management is concerned) are:

count: This is a count of the number of users of this page. The count is greater than one when the page is shared between many processes,

age: This field describes the age of the page and is used to decide if the page is a good candidate for discarding or swapping,

map_nr: This is the physical page frame number that this mem_map_t describes.

The free_area vector is used by the page allocation code to find and free pages. The whole buffer management scheme is supported by this mechanism and so far as the code is concerned, the size of the page and physical paging mechanisms used by the processor are irrelevant.

Each element of free_area contains information about blocks of pages. The first element in the array describes single pages, the next blocks of 2 pages, the next blocks of 4 pages and so on upwards in powers of two. The list element is used as a queue head and has pointers to the page data structures in the mem_map array. Free blocks of pages are queued here. map is a pointer to a bitmap which keeps track of allocated groups of pages of this size. Bit N of the bitmap is set if the Nth block of pages is free.

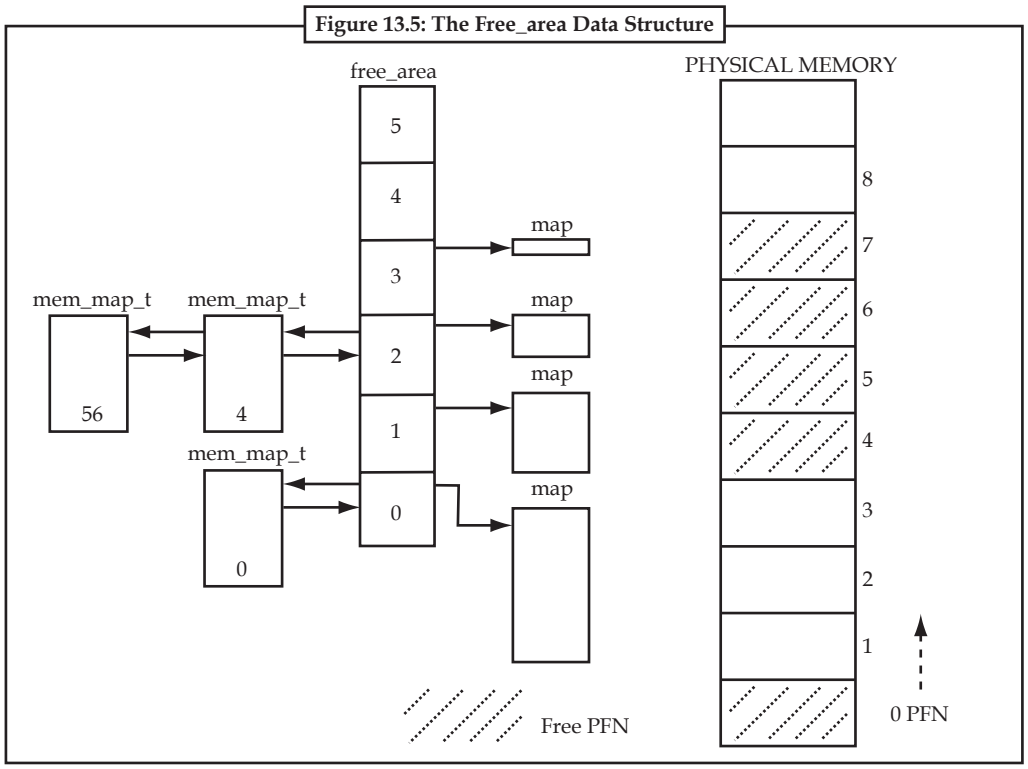
Figure 13.5 shows the free_area structure. Element 0 has one free page (page frame number 0) and element 2 has 2 free blocks of 4 pages, the first starting at page frame number 4 and the second at page frame number 56.

Page Allocation

Linux uses the Buddy algorithm to effectively allocate and deallocate blocks of pages. The page allocation code and attempts to allocate a block of one or more physical pages. Pages are allocated in blocks which are powers of 2 in size. That means that it can allocate a block 1 page, 2 pages, 4 pages and so on. So long as there are enough free pages in the system to grant this request (nr_free_pages ?min_free_pages) the allocation code will search the free_area for a block of pages of the size requested. Each element of the free_area has a map of the allocated and free blocks of pages for that sized block. For example, element 2 of the array has a memory map that describes free and allocated blocks each of 4 pages long.

Notes

Figure 13.5: The Free_area Data Structure



The allocation algorithm first searches for blocks of pages of the size requested. It follows the chain of free pages that is queued on the list element of the free_area data structure. If no blocks of pages of the requested size are free, blocks of the next size (which is twice that of the size requested) are looked for. This process continues until all of the free_area has been searched or until a block of pages has been found. If the block of pages found is larger than that requested it must be broken down until there is a block of the right size. Because the blocks are each a power of 2 pages big then this breaking down process is easy as you simply break the blocks in half. The free blocks are queued on the appropriate queue and the allocated block of pages is returned to the caller.

For example, in Figure 13.5 if a block of 2 pages was requested, the first block of 4 pages (starting at page frame number 4) would be broken into two 2 page blocks. The first, starting at page frame number 4 would be returned to the caller as the allocated pages and the second block, starting at page frame number 6 would be queued as a free block of 2 pages onto element 1 of the free_area array.

Page Deallocation

Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code recombines pages into larger blocks of free pages whenever it can. In fact the page block size is important as it allows for easy combination of blocks into larger blocks.

Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free. If it is, then it is combined with the newly freed block of pages to form a new free block of pages for the next size block of pages. Each time two blocks of pages are recombined into a bigger block of free pages the page deallocation code attempts to recombine that block into a yet larger one. In this way the blocks of free pages are as large as memory usage will allow.

Notes

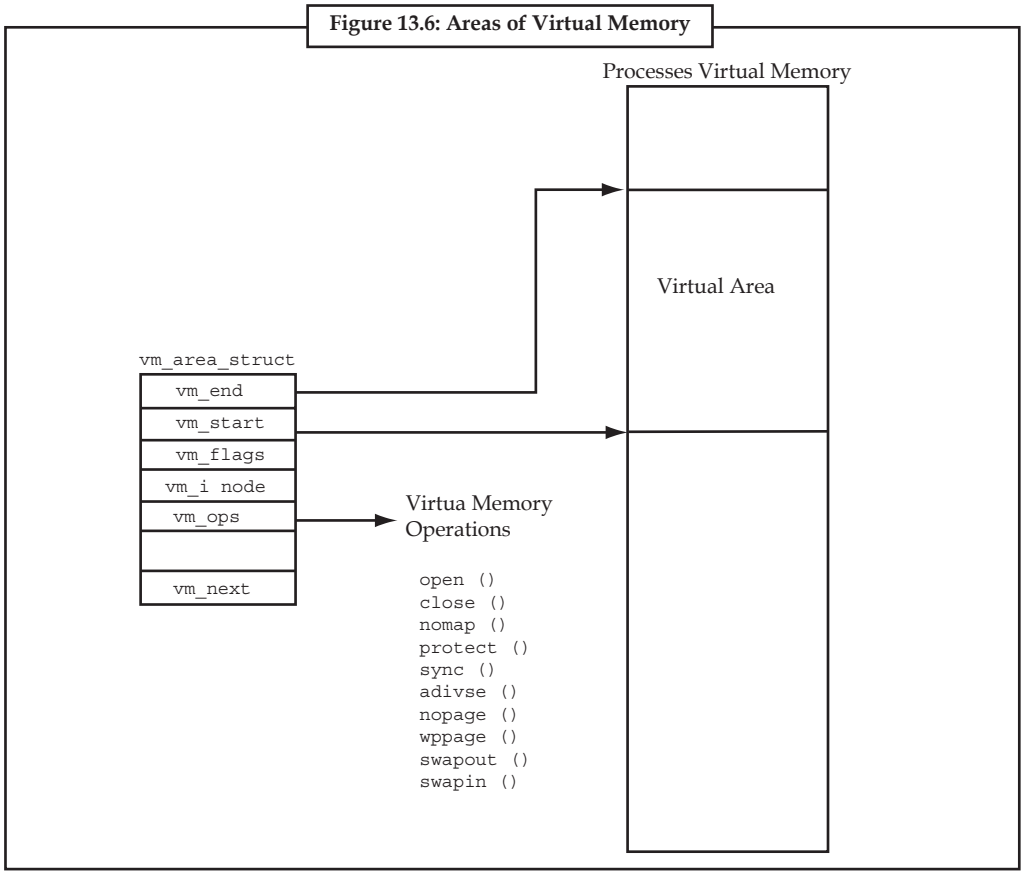
For example, in Figure 13.5, if page frame number 1 were to be freed, then that would be combined with the already free page frame number 0 and queued onto element 1 of the free_area as a free block of size 2 pages.

13.5.9 Memory Mapping

When an image is executed, the contents of the executable image must be brought into the processes virtual address space. The same is also true of any shared libraries that the executable image has been linked to use. The executable file is not actually brought into physical memory, instead it is merely linked into the processes virtual memory. Then, as the parts of the program are referenced by the running application, the image is brought into memory from the executable image. This linking of an image into a processes virtual address space is known as memory mapping.

Every processes virtual memory is represented by an mm_struct data structure. This contains information about the image that it is currently executing (for example bash) and also has pointers to a number of vm_area_struct data structures. Each vm_area_struct data structure describes the start and end of the area of virtual memory, the processes access rights to that memory and a set of operations for that memory. These operations are a set of routines that Linux must use when manipulating this area of virtual memory. For example, one of the virtual memory operations performs the correct actions when the process has attempted to access this virtual memory but finds (via a page fault) that the memory is not actually in physical memory. This operation is the nopage operation. The nopage operation is used when Linux demand pages the pages of an executable image into memory.

Figure 13.6: Areas of Virtual Memory



Notes

When an executable image is mapped into a processes virtual address a set of vm_area_struct data structures is generated. Each vm_area_struct data structure represents a part of the executable image; the executable code, initialized data (variables), unitialized data and so on. Linux supports a number of standard virtual memory operations and as the vm_area_struct data structures are created, the correct set of virtual memory operations are associated with them.

13.5.10 Demand Paging

Once an executable image has been memory mapped into a processes virtual memory it can start to execute. As only the very start of the image is physically pulled into memory it will soon access an area of virtual memory that is not yet in physical memory. When a process accesses a virtual address that does not have a valid page table entry, the processor will report a page fault to Linux.

The page fault describes the virtual address where the page fault occurred and the type of memory access that caused.

Linux must find the vm_area_struct that represents the area of memory that the page fault occurred in. As searching through the vm_area_struct data structures is critical to the efficient handling of page faults, these are linked together in an AVL (Adelson-Velskii and Landis) tree structure. If there is no vm_area_struct data structure for this faulting virtual address, this process has accessed an illegal virtual address. Linux will signal the process, sending a SIGSEGV signal, and if the process does not have a handler for that signal it will be terminated.

Linux next checks the type of page fault that occurred against the types of accesses allowed for this area of virtual memory. If the process is accessing the memory in an illegal way, say writing to an area that it is only allowed to read from, it is also signalled with a memory error.

Now that Linux has determined that the page fault is legal, it must deal with it.

Linux must differentiate between pages that are in the swap file and those that are part of an executable image on a disk somewhere. It does this by using the page table entry for this faulting virtual address.

If the page's page table entry is invalid but not empty, the page fault is for a page currently being held in the swap file. For Alpha AXP page table entries, these are entries which do not have their valid bit set but which have a non-zero value in their PFN field. In this case the PFN field holds information about where in the swap (and which swap file) the page is being held. How pages in the swap file are handled is described later in this unit.

Not all vm_area_struct data structures have a set of virtual memory operations and even those that do may not have a nopage operation. This is because by default Linux will fix up the access by allocating a new physical page and creating a valid page table entry for it. If there is a nopage operation for this area of virtual memory, Linux will use it.

The generic Linux nopage operation is used for memory mapped executable images and it uses the page cache to bring the required image page into physical memory.

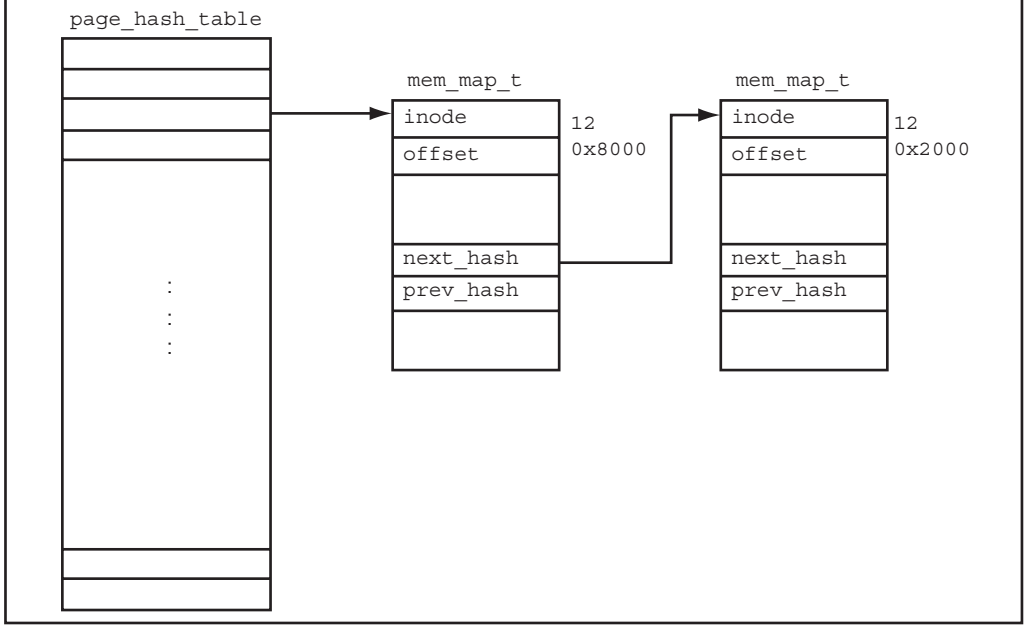
However the required page is brought into physical memory, the processes page tables are updated. It may be necessary for hardware specific actions to update those entries, particularly if the processor uses translation look aside buffers. Now that the page fault has been handled it can be dismissed and the process is restarted at the instruction that made the faulting virtual memory access.

13.5.11 The Linux Page Cache

The role of the Linux page cache is to speed up access to files on disk. Memory mapped files are read a page at a time and these pages are stored in the page cache. Figure 13.7 shows that the page cache consists of the page_hash_table, a vector of pointers to mem_map_t data structures.

Notes

Figure 13.7: The Linux Page Cache



Each file in Linux is identified by a VFS inode data structure and each VFS inode is unique and fully describes one and only one file. The index into the page table is derived from the file's VFS inode and the offset into the file.

Whenever a page is read from a memory mapped file, for example when it needs to be brought back into memory during demand paging, the page is read through the page cache. If the page is present in the cache, a pointer to the mem_map_t data structure representing it is returned to the page fault handling code. Otherwise the page must be brought into memory from the file system that holds the image. Linux allocates a physical page and reads the page from the file on disk.

If it is possible, Linux will initiate a read of the next page in the file. This single page read ahead means that if the process is accessing the pages in the file serially, the next page will be waiting in memory for the process.

Over time the page cache grows as images are read and executed. Pages will be removed from the cache as they are no longer needed, say as an image is no longer being used by any process. As Linux uses memory it can start to run low on physical pages. In this case Linux will reduce the size of the page cache.

13.5.12 Swapping Out and Discarding Pages

When physical memory becomes scarce the Linux memory management subsystem must attempt to free physical pages. This task falls to the kernel swap daemon (kswapd).

The kernel swap daemon is a special type of process, a kernel thread. Kernel threads are processes have no virtual memory, instead they run in kernel mode in the physical address space. The kernel swap daemon is slightly misnamed in that it does more than merely swap pages out to the system's swap files. Its role is make sure that there are enough free pages in the system to keep the memory management system operating efficiently.

The Kernel swap daemon (kswapd) is started by the kernel init process at startup time and sits waiting for the kernel swap timer to periodically expire.

Notes

Every time the timer expires, the swap daemon looks to see if the number of free pages in the system is getting too low. It uses two variables, free_pages_high and free_pages_low to decide if it should free some pages. So long as the number of free pages in the system remains above free_pages_high, the kernel swap daemon does nothing; it sleeps again until its timer next expires. For the purposes of this check the kernel swap daemon takes into account the number of pages currently being written out to the swap file. It keeps a count of these in nr_async_pages; this is incremented each time a page is queued waiting to be written out to the swap file and decremented when the write to the swap device has completed. free_pages_low and free_pages_high are set at system startup time and are related to the number of physical pages in the system. If the number of free pages in the system has fallen below free_pages_high or worse still free_pages_low, the kernel swap daemon will try three ways to reduce the number of physical pages being used by the system:

1. Reducing the size of the buffer and page caches,
2. Swapping out System V shared memory pages,
3. Swapping out and discarding pages.

If the number of free pages in the system has fallen below free_pages_low, the kernel swap daemon will try to free 6 pages before it next runs. Otherwise it will try to free 3 pages. Each of the above methods are tried in turn until enough pages have been freed. The kernel swap daemon remembers which method it was using the last time that it attempted to free physical pages. Each time it runs it will start trying to free pages using this last successful method.

After it has free sufficient pages, the swap daemon sleeps again until its timer expires. If the reason that the kernel swap daemon freed pages was that the number of free pages in the system had fallen below free_pages_low, it only sleeps for half its usual time. Once the number of free pages is more than free_pages_low the kernel swap daemon goes back to sleeping longer between checks.

13.5.13 Reducing the Size of the Page and Buffer Caches

The pages held in the page and buffer caches are good candidates for being freed into the free_area vector. The Page Cache, which contains pages of memory mapped files, may contain unneccessary pages that are filling up the system's memory. Likewise the Buffer Cache, which contains buffers read from or being written to physical devices, may also contain unneeded buffers. When the physical pages in the system start to run out, discarding pages from these caches is relatively easy as it requires no writing to physical devices (unlike swapping pages out of memory). Discarding these pages does not have too many harmful side effects other than making access to physical devices and memory mapped files slower. However, if the discarding of pages from these caches is done fairly, all processes will suffer equally.

Every time the Kernel swap daemon tries to shrink these caches it examines a block of pages in the mem_map page vector to see if any can be discarded from physical memory. The size of the block of pages examined is higher if the kernel swap daemon is intensively swapping; that is if the number of free pages in the system has fallen dangerously low. The blocks of pages are examined in a cyclical manner; a different block of pages is examined each time an attempt is made to shrink the memory map. This is known as the clock algorithm as, rather like the minute hand of a clock, the whole mem_map page vector is examined a few pages at a time.

Each page being examined is checked to see if it is cached in either the page cache or the buffer cache. You should note that shared pages are not considered for discarding at this time and that a page cannot be in both caches at the same time. If the page is not in either cache then the next page in the mem_map page vector is examined.

Pages are cached in the buffer cache (or rather the buffers within the pages are cached) to make buffer allocation and deallocation more efficient. The memory map shrinking code tries to free the buffers that are contained within the page being examined.

Notes

If all the buffers are freed, then the pages that contain them are also be freed. If the examined page is in the Linux page cache, it is removed from the page cache and freed.

When enough pages have been freed on this attempt then the kernel swap daemon will wait until the next time it is periodically woken. As none of the freed pages were part of any process's virtual memory (they were cached pages), then no page tables need updating. If there were not enough cached pages discarded then the swap daemon will try to swap out some shared pages.

13.5.14 Swapping Out System V Shared Memory Pages

System V shared memory is an inter-process communication mechanism which allows two or more processes to share virtual memory in order to pass information amongst themselves. For now it is enough to say that each area of System V shared memory is described by a shmid_ds data structure. This contains a pointer to a list of vm_area_struct data structures, one for each process sharing this area of virtual memory. The vm_area_struct data structures describe where in each processes virtual memory this area of System V shared memory goes. Each vm_area_struct data structure for this System V shared memory is linked together using the vm_next_shared and vm_prev_shared pointers. Each shmid_ds data structure also contains a list of page table entries each of which describes the physical page that a shared virtual page maps to.

The kernel swap daemon also uses a clock algorithm when swapping out System V shared memory pages. Each time it runs it remembers which page of which shared virtual memory area it last swapped out. It does this by keeping two indices, the first is an index into the set of shmid_ds data structures, the second into the list of page table entries for this area of System V shared memory. This makes sure that it fairly victimizes the areas of System V shared memory.

As the physical page frame number for a given virtual page of System V shared memory is contained in the page tables of all of the processes sharing this area of virtual memory, the kernel swap daemon must modify all of these page tables to show that the page is no longer in memory but is now held in the swap file. For each shared page it is swapping out, the kernel swap daemon finds the page table entry in each of the sharing processes page tables (by following a pointer from each vm_area_struct data structure). If this processes page table entry for this page of System V shared memory is valid, it converts it into an invalid but swapped out page table entry and reduces this (shared) page's count of users by one. The format of a swapped out System V shared page table entry contains an index into the set of shmid_ds data structures and an index into the page table entries for this area of System V shared memory.

If the page's count is zero after the page tables of the sharing processes have all been modified, the shared page can be written out to the swap file. The page table entry in the list pointed at by the shmid_ds data structure for this area of System V shared memory is replaced by a swapped out page table entry. A swapped out page table entry is invalid but contains an index into the set of open swap files and the offset in that file where the swapped out page can be found. This information will be used when the page has to be brought back into physical memory.

13.5.15 Swapping Pages In

The dirty pages saved in the swap files may be needed again, for example when an application writes to an area of virtual memory whose contents are held in a swapped out physical page. Accessing a page of virtual memory that is not held in physical memory causes a page fault to occur. The page fault is the processor signalling the operating system that it cannot translate a virtual address into a physical one. In this case this is because the page table entry describing this page of virtual memory was marked as invalid when the page was swapped out. The processor cannot handle the virtual to physical address translation and so hands control back to the operating system describing as it does so the virtual address that faulted and the reason for the fault. The format of this information and how the processor passes control to the operating system is processor specific.

Notes

The processor specific page fault handling code must locate the vm_area_struct data structure that describes the area of virtual memory that contains the faulting virtual address. It does this by searching the vm_area_struct data structures for this process until it finds the one containing the faulting virtual address. This is very time critical code and a processes vm_area_struct data structures are so arranged as to make this search take as little time as possible.

Having carried out the appropriate processor specific actions and found that the faulting virtual address is for a valid area of virtual memory, the page fault processing becomes generic and applicable to all processors that Linux runs on.

The generic page fault handling code looks for the page table entry for the faulting virtual address. If the page table entry it finds is for a swapped out page, Linux must swap the page back into physical memory. The format of the page table entry for a swapped out page is processor specific but all processors mark these pages as invalid and put the information necessary to locate the page within the swap file into the page table entry. Linux needs this information in order to bring the page back into physical memory.

At this point, Linux knows the faulting virtual address and has a page table entry containing information about where this page has been swapped to. The vm_area_struct data structure may contain a pointer to a routine which will swap any page of the area of virtual memory that it describes back into physical memory. This is its swapin operation. If there is a swapin operation for this area of virtual memory then Linux will use it. This is, in fact, how swapped out System V shared memory pages are handled as it requires special handling because the format of a swapped out System V shared page is a little different from that of an ordinary swapped out page. There may not be a swapin operation, in which case Linux will assume that this is an ordinary page that does not need to be specially handled.

It allocates a free physical page and reads the swapped out page back from the swap file. Information telling it where in the swap file (and which swap file) is taken from the the invalid page table entry.

If the access that caused the page fault was not a write access then the page is left in the swap cache and its page table entry is not marked as writable. If the page is subsequently written to, another page fault will occur and, at that point, the page is marked as dirty and its entry is removed from the swap cache. If the page is not written to and it needs to be swapped out again, Linux can avoid the write of the page to its swap file because the page is already in the swap file.

If the access that caused the page to be brought in from the swap file was a write operation, this page is removed from the swap cache and its page table entry is marked as both dirty and writable.

13.6 File Systems

The first thing that most new users shifting from Windows will find confusing is navigating the Linux filesystem. The Linux filesystem does things a lot more differently than the Windows filesystem. This article explains the differences and takes you through the layout of the Linux filesystem.

For starters, there is only a single hierarchal directory structure. Everything starts from the root directory, represented by '/', and then expands into sub-directories. Where DOS/Windows had various partitions and then directories under those partitions, Linux places all the partitions under the root directory by 'mounting' them under specific directories. Closest to root under Windows would be c:.

Under Windows, the various partitions are detected at boot and assigned a drive letter. Under Linux, unless you mount a partition or a device, the system does not know of the existence of that partition or device. This might not seem to be the easiest way to provide access to your partitions or devices but it offers great flexibility.

Notes

This kind of layout, known as the unified filesystem, does offer several advantages over the approach that Windows uses. Let's take the example of the /usr directory. This directory off the root directory contains most of the system executables. With the Linux filesystem, you can choose to mount it off another partition or even off another machine over the network. The underlying system will not know the difference because /usr appears to be a local directory that is part of the local directory structure! How many times have you wished to move around executables and data under Windows, only to run into registry and system errors? Try moving c:windowssystem to another partition or drive.

Another point likely to confuse newbies is the use of the frontslash '/' instead of the backslash '' as in DOS/Windows. So c:windowssystem would be /c/windows/system. Well, Linux is not going against convention here. Unix has been around a lot longer than Windows and was the standard a lot before Windows was. Rather, DOS took the different path, using '/' for command-line options and '' as the directory separator.

To liven up matters even more, Linux also chooses to be case sensitive. What this means that the case, whether in capitals or not, of the characters becomes very important. So this is not the same as THIS or ThIs for that matter. This one feature probably causes the most problems for newbies.

We now move on to the layout or the directory structure of the Linux filesystem. Given below is the result of a 'ls -p' in the root directory.

```
bin/ dev/ home/ lost+found/ proc/ sbin/ usr/ boot/ etc/ lib/ mnt/ root/ tmp/
var/
```

/sbin: This directory contains all the binaries that are essential to the working of the system. These include system administration as well as maintenance and hardware configuration programs. Find lilo, fdisk, init, ifconfig etc here. These are the essential programs that are required by all the users. Another directory that contains system binaries is /usr/sbin.

This directory contains other binaries of use to the system administrator.

This is where you will find the network daemons for your system along with other binaries that only the system administrator has access to, but which are not required for system maintenance, repair etc.

/bin: In contrast to /sbin, the bin directory contains several useful commands that are used by both the system administrator as well as non-privileged users. This directory usually contains the shells like bash, cash etc. as well as much used commands like cp, mv, rm, cat, ls. There also is /usr/bin, which contains other user binaries. These binaries on the other hand are not essential for the user. The binaries in /bin however, a user cannot do without.

/boot: This directory contains the system.map file as well as the Linux kernel. Lilo places the boot sector backups in this directory.

/dev: This is a very interesting directory that highlights one important characteristic of the Linux filesystem - everything is a file or a directory. Look through this directory and you should see hda1, hda2 etc, which represent the various partitions on the first master drive of the system. /dev/cdrom and /dev/fd0 represent your CDROM drive and your floppy drive. This may seem strange but it will make sense if you compare the characteristics of files to that of your hardware. Both can be read from and written to. Take /dev/dsp, for instance. This file represents your speaker device. So any data written to this file will be re-directed to your speaker. Try 'cat /etc/lilo.conf > /dev/dsp' and you should hear some sound on the speaker. That's the sound of your lilo.conf file! Similarly, sending data to and reading from /dev/ttyS0 (COM 1) will allow you to communicate with a device attached there - your modem.

/etc: This directory contains all the configuration files for your system. Your lilo.conf file lies in this directory as does hosts, resolv.conf and fstab. Under this directory will be X11 sub-directory

Notes

which contains the configuration files for X. More importantly, the /etc/rc.d directory contains the system startup scripts. This is a good directory to backup often. It will definitely save you a lot of re-configuration later if you re-install or lose your current installation.

/home: Linux is a multi-user environment so each user is also assigned a specific directory which is accessible only to them and the system administrator. These are the user home directories, which can be found under /home/username. This directory also contains the user specific settings for programs like IRC, X etc.

/lib: This contains all the shared libraries that are required by system programs. Windows equivalent to a shared library would be a DLL file.

/lost+foun: Linux should always go through a proper shutdown. Sometimes your system might crash or a power failure might take the machine down. Either way, at the next boot, a lengthy filesystem check using fsck will be done. Fsck will go through the system and try to recover any corrupt files that it finds. The result of this recovery operation will be placed in this directory. The files recovered are not likely to be complete or make much sense but there always is a chance that something worthwhile is recovered.

/mnt: This is a generic mount point under which you mount your filesystems or devices. Mounting is the process by which you make a filesystem available to the system. After mounting your files will be accessible under the mount-point. This directory usually contains mount points or sub-directories where you mount your floppy and your CD. You can also create additional mount-points here if you want. There is no limitation to creating a mount-point anywhere on your system but convention says that you do not litter your file system with mount-points.

/opt: This directory contains all the software and add-on packages that are not part of the default installation. Generally you will find KDE and StarOffice here. Again, this directory is not used very often as it's mostly a standard in Unix installations.

/proc: This is a special directory on your system.

/root: We talked about user home directories earlier and well this one is the home directory of the user root. This is not to be confused with the system root, which is directory at the highest level in the filesystem.

/tmp: This directory contains mostly files that are required temporarily. Many programs use this to create lock files and for temporary storage of data. On some systems, this directory is cleared out at boot or at shutdown.

/usr: This is one of the most important directories in the system as it contains all the user binaries. X and its supporting libraries can be found here. User programs like telnet, ftp etc are also placed here.

/usr/doc contains useful system documentation. /usr/src/linux contains the source code for the Linux kernel.

/var: This directory contains spooling data like mail and also the output from the printer daemon. The system logs are also kept here in /var/log/messages. You will also find the database for BIND in /var/named and for NIS in /var/yp.

13.7 Input & Output

I/O Event handling is about how your Operating System allows you to manage a large number of open files in your application. You want the OS to notify you when FDs become active (have data ready to be read or are ready for writing).

Notes

The Traditional UNIX Way

Traditional Unix systems provide the select(2) and/or poll(2) system calls. With both of these you pass an array of FDs to the kernel, with an optional timeout. When there is activity, or when the call times out, the system call will return. The application must then scan the array to see which FDs are active. This scheme works well with small numbers of FDs, and is simple to use. Unfortunately, for thousands of FDs, this does not work so well.

The kernel has to scan your array of FDs and check which ones are active. This takes approximately 3 microseconds (3 us) per FD on a Pentium 100 running Linux 2.1.x. Now you might think that 3 us is quite fast, but consider if you have an array of 1000 FDs. This is now 3 milliseconds (3 ms), which is 30% of your timeslice (each timeslice is 10 ms). If it happens that there is initially no activity and you specified a timeout, the kernel will have to perform a second scan after some activity occurs or the syscall times out. Ouch! If you have an even bigger application (like a large http server), you can easily have 10000 FDs. Scanning times will then take 30 ms, which is three timeslices! This is just way too much.

You might say that 3 ms for 1000 FDs is not such a big deal: a user will hardly notice that. The problem is that the entire array of FDs is scanned each time you want to go back to your polling loop. The way these applications work is that after checking for activity on FDs, the application processes the activity (for example, reading data from active FDs). When all the activity has been processed, the application goes back to polling the OS for more FD activity. In many cases, only a small number of FDs are active at any one time (say during each timeslice), so it may only take a few milliseconds to process all the activity. High performance http servers can process hundreds or thousands of transactions per second. A server that takes 2 ms to process each active FD can process 500 transactions per second. If you add 3 ms for FD scanning in the kernel, you now have 5 ms per transaction. That only gives 200 transactions per second, a massive drop in performance.

There is another problem, and that is that the application needs to scan the "returned" FD array that the kernel has updated to see which FDs are active. This is yet another scan of a large array. This isn't as costly as the kernel scan, for reasons I'll get to later, but it is still a finite cost.

New POSIX Interfaces

A fairly simple proposal is to use the POSIX.4 Asynchronous I/O (AIO) interface (aio_read() and friends). Here we would call aio_read() for each FD. This would then queue thousands of asynchronous I/O requests. This model looks appealing, until we look under the hood of some aio_*() implementations. The Linux glibc implementation is a case in point: there is no kernel support. Instead, the C library (glibc 2.1) launches a thread per FD for which there are outstanding AIO requests (up to the maximum number of configured threads). In general, implementing this facility in the C library is reasonable, as it avoids kernel bloat. However, if you use this facility to start thousands of AIO requests, you may end up creating thousands of threads. This is no good, since threads are costly. The "obvious" solution is to implement AIO in the Linux kernel, then. Another solution is to use userspace tricks to avoid the scalability problems (see the description of migrating FDs below). These solutions may be fine if you only want to run under Linux, but is not much help if you want to run under another OS which also implements AIO using threads (and for which you don't have the source code so you can change the implementation). The point here is that there appears to be no guarantee that aio_*() implementations are scalable across platforms which support it.

It is also worth noting that POSIX.4 Asynchronous I/O is not necessarily available on all POSIX.4 compliant systems (facilities defined by POSIX.4 are optional). So even if you were prepared to limit your application to POSIX.4 systems, there is still no guarantee that AIO is available. Many or most implementations will be scalable, but we can't be sure all are scalable, so we need an alternative.

Optimising existing UNIX Interfaces

Notes

There are improvements we can make for the massive FD scanning problem. Firstly we can optimise the way the scanning is done inside the kernel. Right now (2.1.106) the kernel has to call the poll() method for each file structure. This is expensive. Back in the 2.1.5x kernels, I coded a better implementation for the kernel which sped things up almost 3 times. While this requires modifications to drivers to take advantage of this, it has the advantage of not changing the semantics we expect from UNIX. Note one other interesting feature of this optimisation: it centralises event notification, which in turn would make implementing I/O readiness queues simpler. I'm not sure how closure of FDs before readiness events are read should be handled. This could complicate their implementation.

Doing this optimisation does not solve our problem, though. It only pushes the problem away for a while.

Making better use of existing UNIX Interfaces

Note that for my purposes, it is better to optimise the application so that it works well on many OSes rather than optimising a single OS. Creating new interfaces for Linux is a last resort. Also note that this section assumes that an OS of interest does not have an existing (preferably POSIX) mechanism that supports FD management in a scalable way.

Another solution (which would also benefit from the kernel optimisation discussed above) is for the application to divide the FD array into a number of smaller FD arrays, say 10. You then create 10 threads, each of which has a polling loop using its smaller FD array. So each FD array is now 100 entries long. While this doesn't change the total number of FDs that must be scanned, it does change when they have to be scanned. Since most FDs are inactive, not all the threads will be woken up. Too see how this works, consider the example where, at any time (say during a single timeslice of 10 ms), only 5 FDs are active. Assuming these FDs are randomly, uniformly distributed, at most 5 threads will need to be woken up. These threads then process the activity and go back to the start of their polling loops. Where we win is that only 5 threads had to go back and call select(2) or poll(2). Since they each have 100 entry FD arrays, the kernel only has to scan 500 FDs. This has halved the amount of scanning required. The scanning load has gone from 30% to 15% by this simple change. If you were to instead use 100 threads, you would still only have at most 5 threads woken up for activity, and hence the total number of FDs scanned this timeslice would be 50. This takes down the scanning load to 0.15%, which is negligible.

There is one thing to watch out for here: if you use select(2) in your polling loop, be aware that the size of your FD array is equal to the value of your largest FD. This is because select(2) uses a bitmask for its FD array. This means one of your threads will want to poll FDs 991 to 1000. Unfortunately, your FD array is still 1000 long. What's worse, the kernel still has to do a minimal scan for all those 1000 FDs. The solution to this is to use poll(2) instead, where you only have to pass as many FDs as you want to poll, and the kernel scans only those.

This solution sounds ideal: just create lots and lots of threads. At the extreme, you create one thread per FD. There is a problem here, however, as each thread consumes system resources. So you need to compromise between the number of threads and the FD scanning load. Also, the more threads you have the more cache misses you induce, so this is something to avoid as well. Fortunately in this case most threads will be running nearly the same code at the same time, so cache pollution should not be a significant problem.

A more advanced solution is to have dynamic migration of FDs depending on whether they are mostly active or inactive. In the simplest case, you only have two threads. One which polls mostly active FDs and the other polls mostly inactive FDs. The thread for active FDs will be woken up very frequently, but on the other hand will have only a small number of FDs to scan. The other thread will have to scan a large number of FDs, but it will only be woken up occasionally. For

Notes

each FD an activity counter is kept. When a FD on the mostly inactive list is deemed to be fairly active, it is migrated to the mostly active list. A reverse operation occurs for fairly inactive FDs on the mostly active list.

13.8 Inter-process Communication

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix TM release in which they first appeared.

13.8.1 Signals

Signals are one of the oldest inter-process communication methods used by Unix TM systems. They are used to signal asynchronous events to one or more processes. A signal could be generated by a keyboard interrupt or an error condition such as the process attempting to access a non-existent location in its virtual memory. Signals are also used by the shells to signal job control commands to their child processes.

There are a set of defined signals that the kernel can generate or that can be generated by other processes in the system, provided that they have the correct privileges. You can list a system's set of signals using the kill command (kill -l), on my Intel Linux box this gives:

1)	SIGHUP	2)	SIGINT	3)	SIGQUIT	4)	SIGILL
5)	SIGTRAP	6)	SIGIOT	7)	SIGBUS	8)	SIGFPE
9)	SIGKILL	10)	SIGUSR1	11)	SIGSEGV	12)	SIGUSR2
13)	SIGPIPE	14)	SIGALRM	15)	SIGTERM	17)	SIGCHLD
18)	SIGCONT	19)	SIGSTOP	20)	SIGTSTP	21)	SIGTTIN
22)	SIGTTOU	23)	SIGURG	24)	SIGXCPU	25)	SIGXFSZ
26)	SIGVTALRM	27)	SIGPROF	28)	SIGWINCH	29)	SIGIO
30)	SIGPWR						

The numbers are different for an Alpha AXP Linux box. Processes can choose to ignore most of the signals that are generated, with two notable exceptions: neither the SIGSTOP signal which causes a process to halt its execution nor the SIGKILL signal which causes a process to exit can be ignored. Otherwise though, a process can choose just how it wants to handle the various signals. Processes can block the signals and, if they do not block them, they can either choose to handle them themselves or allow the kernel to handle them. If the kernel handles the signals, it will do the default actions required for this signal. For example, the default action when a process receives the SIGFPE (floating point exception) signal is to core dump and then exit. Signals have no inherent relative priorities. If two signals are generated for a process at the same time then they may be presented to the process or handled in any order. Also there is no mechanism for handling multiple signals of the same kind. There is no way that a process can tell if it received 1 or 42 SIGCONT signals.

Linux implements signals using information stored in the task_struct for the process. The number of supported signals is limited to the word size of the processor. Processes with a word size of 32 bits can have 32 signals whereas 64 bit processors like the Alpha AXP may have up to 64 signals. The currently pending signals are kept in the signal field with a mask of blocked signals held in blocked. With the exception of SIGSTOP and SIGKILL, all signals can be blocked. If a blocked signal is generated, it remains pending until it is unblocked. Linux also holds information about how each process handles every possible signal and this is held in an array of sigaction data

Notes

structures pointed at by the task_struct for each process. Amongst other things it contains either the address of a routine that will handle the signal or a flag which tells Linux that the process either wishes to ignore this signal or let the kernel handle the signal for it. The process modifies the default signal handling by making system calls and these calls alter the sigaction for the appropriate signal as well as the blocked mask.

Not every process in the system can send signals to every other process, the kernel can and super users can. Normal processes can only send signals to processes with the same uid and gid or to processes in the same process group. Signals are generated by setting the appropriate bit in the task_struct's signal field. If the process has not blocked the signal and is waiting but interruptible (in state Interruptible) then it is woken up by changing its state to Running and making sure that it is in the run queue. That way the scheduler will consider it a candidate for running when the system next schedules. If the default handling is needed, then Linux can optimize the handling of the signal.



Example: If the signal SIGWINCH (the X window changed focus) and the default handler is being used then there is nothing to be done.

Signals are not presented to the process immediately they are generated., they must wait until the process is running again. Every time a process exits from a system call its signal and blocked fields are checked and, if there are any unblocked signals, they can now be delivered. This might seem a very unreliable method but every process in the system is making system calls, for example to write a character to the terminal, all of the time. Processes can elect to wait for signals if they wish, they are suspended in state Interruptible until a signal is presented. The Linux signal processing code looks at the sigaction structure for each of the current unblocked signals.

If a signal's handler is set to the default action then the kernel will handle it. The SIGSTOP signal's default handler will change the current process's state to Stopped and then run the scheduler to select a new process to run. The default action for the SIGFPE signal will core dump the process and then cause it to exit. Alternatively, the process may have specified its own signal handler. This is a routine which will be called whenever the signal is generated and the sigaction structure holds the address of this routine. The kernel must call the process's signal handling routine and how this happens is processor specific but all CPUs must cope with the fact that the current process is running in kernel mode and is just about to return to the process that called the kernel or system routine in user mode. The problem is solved by manipulating the stack and registers of the process. The process's program counter is set to the address of its signal handling routine and the parameters to the routine are added to the call frame or passed in registers. When the process resumes operation it appears as if the signal handling routine were called normally.

Linux is POSIX compatible and so the process can specify which signals are blocked when a particular signal handling routine is called. This means changing the blocked mask during the call to the processes signal handler. The blocked mask must be returned to its original value when the signal handling routine has finished. Therefore Linux adds a call to a tidy up routine which will restore the original blocked mask onto the call stack of the signalled process. Linux also optimizes the case where several signal handling routines need to be called by stacking them so that each time one handling routine exits, the next one is called until the tidy up routine is called.

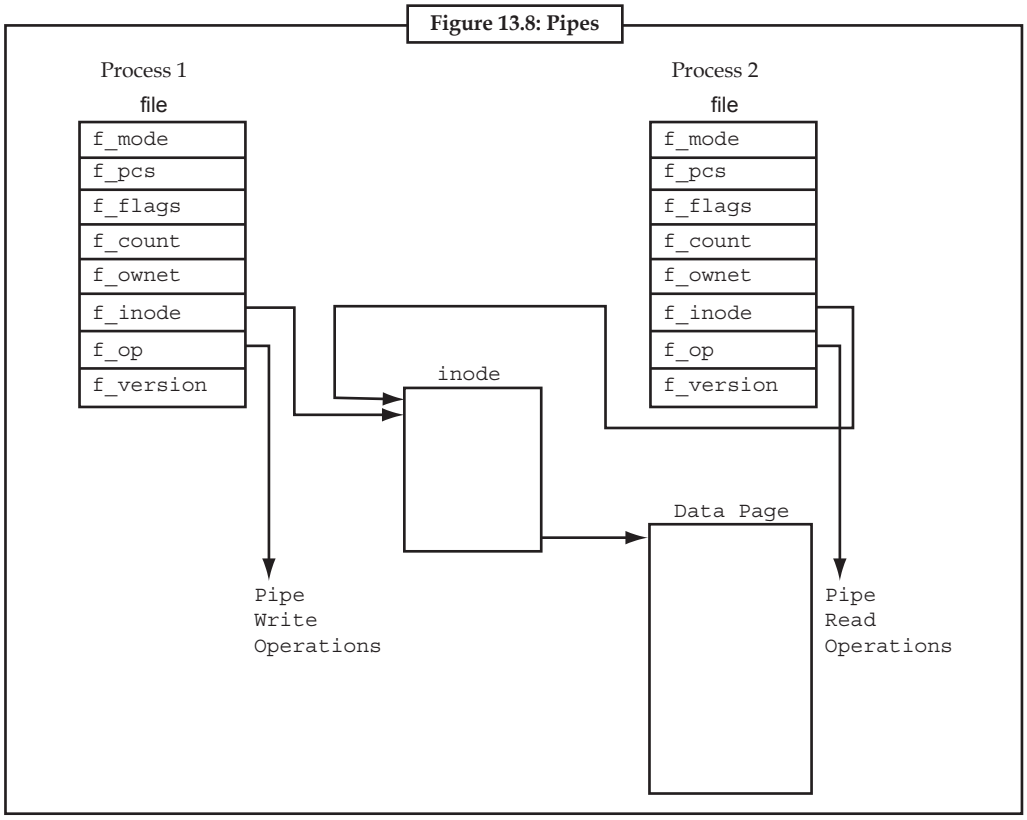
13.8.2 Pipes

The common Linux shells all allow redirection. For example

```
$ ls | pr | lpr
```

Notes

Figure 13.8: Pipes



pipes the output from the ls command listing the directory's files into the standard input of the pr command which paginates them. Finally the standard output from the pr command is piped into the standard input of the lpr command which prints the results on the default printer. Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is the shell which sets up these temporary pipes between the processes.

In Linux, a pipe is implemented using two file data structures which both point at the same temporary VFS inode which itself points at a physical page within memory. Figure 13.8 shows that each file data structure contains pointers to different file operation routine vectors; one for writing to the pipe, the other for reading from the pipe.

This hides the underlying differences from the generic system calls which read and write to ordinary files. As the writing process writes to the pipe, bytes are copied into the shared data page and when the reading process reads from the pipe, bytes are copied from the shared data page. Linux must synchronize access to the pipe. It must make sure that the reader and the writer of the pipe are in step and to do this it uses locks, wait queues and signals.

When the writer wants to write to the pipe it uses the standard write library functions. These all pass file descriptors that are indices into the process's set of file data structures, each one representing an open file or, as in this case, an open pipe. The Linux system call uses the write routine pointed at by the file data structure describing this pipe. That write routine uses information held in the VFS inode representing the pipe to manage the write request.

If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the process's address space into the shared data page. If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe inode's wait queue and the scheduler is called so that another process can run. It is interruptible, so it can

Notes

receive signals and it will be woken by the reader when there is enough room for the write data or when the pipe is unlocked. When the data has been written, the pipe's VFS inode is unlocked and any waiting readers sleeping on the inode's wait queue will themselves be woken up.

Reading data from the pipe is a very similar process to writing to it.

Processes are allowed to do non-blocking reads (it depends on the mode in which they opened the file or pipe) and, in this case, if there is no data to be read or if the pipe is locked, an error will be returned. This means that the process can continue to run. The alternative is to wait on the pipe inode's wait queue until the write process has finished. When both processes have finished with the pipe, the pipe inode is discarded along with the shared data page.

Linux also supports named pipes, also known as FIFOs because pipes operate on a First In, First Out principle. The first data written into the pipe is the first data read from the pipe. Unlike pipes, FIFOs are not temporary objects, they are entities in the file system and can be created using the mkfifo command. Processes are free to use a FIFO so long as they have appropriate access rights to it. The way that FIFOs are opened is a little different from pipes. A pipe (its two file data structures, its VFS inode and the shared data page) is created in one go whereas a FIFO already exists and is opened and closed by its users. Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it. That aside, FIFOs are handled almost exactly the same way as pipes and they use the same data structures and operations.

13.8.3 System V IPC Mechanisms

Linux supports three types of interprocess communication mechanisms that first appeared in Unix TM System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index.

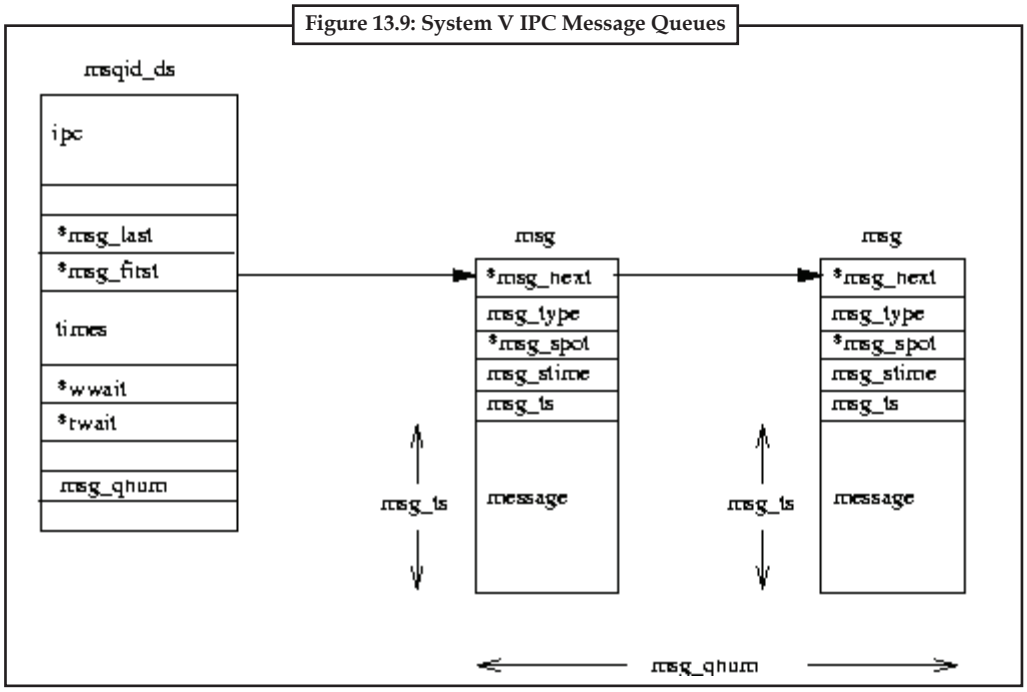
All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator process's user and group identifiers. The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

13.8.4 Message Queues

Message queues allow one or more processes to write messages, which will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector; each element of which points to a msqid_ds data structure that fully describes the message queue. When message queues arxe created a new msqid_ds data structure is allocated from system memory and inserted into the vector.

Notes

Figure 13.9: System V IPC Message Queues



Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid_ds also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the process's address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue.

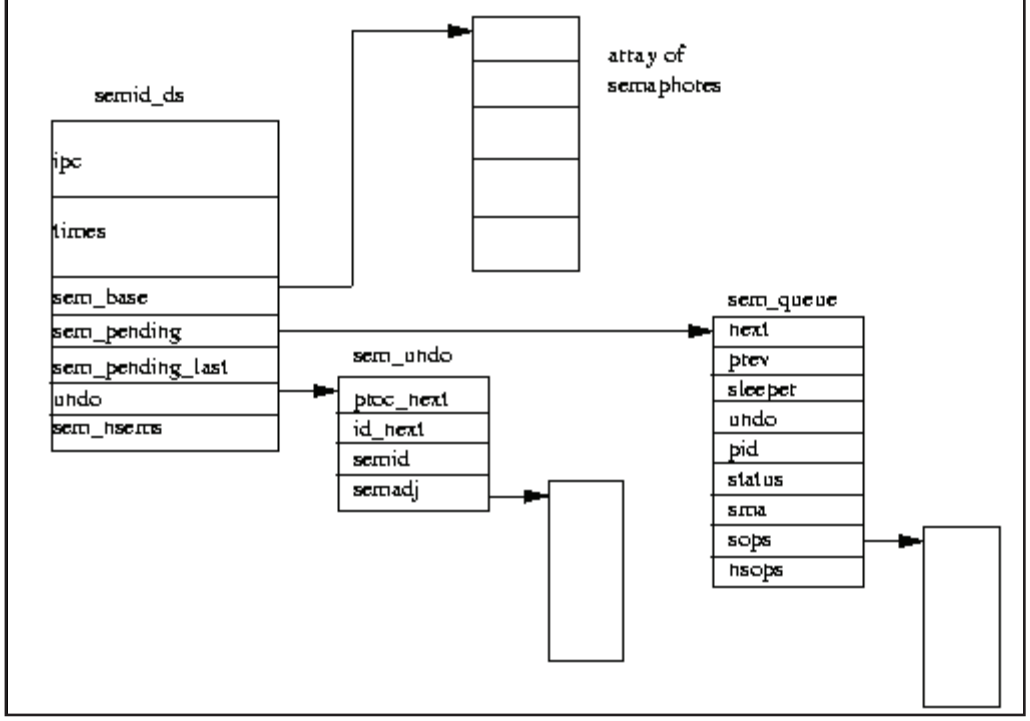
Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

13.8.5 Semaphores

In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation is, so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement critical regions, areas of critical code that only one process at a time should be executing.

Notes

Figure 13.10: System V IPC Semaphores



Say you had many cooperating processes reading records from and writing records to a single data file. You would want that file access to be strictly coordinated. You could use a semaphore with an initial value of 1 and, around the file operating code, put two semaphore operations, the first to test and decrement the semaphore's value and the second to test and increment it. The first process to access the file would try to decrement the semaphore's value and it would succeed, the semaphore's value now being 0. This process can now go ahead and use the data file but if another process wishing to use it now tries to decrement the semaphore's value it would fail as the result would be -1. That process will be suspended until the first process has finished with the data file. When the first process has finished with the data file it will increment the semaphore's value, making it 1 again. Now the waiting process can be woken and this time its attempt to increment the semaphore will succeed.

System V IPC semaphore objects each describe a semaphore array and Linux uses the semid_ds data structure to represent this. All of the semid_ds data structures in the system are pointed at by the smeary, a vector of pointers. There are sem_nsems in each semaphore array, each one described by a sem data structure pointed at by sem_base. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs; the semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building a sem_queue data structure on the stack and filling it out. The new sem_queue data structure is put at the end of this semaphore object's wait queue (using the sem_pending and sem_pending_

Notes

last pointers). The current process is put on the wait queue in the sem_queue data structure (sleeper) and the scheduler called to choose another process to run.

If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array. Now Linux must check that any waiting, suspended, processes may now apply their semaphore operations. It looks at each member of the operations pending queue (sem_pending) in turn, testing to see if the semphore operations will succeed this time. If they will then it removes the sem_queue data structure from the operations pending list and applies the semaphore operations to the semaphore array. It wakes up the sleeping process making it available to be restarted the next time the scheduler runs. Linux keeps looking through the pending list from the start until there is a pass where no semaphore operations can be applied and so no more processes can be woken.

There is a problem with semaphores, deadlocks. These occur when one process has altered the semaphores value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before the a process's set of semaphore operations were applied. These adjustments are kept in sem_undo data structures queued both on the semid_ds data structure and on the task_struct data structure for the processes using these semaphore arrays.

Each individual semaphore operation may request that an adjustment be maintained. Linux will maintain at most one sem_undo data structure per process for each semaphore array. If the requesting process does not have one, then one is created when it is needed. The new sem_undo data structure is queued both onto this process's task_struct data structure and onto the semaphore array's semid_ds data structure. As operations are applied to the semphores in the semaphore array the negation of the operation value is added to this semphore's entry in the adjustment array of this process's sem_undo data structure. So, if the operation value is 2, then -2 is added to the adjustment entry for this semaphore.

When processes are deleted, as they exit Linux works through their set of sem_undo data structures applying the adjustments to the semaphore arrays. If a semaphore set is deleted, the sem_undo data structures are left queued on the process's task_struct but the semaphore array identifier is made invalid. In this case the semaphore clean up code simply discards the sem_undo data structure.

13.8.6 Shared Memory

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.

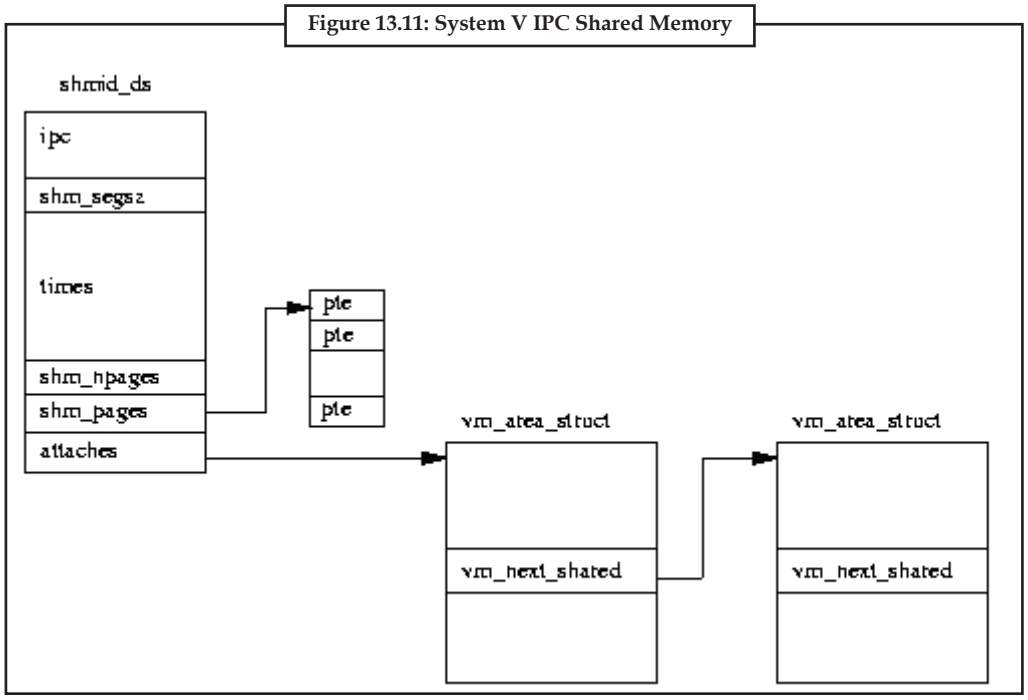
Each newly created shared memory area is represented by a shmid_ds data structure. These are kept in the shm_segs vector.

The shmid_ds data structure describes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

Notes

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a new vm_area_struct data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new vm_area_struct structure is put into the list of vm_area_struct pointed at by the shmid_ds. The vm_next_shared and vm_prev_shared pointers are used to link them together. The virtual memory is not actually created during the attach; it happens when the first process attempts to access it.

Figure 13.11: System V IPC Shared Memory



The first time that a process accesses one of the pages of the shared virtual memory, a page fault will occur. When Linux fixes up that page fault it finds the vm_area_struct data structure describing it. This contains pointers to handler routines for this type of shared virtual memory. The shared memory page fault handling code looks in the list of page table entries for this shmid_ds to see if one exists for this page of the shared virtual memory. If it does not exist, it will allocate a physical page and create a page table entry for it. As well as going into the current process's page tables, this entry is saved in the shmid_ds. This means that when the next process that attempts to access this memory gets a page fault, the shared memory fault handling code will use this newly created physical page for that process too. So, the first process that accesses a page of the shared memory causes it to be created and thereafter access by the other processes cause that page to be added into their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its vm_area_struct is removed from the shmid_ds data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the shmid_ds data structure for this shared memory.

Further complications arise when shared virtual memory is not locked into physical memory. In this case the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage.

Notes

13.9 Network Structure

Networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code, and Linux itself is often used to support the networking needs of organizations. This unit describes how Linux supports the network protocols known collectively as TCP/IP.

The TCP/IP protocols were designed to support communications between computers connected to the ARPANET, an American research network funded by the US government. The ARPANET pioneered networking concepts such as packet switching and protocol layering where one protocol uses the services of another. ARPANET was retired in 1988 but its successors (NSF NET and the Internet) have grown even larger. What is now known as the World Wide Web grew from the ARPANET and is itself supported by the TCP/IP protocols. Unix TM was extensively used on the ARPANET and the first released networking version of Unix TM was 4.3 BSD. Linux's networking implementation is modeled on 4.3 BSD in that it supports BSD sockets (with some extensions) and the full range of TCP/IP networking. This programming interface was chosen because of its popularity and to help applications be portable between Linux and other Unix TM platforms.

13.9.1 An Overview of TCP/IP Networking

In an IP network every machine is assigned an IP address, this is a 32 bit number that uniquely identifies the machine. The WWW is a very large, and growing, IP network and every machine that is connected to it has to have a unique IP address assigned to it. IP addresses are represented by four numbers separated by dots, for example, 16.42.0.9. This IP address is actually in two parts, the network address and the host address. The sizes of these parts may vary (there are several classes of IP addresses) but using 16.42.0.9 as an example, the network address would be 16.42 and the host address 0.9. The host address is further subdivided into a subnetwork and a host address. Again, using 16.42.0.9 as an example, the subnetwork address would be 16.42.0 and the host address 16.42.0.9. This subdivision of the IP address allows organizations to subdivide their networks. For example, 16.42 could be the network address of the ACME Computer Company; 16.42.0 would be subnet 0 and 16.42.1 would be subnet 1. These subnets might be in separate buildings, perhaps connected by leased telephone lines or even microwave links. IP addresses are assigned by the network administrator and having IP subnetworks is a good way of distributing the administration of the network. IP subnet administrators are free to allocate IP addresses within their IP subnetworks.

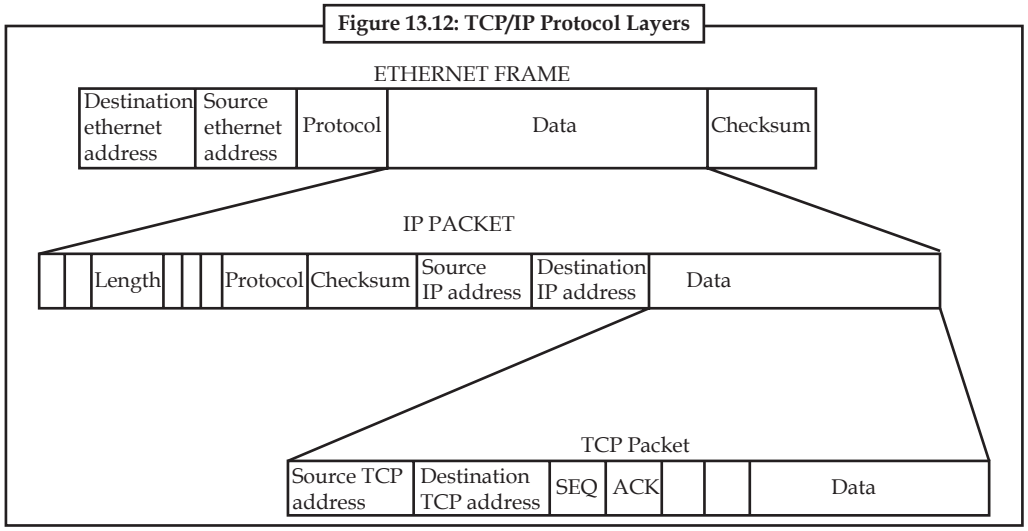
Generally though, IP addresses are somewhat hard to remember. Names are much easier. linux.acme.com is much easier to remember than 16.42.0.9 but there must be some mechanism to convert the network names into an IP address. These names can be statically specified in the /etc/hosts file or Linux can ask a Distributed Name Server (DNS server) to resolve the name for it. In this case the local host must know the IP address of one or more DNS servers and these are specified in /etc/resolv.conf.

Whenever you connect to another machine, say when reading a web page, its IP address is used to exchange data with that machine. This data is contained in IP packets each of which have an IP header containing the IP addresses of the source and destination machine's IP addresses, a checksum and other useful information. The checksum is derived from the data in the IP packet and allows the receiver of IP packets to tell if the IP packet was corrupted during transmission, perhaps by a noisy telephone line. The data transmitted by an application may have been broken down into smaller packets which are easier to handle. The size of the IP data packets varies depending on the connection media; ethernet packets are generally bigger than PPP packets. The destination host must reassemble the data packets before giving the data to the receiving application. You can see this fragmentation and reassembly of data graphically if you access a web page containing a lot of graphical images via a moderately slow serial link.

Notes

Hosts connected to the same IP subnet can send IP packets directly to each other, all other IP packets will be sent to a special host, a gateway. Gateways (or routers) are connected to more than one IP subnet and they will resend IP packets received on one subnet, but destined for another onwards. For example, if subnets 16.42.1.0 and 16.42.0.0 are connected together by a gateway then any packets sent from subnet 0 to subnet 1 would have to be directed to the gateway so that it could route them. The local host builds up routing tables which allow it to route IP packets to the correct machine. For every IP destination there is an entry in the routing tables which tells Linux which host to send IP packets to in order that they reach their destination. These routing tables are dynamic and change over time as applications use the network and as the network topology changes.

Figure 13.12: TCP/IP Protocol Layers



The IP protocol is a transport layer that is used by other protocols to carry their data. The Transmission Control Protocol (TCP) is a reliable end to end protocol that uses IP to transmit and receive its own packets. Just as IP packets have their own header, TCP has its own header. TCP is a connection based protocol where two networking applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. TCP reliably transmits and receives data between the two applications and guarantees that there will be no lost or duplicated data. When TCP transmits its packet using IP, the data contained within the IP packet is the TCP packet itself. The IP layer on each communicating host is responsible for transmitting and receiving IP packets. User Datagram Protocol (UDP) also uses the IP layer to transport its packets, unlike TCP, UDP is not a reliable protocol but offers a datagram service. This use of IP by other protocols means that when IP packets are received the receiving IP layer must know which upper protocol layer to give the data contained in this IP packet to. To facilitate this every IP packet header has a byte containing a protocol identifier. When TCP asks the IP layer to transmit an IP packet , that IP packet's header states that it contains a TCP packet. The receiving IP layer uses that protocol identifier to decide which layer to pass the received data up to, in this case the TCP layer. When applications communicate via TCP/IP they must specify not only the target's IP address but also the port address of the application. A port address uniquely identifies an application and standard network applications use standard port addresses; for example, web servers use port 80. These registered port addresses can be seen in /etc/services.

This layering of protocols does not stop with TCP, UDP and IP. The IP protocol layer itself uses many different physical media to transport IP packets to other IP hosts. These media may themselves add their own protocol headers. One such example is the ethernet layer, but PPP and SLIP are others. An ethernet network allows many hosts to be simultaneously connected to a single physical cable. Every transmitted ethernet frame can be seen by all connected hosts and so

Notes

every ethernet device has a unique address. Any ethernet frame transmitted to that address will be received by the addressed host but ignored by all the other hosts connected to the network. These unique addresses are built into each ethernet device when they are manufactured and it is usually kept in an SROM on the ethernet card. Ethernet addresses are 6 bytes long, an example would be 08-00-2b-00-49-A4. Some ethernet addresses are reserved for multicast purposes and ethernet frames sent with these destination addresses will be received by all hosts on the network. As ethernet frames can carry many different protocols (as data) they, like IP packets, contain a protocol identifier in their headers. This allows the ethernet layer to correctly receive IP packets and to pass them onto the IP layer.

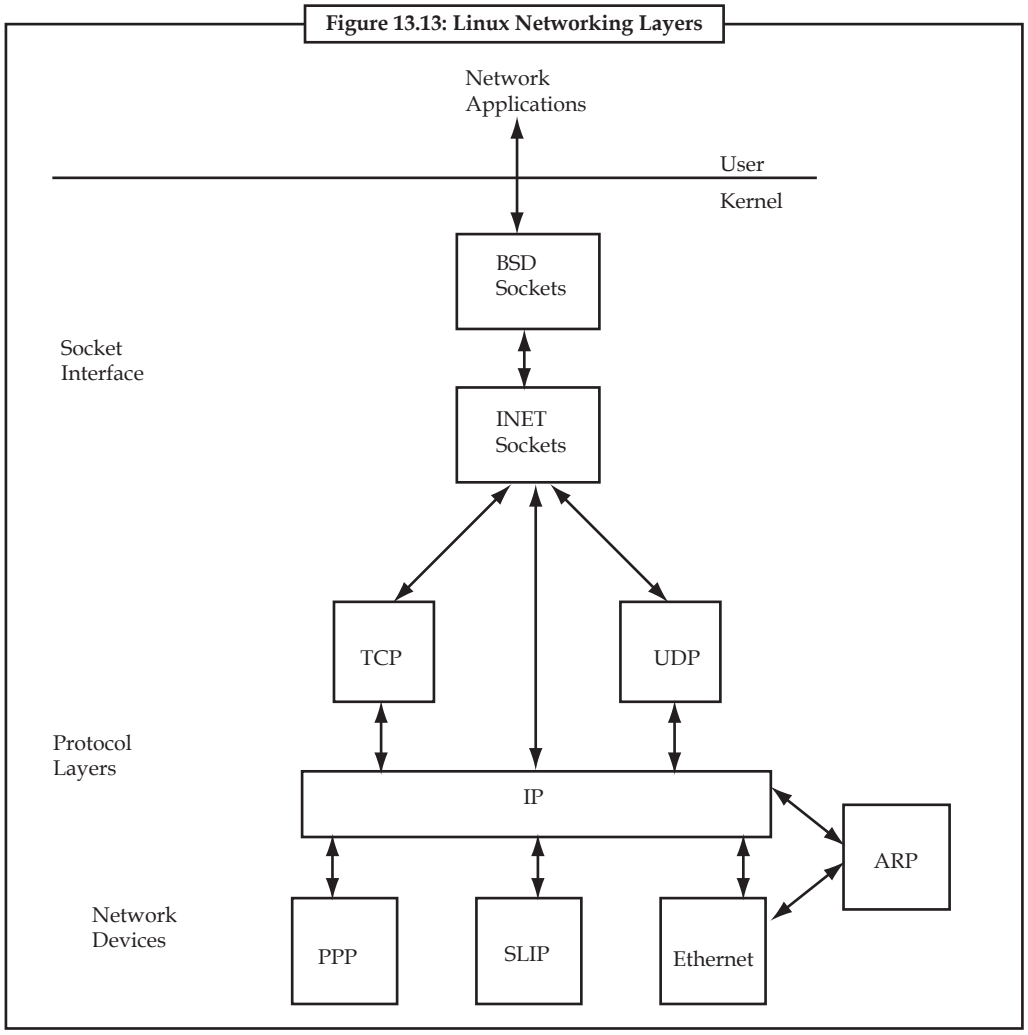
In order to send an IP packet via a multi-connection protocol such as ethernet, the IP layer must find the ethernet address of the IP host. This is because IP addresses are simply an addressing concept, the ethernet devices themselves have their own physical addresses. IP addresses on the other hand can be assigned and reassigned by network administrators at will but the network hardware responds only to ethernet frames with its own physical address or to special multicast addresses which all machines must receive. Linux uses the Address Resolution Protocol (or ARP) to allow machines to translate IP addresses into real hardware addresses such as ethernet addresses. A host wishing to know the hardware address associated with an IP address sends an ARP request packet containing the IP address that it wishes translating to all nodes on the network by sending it to a multicast address. The target host that owns the IP address, responds with an ARP reply that contains its physical hardware address. ARP is not just restricted to ethernet devices, it can resolve IP addresses for other physical media, for example FDDI. Those network devices that cannot ARP are marked so that Linux does not attempt to ARP. There is also the reverse function, Reverse ARP or RARP, which translates physical network addresses into IP addresses. This is used by gateways, which respond to ARP requests on behalf of IP addresses that are in the remote network.

13.9.2 The Linux TCP/IP Networking Layers

Just like the network protocols themselves, Figure 13.13 shows that Linux implements the internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. UDP (User Datagram Protocol) is a connectionless protocol whereas TCP (Transmission Control Protocol) is a reliable end to end protocol. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code prepends IP headers to transmitted data and understands how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux's networking are the network devices, for example PPP and ethernet. Network devices do not always represent physical devices; some like the loopback device are purely software devices. Unlike standard Linux devices that are created via the mknod command, network devices appear only if the underlying software has found and initialized them. You will only see /dev/eth0 when you have built a kernel with the appropriate ethernet device driver in it. The ARP protocol sits between the IP layer and the protocols that support ARPing for addresses.

Notes

Figure 13.13: Linux Networking Layers



13.9.3 The BSD Socket Interface

This is a general interface which not only supports various forms of networking but is also an inter-process communications mechanism. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. Sockets could be thought of as a special case of pipes but, unlike pipes, sockets have no limit on the amount of data that they can contain. Linux supports several classes of socket and these are known as address families. This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets,
INET	The Internet address family supports communications via
	TCP/IP protocols
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

Notes

There are several socket types and these represent the type of service that supports the connection. Not all address families support all types of service. Linux BSD sockets support a number of socket types:

Stream: These sockets provide reliable two way sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the Internet (INET) address family.

Datagram: These sockets also provide two way data transfer but, unlike stream sockets, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported by the UDP protocol of the Internet address family.

Raw: This allows processes direct (hence ''raw'') access to the underlying protocols. It is, for example, possible to open a raw socket to an ethernet device and see raw IP data traffic.

Reliable Delivered Messages: These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets: These are like stream sockets except that the data packet sizes are fixed.

Packet: This is not a standard BSD socket type, it is a Linux specific extension that allows processes to access packets directly at the device level.

Processes that communicate using sockets use a client server model. A server provides a service and clients make use of that service. One example would be a Web Server, which provides web pages and a web client, or browser, which reads those pages. A server using sockets, first creates a socket and then binds a name to it. The format of this name is dependent on the socket's address family and it is, in effect, the local address of the server. The socket's name or address is specified using the sockaddr data structure. An INET socket would have an IP port address bound to it. The registered port numbers can be seen in /etc/services; for example, the port number for a web server is 80. Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. The originator of the request, the client, creates a socket and makes a connection request on it, specifying the target address of the server. For an INET socket the address of the server is its IP address and its port number. These incoming requests must find their way up through the various protocol layers and then wait on the server's listening socket. Once the server has received the incoming request it either accepts or rejects it. If the incoming request is to be accepted, the server must create a new socket to accept it on. Once a socket has been used for listening for incoming connection requests it cannot be used to support a connection. With the connection established both ends are free to send and receive data. Finally, when the connection is no longer needed it can be shutdown. Care is taken to ensure that data packets in transit are correctly dealt with.

The exact meaning of operations on a BSD socket depends on its underlying address family. Setting up TCP/IP connections is very different from setting up an amateur radio X.25 connection. Like the virtual filesystem, Linux abstracts the socket interface with the BSD socket layer being concerned with the BSD socket interface to the application programs which is in turn supported by independent address family specific software. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family. This association is made via cross-linking data structures and tables of address family specific support routines. For example there is an address family specific socket creation routine which the BSD socket interface uses when an application creates a new socket.

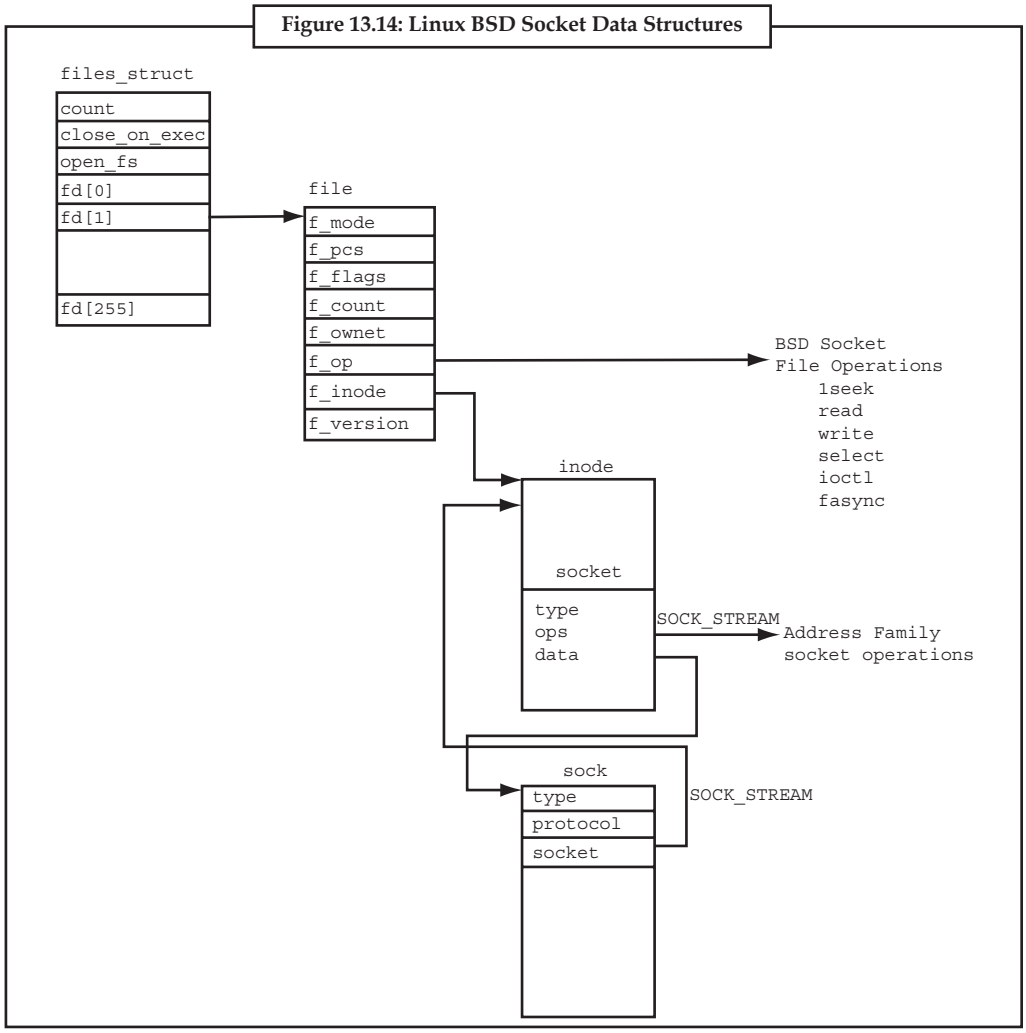
When the kernel is configured, a number of address families and protocols are built into the protocols vector. Each is represented by its name, for example ``INET'' and the address of its initialization routine. When the socket interface is initialized at boot time each protocol's initialization routine is called. For the socket address families this results in them registering a set

Notes

of protocol operations. This is a set of routines, each of which performs a a particular operation specific to that address family. The registered protocol operations are kept in the pops vector, a vector of pointers to proto_ops data structures.

The proto_ops data structure consists of the address family type and a set of pointers to socket operation routines specific to a particular address family. The pops vector is indexed by the address family identifier, for example the Internet address family identifier (AF_INET is 2).

Figure 13.14: Linux BSD Socket Data Structures



13.9.4 The INET Socket Layer

The INET socket layer supports the internet address family which contains the TCP/IP protocols. As discussed above, these protocols are layered, one protocol using the services of another. Linux's TCP/IP code and data structures reflect this layering. Its interface with the BSD socket layer is through the set of Internet address family socket operations which it registers with the BSD socket layer during network initialization. These are kept in the pops vector along with the other registered address families. The BSD socket layer calls the INET layer socket support routines from the registered INET proto_ops data structure to perform work for it. For example a BSD socket create request that gives the address family as INET will use the underlying INET socket create function. The BSD socket layer passes the socket data structure representing the BSD socket to the INET layer in each of these operations. Rather than clutter the BSD socket wiht TCP/IP specific information, the INET socket layer uses its own data structure, the sock which it

Notes

links to the BSD socket data structure. This linkage can be seen in Figure 13.14. It links the sock data structure to the BSD socket data structure using the data pointer in the BSD socket. This means that subsequent INET socket calls can easily retrieve the sock data structure. The sock data structure's protocol operations pointer is also set up at creation time and it depends on the protocol requested. If TCP is requested, then the sock data structure's protocol operations pointer will point to the set of TCP protocol operations needed for a TCP connection.

Creating a BSD Socket

The system call to create a new socket passes identifiers for its address family, socket type and protocol.

Firstly the requested address family is used to search the pops vector for a matching address family. It may be that a particular address family is implemented as a kernel module and, in this case, the kerneld daemon must load the module before we can continue. A new socket data structure is allocated to represent the BSD socket. Actually the socket data structure is physically part of the VFS inode data structure and allocating a socket really means allocating a VFS inode. This may seem strange unless you consider that sockets can be operated on in just the same way that ordinary files can. As all files are represented by a VFS inode data structure, then in order to support file operations, BSD sockets must also be represented by a VFS inode data structure.

The newly created BSD socket data structure contains a pointer to the address family specific socket routines and this is set to the proto_ops data structure retrieved from the pops vector. Its type is set to the socket type requested; one of SOCK_STREAM, SOCK_DGRAM and so on. The address family specific creation routine is called using the address kept in the proto_ops data structure.

A free file descriptor is allocated from the current processes fd vector and the file data structure that it points at is initialized. This includes setting the file operations pointer to point to the set of BSD socket file operations supported by the BSD socket interface. Any future operations will be directed to the socket interface and it will in turn pass them to the supporting address family by calling its address family operation routines.

Binding an Address to an INET BSD Socket

In order to be able to listen for incoming internet connection requests, each server must create an INET BSD socket and bind its address to it. The bind operation is mostly handled within the INET socket layer with some support from the underlying TCP and UDP protocol layers. The socket having an address bound to cannot be being used for any other communication. This means that the socket's state must be TCP_CLOSE. The sockaddr pass to the bind operation contains the IP address to be bound to and, optionally, a port number. Normally the IP address bound to would be one that has been assigned to a network device that supports the INET address family and whose interface is up and able to be used. You can see which network interfaces are currently active in the system by using the ifconfig command. The IP address may also be the IP broadcast address of either all 1's or all 0's. These are special addresses that mean ``send to everybody''. The IP address could also be specified as any IP address if the machine is acting as a transparent proxy or firewall, but only processes with superuser privileges can bind to any IP address. The IP address bound to is saved in the sock data structure in the recv_addr and saddr fields. These are used in hash lookups and as the sending IP address respectively. The port number is optional and if it is not specified the supporting network is asked for a free one. By convention, port numbers less than 1024 cannot be used by processes without superuser privileges. If the underlying network does allocate a port number it always allocates ones greater than 1024.

As packets are being received by the underlying network devices they must be routed to the correct INET and BSD sockets so that they can be processed. For this reason UDP and TCP

Notes

maintain hash tables which are used to lookup the addresses within incoming IP messages and direct them to the correct socket/sock pair. TCP is a connection oriented protocol and so there is more information involved in processing TCP packets than there is in processing UDP packets.

UDP maintains a hash table of allocated UDP ports, the udp_hash table. This consists of pointers to sock data structures indexed by a hash function based on the port number. As the UDP hash table is much smaller than the number of permissible port numbers (udp_hash is only 128 or UDP_HTABLE_SIZE entries long) some entries in the table point to a chain of sock data structures linked together using each sock's next pointer.

TCP is much more complex as it maintains several hash tables. However, TCP does not actually add the binding sock data stucture into its hash tables during the bind operation, it merely checks that the port number requested is not currently being used. The sock data structure is added to TCP's hash tables during the listen operation.

Making a Connection on an INET BSD Socket

Once a socket has been created and, provided it has not been used to listen for inbound connection requests, it can be used to make outbound connection requests. For connectionless protocols like UDP this socket operation does not do a whole lot but for connection orientated protocols like TCP it involves building a virtual circuit between two applications.

An outbound connection can only be made on an INET BSD socket that is in the right state; that is to say one that does not already have a connection established and one that is not being used for listening for inbound connections. This means that the BSD socket data structure must be in state SS_UNCONNECTED. The UDP protocol does not establish virtual connections between applications, any messages sent are datagrams, one off messages that may or may not reach their destinations. It does, however, support the connect BSD socket operation. A connection operation on a UDP INET BSD socket simply sets up the addresses of the remote application; its IP address and its IP port number. Additionally it sets up a cache of the routing table entry so that UDP packets sent on this BSD socket do not need to check the routing database again (unless this route becomes invalid). The cached routing information is pointed at from the ip_route_cache pointer in the INET sock data structure. If no addressing information is given, this cached routing and IP addressing information will be automatically be used for messages sent using this BSD socket. UDP moves the sock's state to TCP_ESTABLISHED.

For a connect operation on a TCP BSD socket, TCP must build a TCP message containing the connection information and send it to IP destination given. The TCP message contains information about the connection, a unique starting message sequence number, the maximum sized message that can be managed by the initiating host, the transmit and receive window size and so on. Within TCP all messages are numbered and the initial sequence number is used as the first message number. Linux chooses a reasonably random value to avoid malicious protocol attacks. Every message transmitted by one end of the TCP connection and successfully received by the other is acknowledged to say that it arrived successfully and uncorrupted. Unacknowledges messages will be retransmitted. The transmit and receive window size is the number of outstanding messages that there can be without an acknowledgement being sent. The maximum message size is based on the network device that is being used at the initiating end of the request. If the receiving end's network device supports smaller maximum message sizes then the connection will use the minimum of the two. The application making the outbound TCP connection request must now wait for a response from the target application to accept or reject the connection request. As the TCP sock is now expecting incoming messages, it is added to the tcp_listening_hash so that incoming TCP messages can be directed to this sock data structure. TCP also starts timers so that the outbound connection request can be timed out if the target application does not respond to the request.

Notes

Listening on an INET BSD Socket

Once a socket has had an address bound to it, it may listen for incoming connection requests specifying the bound addresses. A network application can listen on a socket without first binding an address to it; in this case the INET socket layer finds an unused port number (for this protocol) and automatically binds it to the socket. The listen socket function moves the socket into state TCP_LISTEN and does any network specific work needed to allow incoming connections.

For UDP sockets, changing the socket's state is enough but TCP now adds the socket's sock data structure into two hash tables as it is now active. These are the tcp_bound_hash table and the tcp_listening_hash. Both are indexed via a hash function based on the IP port number.

Whenever an incoming TCP connection request is received for an active listening socket, TCP builds a new sock data structure to represent it. This sock data structure will become the bottom half of the TCP connection when it is eventually accepted. It also clones the incoming sk_buff containing the connection request and queues it onto the receive_queue for the listening sock data structure. The clone sk_buff contains a pointer to the newly created sock data structure.

Accepting Connection Requests

UDP does not support the concept of connections, accepting INET socket connection requests only applies to the TCP protocol as an accept operation on a listening socket causes a new socket data structure to be cloned from the original listening socket. The accept operation is then passed to the supporting protocol layer, in this case INET to accept any incoming connection requests. The INET protocol layer will fail the accept operation if the underlying protocol, say UDP, does not support connections. Otherwise the accept operation is passed through to the real protocol, in this case TCP. The accept operation can be either blocking or non-blocking. In the non-blocking case if there are no incoming connections to accept, the accept operation will fail and the newly created socket data structure will be thrown away. In the blocking case the network application performing the accept operation will be added to a wait queue and then suspended until a TCP connection request is received. Once a connection request has been received the sk_buff containing the request is discarded and the sock data structure is returned to the INET socket layer where it is linked to the new socket data structure created earlier. The file descriptor (fd) number of the new socket is returned to the network application, and the application can then use that file descriptor in socket operations on the newly created INET BSD socket.

13.9.5 The IP Layer

Socket Buffers

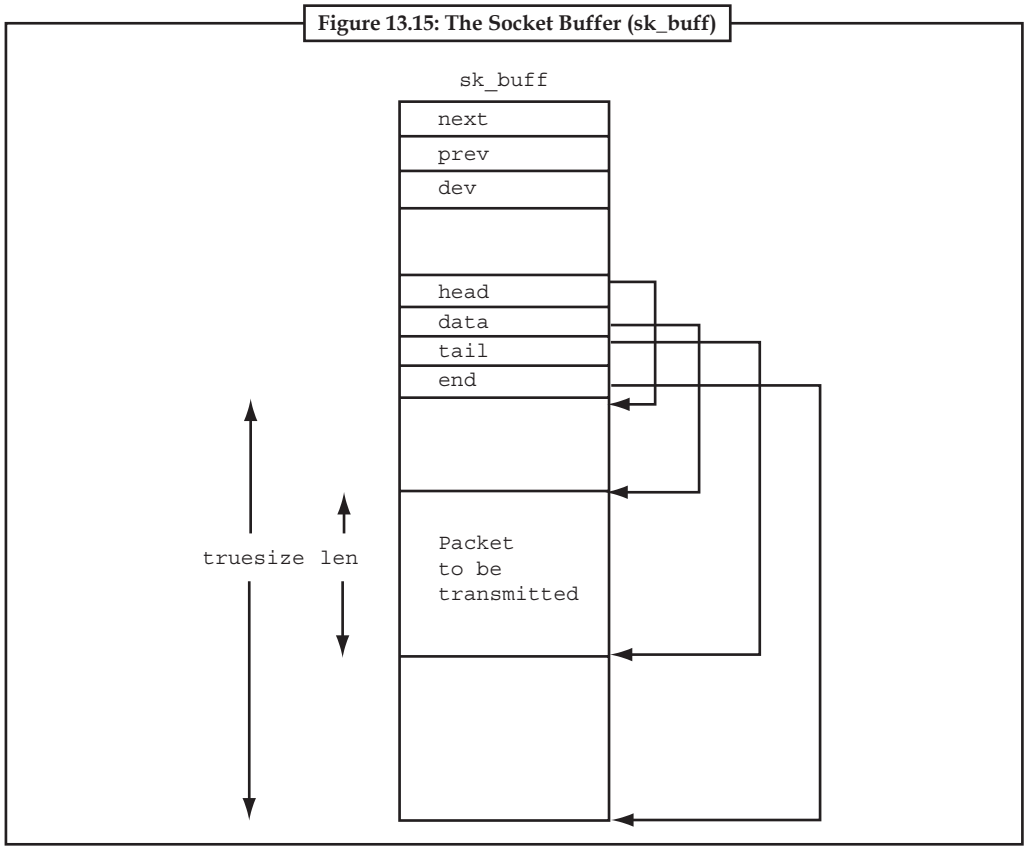
One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or sk_buffs to pass data between the protocol layers and the network device drivers. sk_buffs contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or ``methods''.

Figure 13.15 shows the sk_buff data structure; each sk_buff has a block of data associated with it. The sk_buff has four data pointers, which are used to manipulate and manage the socket buffer's data:

Head: Points to the start of the data area in memory. This is fixed when the sk_buff and its associated data block is allocated.

Notes

Figure 13.15: The Socket Buffer (sk_buff)



Data: Points at the current start of the protocol data. This pointer varies depending on the protocol layer that currently owns the sk_buff.

Tail: Points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer,

End: Points at the end of the data area in memory. This is fixed when the sk_buff is allocated.

There are two length fields len and truesize, which describe the length of the current protocol packet and the total size of the data buffer respectively. The sk_buff handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the data, tail and len fields in the sk_buff:

Push: This moves the data pointer towards the start of the data area and increments the len field. This is used when adding data or protocol headers to the start of the data to be transmitted,

Pull: This moves the data pointer away from the start, towards the end of the data area and decrements the len field. This is used when removing data or protocol headers from the start of the data that has been received.

Put: This moves the tail pointer towards the end of the data area and increments the len field. This is used when adding data or protocol information to the end of the data to be transmitted,

Trim: This moves the tail pointer towards the start of the data area and decrements the len field. This is used when removing data or protocol tails from the received packet.

The sk_buff data structure also contains pointers that are used as it is stored in doubly linked circular lists of sk_buff's during processing. There are generic sk_buff routines for adding sk_buffs to the front and back of these lists and for removing them.

Notes

Receiving IP Packets

Each device data structure describes its device and provides a set of callback routines that the network protocol layers call when they need the network driver to perform work. These functions are mostly concerned with transmitting data and with the network device's addresses. When a network device receives packets from its network it must convert the received data into sk_buff data structures. These received sk_buff's are added onto the backlog queue by the network drivers as they are received.

If the backlog queue grows too large, then the received sk_buff's are discarded. The network bottom half is flagged as ready to run as there is work to do.

When the network bottom half handler is run by the scheduler it processes any network packets waiting to be transmitted before processing the backlog queue of sk_buff's determining which protocol layer to pass the received packets to.

As the Linux networking layers were initialized, each protocol registered itself by adding a packet_type data structure onto either the ptype_all list or into the ptype_base hash table. The packet_type data structure contains the protocol type, a pointer to a network device, a pointer to the protocol's receive data processing routine and, finally, a pointer to the next packet_type data structure in the list or hash chain. The ptype_all chain is used to snoop all packets being received from any network device and is not normally used. The ptype_base hash table is hashed by protocol identifier and is used to decide which protocol should receive the incoming network packet. The network bottom half matches the protocol types of incoming sk_buff's against one or more of the packet_type entries in either table. The protocol may match more than one entry, for example when snooping all network traffic, and in this case the sk_buff will be cloned. The sk_buff is passed to the matching protocol's handling routine.

Sending IP Packets

Packets are transmitted by applications exchanging data or else they are generated by the network protocols as they support established connections or connections being established. Whichever way the data is generated, an sk_buff is built to contain the data and various headers are added by the protocol layers as it passes through them.

The sk_buff needs to be passed to a network device to be transmitted. First though the protocol, for example IP, needs to decide which network device to use. This depends on the best route for the packet. For computers connected by modem to a single network, say via the PPP protocol, the routing choice is easy. The packet should either be sent to the local host via the loopback device or to the gateway at the end of the PPP modem connection. For computers connected to an ethernet the choices are harder as there are many computers connected to the network.

For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address. Each IP destination successfully looked up in the routing tables returns a rtable data structure describing the route to use. This includes the source IP address to use, the address of the network device data structure and, sometimes, a prebuilt hardware header. This hardware header is network device specific and contains the source and destination physical addresses and other media specific information. If the network device is an ethernet device, the hardware header would be as shown in Figure 13 and the source and destination addresses would be physical ethernet addresses. The hardware header is cached with the route because it must be appended to each IP packet transmitted on this route and constructing it takes time. The hardware header may contain physical addresses that have to be resolved using the ARP protocol. In this case the outgoing packet is stalled until the address has been resolved. Once it has been resolved and the hardware header built, the hardware header is cached so that future IP packets sent using this interface do not have to ARP.

Notes

Data Fragmentation

Every network device has a maximum packet size and it cannot transmit or receive a data packet bigger than this. The IP protocol allows for this and will fragment data into smaller units to fit into the packet size that the network device can handle. The IP protocol header includes a fragment field which contains a flag and the fragment offset.

When an IP packet is ready to be transmited, IP finds the network device to send the IP packet out on. This device is found from the IP routing tables. Each device has a field describing its maximum transfer unit (in bytes), this is the mtu field. If the device's mtu is smaller than the packet size of the IP packet that is waiting to be transmitted, then the IP packet must be broken down into smaller (mtu sized) fragments. Each fragment is represented by an sk_buff; its IP header marked to show that it is a fragment and what offset into the data this IP packet contains. The last packet is marked as being the last IP fragment. If, during the fragmentation, IP cannot allocate an sk_buff, the transmit will fail.

Receiving IP fragments is a little more difficult than sending them because the IP fragments can be received in any order and they must all be received before they can be reassembled. Each time an IP packet is received it is checked to see if it is an IP fragment. The first time that the fragment of a message is received, IP creates a new ipq data structure, and this is linked into the ipqueue list of IP fragments awaiting recombination. As more IP fragments are received, the correct ipq data structure is found and a new ipfrag data structure is created to describe this fragment. Each ipq data structure uniquely describes a fragmented IP receive frame with its source and destination IP addresses, the upper layer protocol identifier and the identifier for this IP frame. When all of the fragments have been received, they are combined into a single sk_buff and passed up to the next protocol level to be processed. Each ipq contains a timer that is restarted each time a valid fragment is received. If this timer expires, the ipq data structure and its ipfrag's are dismantled and the message is presumed to have been lost in transit. It is then up to the higher level protocols to retransmit the message.

13.9.6 The Address Resolution Protocol (ARP)

The Address Resolution Protocol's role is to provide translations of IP addresses into physical hardware addresses such as ethernet addresses. IP needs this translation just before it passes the data (in the form of an sk_buff) to the device driver for transmission.

It performs various checks to see if this device needs a hardware header and, if it does, if the hardware header for the packet needs to be rebuilt. Linux caches hardware headers to avoid frequent rebuilding of them. If the hardware header needs rebuilding, it calls the device specific hardware header rebuilding routine. All ethernet devices use the same generic header rebuilding routine which in turn uses the ARP services to translate the destination IP address into a physical address.

The ARP protocol itself is very simple and consists of two message types, an ARP request and an ARP reply. The ARP request contains the IP address that needs translating and the reply (hopefully) contains the translated IP address, the hardware address. The ARP request is broadcast to all hosts connected to the network, so, for an ethernet network, all of the machines connected to the ethernet will see the ARP request. The machine that owns the IP address in the request will respond to the ARP request with an ARP reply containing its own physical address.

The ARP protocol layer in Linux is built around a table of arp_table data structures which each describe an IP to physical address translation. These entries are created as IP addresses need to

Notes

be translated and removed as they become stale over time. Each arp_table data structure has the following fields:

last used	the time that this ARP entry was last used,
last updated	the time that this ARP entry was last updated,
flags	these describe this entry's state, if it is complete and so on,
IP address	The IP address that this entry describes
hardware address	The translated hardware address
hardware header	This is a pointer to a cached hardware header,
timer	This is a timer_list entry used to time out ARP requests that do not get a response,
retries	The number of times that this ARP request has been retried,
sk_buff queue	List of sk_buff entries waiting for this IP address to be resolved

The ARP table consists of a table of pointers (the arp_tables vector) to chains of arp_table entries. The entries are cached to speed up access to them, each entry is found by taking the last two bytes of its IP address to generate an index into the table and then following the chain of entries until the correct one is found. Linux also caches prebuilt hardware headers off the arp_table entries in the form of hh_cache data structures.

When an IP address translation is requested and there is no corresponding arp_table entry, ARP must send an ARP request message. It creates a new arp_table entry in the table and queues the sk_buff containing the network packet that needs the address translation on the sk_buff queue of the new entry. It sends out an ARP request and sets the ARP expiry timer running. If there is no response then ARP will retry the request a number of times and if there is still no response ARP will remove the arp_table entry. Any sk_buff data structures queued waiting for the IP address to be translated will be notified and it is up to the protocol layer that is transmitting them to cope with this failure. UDP does not care about lost packets but TCP will attempt to retransmit on an established TCP link. If the owner of the IP address responds with its hardware address, the arp_table entry is marked as complete and any queued sk_buff's will be removed from the queue and will go on to be transmitted. The hardware address is written into the hardware header of each sk_buff.

The ARP protocol layer must also respond to ARP requests that specfy its IP address. It registers its protocol type (ETH_P_ARP), generating a packet_type data structure. This means that it will be passed all ARP packets that are received by the network devices. As well as ARP replies, this includes ARP requests. It generates an ARP reply using the hardware address kept in the receiving device's device data structure.

Network topologies can change over time and IP addresses can be reassigned to different hardware addresses. For example, some dial up services assign an IP address as each connection is established. In order that the ARP table contains up to date entries, ARP runs a periodic timer which looks through all of the arp_table entries to see which have timed out. It is very careful not to remove entries that contain one or more cached hardware headers. Removing these entries is dangerous as other data structures rely on them. Some arp_table entries are permanent and these are marked so that they will not be deallocated. The ARP table cannot be allowed to grow too large; each arp_table entry consumes some kernel memory. Whenever the a new entry needs to be allocated and the ARP table has reached its maximum size the table is pruned by searching out the oldest entries and removing them.

13.9.7 IP Routing

The IP routing function determines where to send IP packets destined for a particular IP address. There are many choices to be made when transmitting IP packets. Can the destination be reached at all? If it can be reached, which network device should be used to transmit it? If there is more than one network device that could be used to reach the destination, which is the better one? The

Notes

IP routing database maintains information that gives answers to these questions. There are two databases, the most important being the Forwarding Information Database. This is an exhaustive list of known IP destinations and their best routes. A smaller and much faster database, the route cache is used for quick lookups of routes for IP destinations. Like all caches, it must contain only the frequently accessed routes; its contents are derived from the Forwarding Information Database.

Routes are added and deleted via IOCTL requests to the BSD socket interface. These are passed onto the protocol to process. The INET protocol layer only allows processes with superuser privileges to add and delete IP routes. These routes can be fixed or they can be dynamic and change over time. Most systems use fixed routes unless they themselves are routers. Routers run routing protocols which constantly check on the availability of routes to all known IP destinations. Systems that are not routers are known as end systems. The routing protocols are implemented as daemons, for example GATED, and they also add and delete routes via the IOCTL BSD socket interface.

Route Cache

Whenever an IP route is looked up, the route cache is first checked for a matching route. If there is no matching route in the route cache the Forwarding Information Database is searched for a route. If no route can be found there, the IP packet will fail to be sent and the application notified. If a route is in the Forwarding Information Database and not in the route cache, then a new entry is generated and added into the route cache for this route. The route cache is a table (ip_rt_hash_table) that contains pointers to chains of rtable data structures. The index into the route table is a hash function based on the least significant two bytes of the IP address. These are the two bytes most likely to be different between destinations and provide the best spread of hash values. Each rtable entry contains information about the route; the destination IP address, the network device to use to reach that IP address, the maximum size of message that can be used and so on. It also has a reference count, a usage count and a timestamp of the last time that they were used (in jiffies). The reference count is incremented each time the route is used to show the number of network connections using this route. It is decremented as applications stop using the route. The usage count is incremented each time the route is looked up and is used to order the rtable entry in its chain of hash entries. The last used timestamp for all of the entries in the route cache is periodically checked to see if the rtable is too old. If the route has not been recently used, it is discarded from the route cache. If routes are kept in the route cache they are ordered so that the most used entries are at the front of the hash chains. This means that finding them will be quicker when routes are looked up.

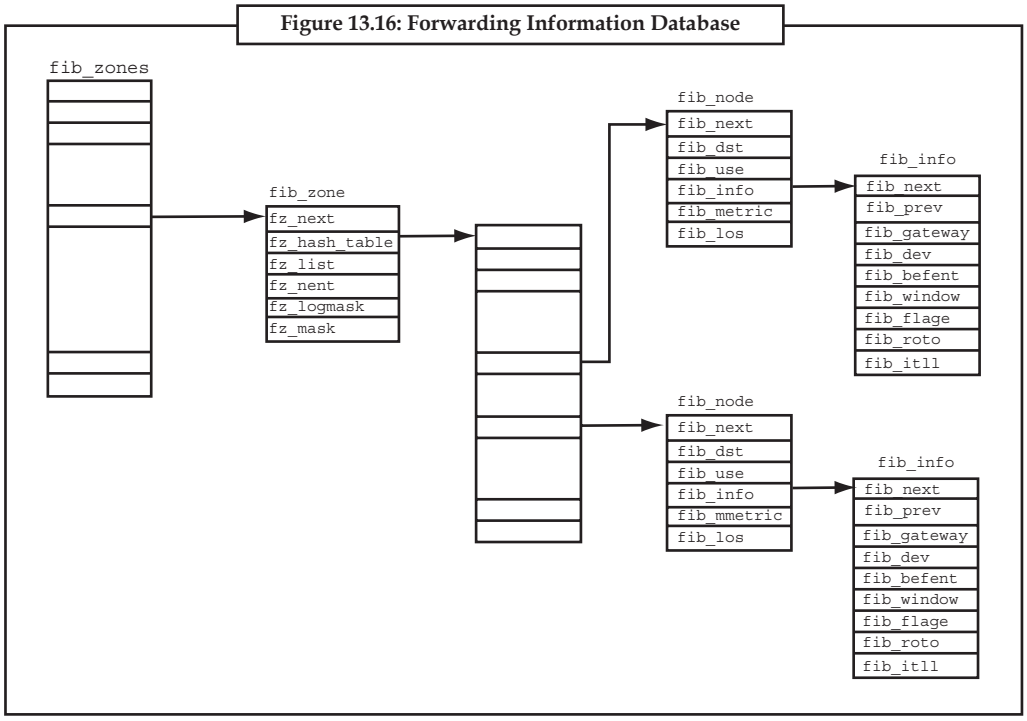
Forwarding Information Database

The forwarding information database (shown in Figure 13.16 contains IP's view of the routes available to this system at this time. It is quite a complicated data structure and, although it is reasonably efficiently arranged, it is not a quick database to consult. In particular it would be very slow to look up destinations in this database for every IP packet transmitted. This is the reason that the route cache exists: to speed up IP packet transmission using known good routes. The route cache is derived from the forwarding database and represents its commonly used entries.

Each IP subnet is represented by a fib_zone data structure. All of these are pointed at from the fib_zones hash table. The hash index is derived from the IP subnet mask. All routes to the same subnet are described by pairs of fib_node and fib_info data structures queued onto the fz_list of each fib_zone data structure. If the number of routes in this subnet grows large, a hash table is generated to make finding the fib_node data structures easier.

Notes

Figure 13.16: Forwarding Information Database



Several routes may exist to the same IP subnet and these routes can go through one of several gateways. The IP routing layer does not allow more than one route to a subnet using the same gateway. In other words, if there are several routes to a subnet, then each route is guaranteed to use a different gateway. Associated with each route is its metric. This is a measure of how advantageous this route is. A route's metric is, essentially, the number of IP subnets that it must hop across before it reaches the destination subnet. The higher the metric, the worse the route.

13.10 Security

Linux, like any computer system, has a set of security issues that need to be considered. Regardless of what mechanisms are in place, the basic concepts are the same. In fact, the security of a computer system is very much like the security of a house, just as running a computer system is like running a household.

A knowledgeable user with root access to another Linux system can gain access to yours if they have physical access. Even without access to another system, if that user has access to the installation floppies, they can get into your system. Once in, it doesn't matter what kind of security is has been configured on the hard disk since the only security the system knows is what it has been told by the floppy.

Regardless of what security issue you are talking about, any breach in security can be prevented by not allowing access to the system. Now, this can be taken to extremes by not letting anyone to have access. However, by limiting access to the system to only authorized users, you substantially lower the risk of breaches in security. Keep in mind that there is no such thing as a secure system. This is especially important when you consider that the most serious threat comes from people who already have an account on that system.

Access control has been a part of Linux for a long time. It is a fundamental aspect of any multi-user system. The most basic form of access control is in the form of user accounts. The only way you should be able to gain access to a Linux system is through an account. Users usually gain access to the system when they have an account set up for them. Each user is assigned an

Notes

individual password that allows the access. Access to files is determined by the permissions that are set on the files and directories.

Access to various services such as printers or even local file systems is also determined by permissions. In addition, some services, as we shall see shortly can be restricted at a much lower level.

In some cases, passwords may be blank, meaning you only need to press enter. In other cases it can be removed altogether so you are never even prompted to input your password. Removing the password may not always be a good idea. Since you have the source code, Linux allows you the option to prevent users from either having no password or having to just press return. Since we are talking here about security and accounts without passwords are not very secure, we'll restrict ourselves to talking about accounts that have passwords.

On many systems (including many Linux versions) one cannot force users to use (or not use) specific passwords. As a system administrator it is your responsibility to not only enforce a strong password policy, but to educate your users as to why this is important. Later, we'll go over some examples of what happens when users are not aware of the issues involved with password security.

Although this password protection stops most attempts to gain unauthorized access to the system, many security issues involve users that already have accounts. Unchecked, curious users could access payroll information and find out what their boss gets paid. Corporate spies could steal company secrets. Disgruntled workers could wreak havoc by destroying data or slowing down the system.

Once logged in, Linux provides a means of limiting the access of "authorized" users. This is in the form of file permissions, which we already talked about. File permissions are one aspect of security that most people are familiar with in regard to UNIX security. In many cases, this is the only kind of security other that user accounts.

Each file has an owner, whether or not some user explicitly went out there and "claimed" ownership. Its a basic characteristic of each file and is imposed upon them by the operating system. The owner of the file is stored, along with other information, in the inode table in the form of a number. This number corresponds to the User ID (UID) number from /etc/password.

Normally, files are initially owned by the user who creates them. However, there are many circumstances that would change the ownership. One of the obvious ways is that the ownership is intentionally changed. Only the owner of the file and root can change its ownership. If you are the owner of a file, you can, in essence, "transfer ownership" of the file to someone else. Once you do, you are no longer the owner (obviously) and have no more control over that file.

Another characteristic of a file is its group. Like the owner, the file's group is an intrinsic part of that files characteristics. The file's group is also stored in the inode as a number. The translation from this number to the group name is made from the /etc/group file. As we talked about in the section on users, the concept of a group has only real meaning in terms of security. That is, who can access which files.

What this means is that only "authorized" users can access files in any of the three manners: read, write and execute. It makes sense that normal users cannot run the fdisk utility, otherwise they would have the ability to re-partition the hard disk, potentially destroying data. It also makes sense that normal users do not have write permission on the/etc/password file, otherwise they could change it so that they would have access to the root account. Since we talked about it in the section on shell basics and on users, there is no need to go into more details here.

There is also access to the all powerful root account. On a Linux system root can do anything. Although it is possible to restrict roots access to certain functions, a knowledgeable user with root privileges can overcome that. There are many instances where you have several people administering some aspect of the system, such as printers or the physical network.

Notes

Access to the root account should be limited for a couple of reasons. First, the more people with root access, the more people who have complete control over the system. This makes access control difficult.

Also, the more people that have root access, the more fingers get pointed. There are people who are going to deny having done something wrong. Often this results in a corrupt system, as there are everyone has the power to do everything, someone did something that messed up the system somehow and no one will admit. Sound familiar?

The fewer people that have root, the fewer fingers need to be pointed and the fewer people can pass the buck. Not that what they did was malicious, mistakes do happen. If there are fewer people with root access and something goes wrong, tracking down the cause is much easier.

Rather than several users all having the root password, some people think that it is safer to create several users all with the UID of root. Their belief is that since there are several lognames, it's easier to keep track of things. Well, the problem in that thinking is that the system keeps track of track of users by the UID. There is no way to keep these users separate, once they log in.

Another security precaution is to define secure terminals. These are the only terminals that the root user can login from. In my opinion, it is best to only consider directly connected terminals as "secure". That is, the root user can log into the system console, but not across the network. To get access as root across the network, a user must first login under their own account and then use su. This also provides a record of who used the root account and when.

If the system is connected to the Internet, such as for a HTTP or FTP server, then security is a primary consideration.

One way of avoiding compromising your system is to have your WWW server connected to the Internet, but not to your internal network. Should someone be able to break into the WWW server, the worst that can happen is that the WWW server is down for a day or so as you reload from backups. If the intruder had access to the internal network, your livelihood could be threatened.

One very common attack is the dictionary attack. Here the hacker uses common words, encrypts them using the same as the password taken from the password file and then the two are compared. Remember that the /etc/passwd file is readable by everyone and the seed is contained within the encrypted password is always the first two characters.

Although this seems to be a major security hole, it is very effective if you use passwords that are not easy to guess. The reason is that the encryption goes only one way.

13.11 Summary

Linux is a modular Unix-like operating system. Linux operates in two modes - the Kernel mode (kernel space) and the User mode (user space). The Linux kernel is a monolithic kernel. Any application that runs on a Linux system is assigned a process ID or PID. This is a numerical representation of the instance of the application on the system. There are generally two types of processes that run on Linux. Interactive processes are those processes that are invoked by a user and can interact with the user. Interactive processes can be classified into foreground and background processes. The foreground process is the process that you are currently interacting with, and is using the terminal as its stdin (standard input) and stdout (standard output). A background process is not interacting with the user and can be in one of two states - paused or running. All processes are derived from the init process and can trace their roots back to init. All processes run partially in user mode and partially in system mode. Virtual memory is used in Linux. It uses the Buddy algorithm to effectively allocate and deallocate blocks of pages. Its developers and users use the web to exchange information ideas, code, and Linux itself is often used to support the networking needs of organizations. This unit describes how Linux supports the network protocols known collectively as TCP/IP.

Notes

13.12 Keywords

Background process: It is a process which is not interacting with the user and can be in one of two states - paused or running.

Foreground process: This is the process that user is currently interacting with, and is using the terminal as its stdin (standard input) and stdout (standard output).

Interactive processes: Those processes that are invoked by a user and can interact with the user.

PID: This is a numerical representation of the instance of the application on the Linux system.

Semaphore: It is a location in memory whose value can be tested and set by more than one process.

13.13 Review Questions

1. Does Linux Support Threads or Lightweight Processes?
2. What is virtual memory? How it is implemented in Linux?
3. Describe the page allocation process in Linux.
4. Write a brief description about the process scheduling in Linux.

13.14 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Notes

Unit 14: Windows 2000

CONTENTS

Objectives

Introduction

14.1 Design Principles

14.2 System Components

14.3 Environmental Sub-systems

14.4 File System

14.5 Networking

14.5.1 Protocols

14.5.2 Distributed-Processing Mechanisms

14.5.3 Redirectors and Servers

14.5.4 Domains

14.5.5 Name Resolution in TCP/IP Networks

14.6 Programmer Interface

14.6.1 Access to Kernel Objects

14.6.2 Inter-process Communication

14.6.3 Memory Management

14.7 Summary

14.8 Keywords

14.9 Review Questions

14.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain design principles
- Describe system components
- Know environmental subsystems
- Define file system
- Describe networking and programmer interface

Introduction

The objective of this unit is to introduce to the Windows 2000 operating system. Windows 2000 (W2K) is a commercial version of Microsoft's evolving Windows operating system. Previously called Windows NT 5.0, Microsoft emphasizes that Windows 2000 is evolutionary and "Built on NT Technology." Windows 2000 is designed to appeal to small business and professional users as well as to the more technical and larger business market for which the NT was designed.

Notes

Windows 2000 (also referred to as Win2K) is a preemptive, interruptible, graphical and business-oriented operating system designed to work with either uni-processor or symmetric multi-processor computers. It is part of the Microsoft Windows NT line of operating systems and was released on February 17, 2000. It was succeeded by Windows XP in October 2001 and Windows Server 2003 in April 2003. Windows 2000 is classified as a hybrid kernel operating system. Key goals for the system are portability, security, Portable Operating System Interface (POSIX) or IEEE Std. 1003.1 compliance, multiprocessor support, extensibility, international support, and compatibility with MS-DOS and MS-Windows applications.

The Windows 2000 product line consists of four products:

1. ***Windows 2000 Professional:*** Supports up to two processors and up to 4GB of RAM. Used as a workstation or client computer and it is the replacement for Windows NT Workstation.
2. ***Windows 2000 Server:*** Supports up to four processors and up to 4GB of RAM. It is used for web, application, print and file servers.
3. ***Windows 2000 Advanced Server:*** Supports up to eight processors and up to 8GB of RAM. It is used in an enterprise network and very useful as an SQL server.
4. ***Windows 2000 Datacenter Server:*** Supports up to 32 processors and up to 64GB of RAM. It is used in an enterprise network to support extremely large databases and real time processing.

In this unit, we discuss the key goals for this system, the layered architecture of the system that makes it so easy to use, the file system, networks and the programming interface.

14.1 Design Principles

The design goals that Microsoft has stated for Windows 2000 include extensibility, portability, reliability, compatibility, performance, and international support.

Extensibility refers to the capacity of an operating system to keep up with advancements in computing technology. So that changes are facilitated over time, the developers implemented Windows 2000 using a layered architecture.

The Windows 2000 executive, which runs in kernel or protected mode, provides the basic system services. On top of the executive, several server subsystems operate in user mode. Among them are environmental subsystems that emulate different operating systems. Thus, programs written for MS-DOS, Microsoft Windows, and POSIX can all run on Windows 2000 in the appropriate environment. Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows 2000 uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows 2000 uses a client-server model like the Mach operating system, and supports distributed processing by remote procedure calls (RPCs) as defined by the Open Software Foundation.

An operating system is portable if it can be moved from one hardware architecture to another with relatively few changes. Windows 2000 is designed to be portable. As is true of the UNIX operating system, the majority of the system is written in C and C++. All processor-dependent code is isolated in a dynamic link library (DLL), called the hardware-abstraction layer (HAL). A DLL is a file that gets mapped into a process's address space such that any functions in the DLL appear as though they are part of the process. The upper layers of Windows 2000 depend on HAL, rather than on the underlying hardware, and that helps Windows 2000 to be portable. HAL manipulates hardware directly, isolating the rest of Windows 2000 from hardware differences among the platforms on which it runs.

Reliability is the ability to handle error conditions, including the ability of the operating system to protect itself and its users from defective or malicious software. Windows 2000 resists defects and

Notes

attacks by using hardware protection for virtual memory, and software protection mechanisms for operating system resources.

Windows 2000 comes with a native file system; the NTFS file system that recovers automatically from many kinds of file-system errors after a system crash. Windows NT Version 4.0 has a C-2 security classification from the U.S. government, which signifies a moderate level of protection from defective software and malicious attacks. Windows 2000 is currently under evaluation by the government for that classification as well.

Windows 2000 provides source-level compatibility to applications that follow the IEEE 1003.1 (POSIX) standard. Thus, they can be compiled to run on Windows 2000 without changes to the source code. In addition, Windows 2000 can run the executable binaries for many programs compiled for Intel X86 architectures running MS-DOS, 16-bitWindows, OS/2, LAN Manager, and 32-bit Windows, by using the environmental subsystems mentioned earlier. These environmental subsystems support a variety of file systems, including the MSDOS FAT file system; the OS/2 HPFS file system, the ISO9660 CD file system, and NTFS. Windows 2000Õs binary compatibility, however, is not perfect. In MS-DOS, for example, applications can access hardware ports directly. For reliability and security, Windows 2000 prohibits such access.

Windows 2000 is designed to afford good performance. The subsystems that constitute Windows 2000 can communicate with one another efficiently by a local-procedure-call facility that provides high-performance message passing.

Except for the kernel, threads in the subsystems of Windows 2000 can be preempted by higher-priority threads. Thus, the system can respond quickly to external events. In addition, Windows 2000 is designed for symmetrical multiprocessing: On a multiprocessor computer, several threads can run at the same time. The current scalability of Windows 2000 is limited, compared to that of UNIX. As of late 2000, Windows 2000 supported systems with up to 32 CPUs, whereas Solaris ran on systems with up to 64 processors. Previous versions of NT supported only up to 8 processors.

Windows 2000 is also designed for international use. It provides support for different locales via the national language support (NLS) API. NLS API provides specialized routines to format dates, time, and money in accordance with various national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows 2000Õs native character code.

Windows 2000 supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion).

14.2 System Components

Windows 2000 is a highly modular system that consists of two main layers: a user mode and a kernel mode. The user mode refers to the mode in which user programs are run. Such programs are limited in terms of what system resources they have access to, while the kernel mode has unrestricted access to the system memory and external devices. All user mode applications access system resources through the executive which runs in kernel mode.

User Mode

User mode in Windows 2000 is made of subsystems capable of passing I/O requests to the appropriate kernel mode drivers by using the I/O manager. Two subsystems make up the user mode layer of Windows 2000: the environment subsystem and the integral subsystem.

The environment subsystem was designed to run applications written for many different types of operating systems. These applications, however, run at a lower priority than kernel mode processes.

Notes

There are three main environment subsystems:

1. Win32 subsystem runs 32-bit Windows applications and also supports Virtual DOS Machines (VDMs) , which allows MS-DOS and 16-bit Windows 3.1x (Win16) applications to run on Windows.
2. OS/2 environment subsystem supports 16-bit character-based OS/2 applications and emulates OS/2 1.3 and 1.x, but not 32-bit or graphical OS/2 applications as used on OS/2 2.x or later.
3. POSIX environment subsystem supports applications that are strictly written to either the POSIX.1 standard or the related ISO/IEC standards.

The integral subsystem looks after operating system specific functions on behalf of the environment subsystem. It consists of a security subsystem (grants/denies access and handles logons), workstation service (helps the computer gain network access) and a server service (lets the computer provide network services).

Kernel Mode

Kernel mode in Windows 2000 has full access to the hardware and system resources of the computer. The kernel mode stops user mode services and applications from accessing critical areas of the operating system that they should not have access to.

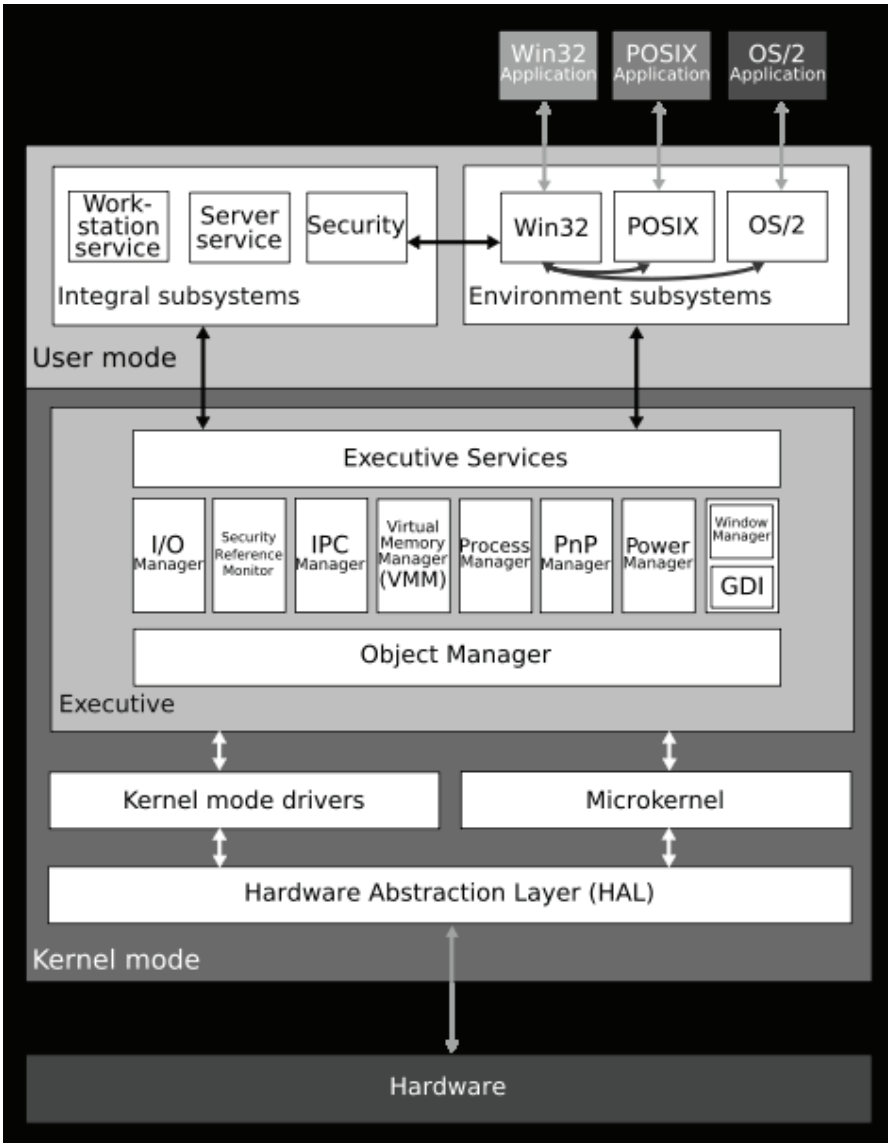
The executive interfaces with all the user mode subsystems. It deals with I/O, object management, security and process management. It contains various components, including:

1. ***Object Manager:*** a special executive subsystem that all other executive subsystems must pass through to gain access to Windows 2000 resources. This is essentially a resource management infrastructure service that allows Windows 2000 to be an object oriented operating system.
2. ***I/O Manager:*** allows devices to communicate with user-mode subsystems by translating user-mode read and write commands and passing them to device drivers.
3. ***Security Reference Monitor (SRM):*** the primary authority for enforcing the security rules of the security integral subsystem.
4. ***IPC Manager:*** short for Inter-Process Communication Manager, manages the communication between clients (the environment subsystem) and servers (components of the executive).
5. ***Virtual Memory Manager:*** manages virtual memory, allowing Windows 2000 to use the hard disk as a primary storage device (although strictly speaking it is secondary storage).
6. ***Process Manager:*** handles process and thread creation and termination.
7. ***PnP Manager:*** handles Plug and Play and supports device detection and installation at boot time.
8. ***Power Manager:*** the power manager coordinates power events and generates power IRPs.
9. The display system is handled by a device driver contained in Win32k.sys. The Window Manager component of this driver is responsible for drawing windows and menus while the GDI (Graphics Device Interface) component is responsible for tasks such as drawing lines and curves, rendering fonts and handling palettes. Windows 2000 also introduced alpha blending into the Graphics Device Interface which reflects in the fade effect in menus.

The Windows 2000 Hardware Abstraction Layer, or HAL, is a layer between the physical hardware of the computer and the rest of the operating system. It was designed to hide differences in hardware and therefore provide a consistent platform to run applications on. The HAL includes hardware specific code that controls I/O interfaces, interrupt controllers and multiple processors.

Notes

Figure 14.1: Windows 2000 Block Diagram



The hybrid kernel sits between the HAL and the executive and provides multiprocessor synchronization, thread and interrupt scheduling and dispatching, trap handling and exception dispatching. The hybrid kernel often interfaces with the process manager and is responsible for initializing device drivers at boot-up that are necessary to get the operating system up and running.

14.3 Environmental Sub-systems

Environmental subsystems are user-mode processes layered over the native Windows 2000 executive services to enable Windows 2000 to run programs developed for other operating systems, including 16-bit Windows, MS-DOS, POSIX, and character-based applications for 16-bit OS/2. Each environmental subsystem provides one API or application environment.

Windows 2000 uses the Win32 subsystem as the main operating environment, and thus to start all processes. When an application is executed, the Win32 subsystem calls the VM manager to load

Notes

the application's executable code. The memory manager returns a status to Win32 that tells what kind of executable the code is. If it is not a nativeWin32 executable, theWin32 environment checks whether the appropriate environmental subsystem is running; if the subsystem is not running, it is started as a user-mode process. Then, Win32 creates a process to run the application, and passes control to the environmental subsystem.

The environmental subsystem uses the Windows 2000 LPC facility to get kernel services for the process. This approach helps Windows 2000 to be robust, because the parameters passed to a system call can be checked for correctness before the actual kernel routine is invoked. Windows 2000 prohibits applications from mixing API routines from different environments. For instance, a Win32 application cannot call a POSIX routine.

Since each subsystem is run as a separate user-mode process, a crash in one has no effect on the others. The exception is Win32, which provides all the keyboard, mouse, and graphical display capabilities. If it fails, the system is effectively disabled.

The Win32 environment categorizes applications as either graphical or character based, where a character-based application is one that thinks that interactive output goes to an 80 by 24 ASCII display. Win32 transforms the output of a character-based application to a graphical representation in a window. This transformation is easy: Whenever an output routine is called, the environmental subsystem calls a Win32 routine to display the text. Since the Win32 environment performs this function for all character-based windows, it can transfer screen text between windows via the clipboard. This transformation works for MS-DOS applications, as well as for POSIX command-line applications.

MS-DOS Environment

The MS-DOS environment does not have the complexity of the other Windows 2000 environmental subsystems. It is provided by a Win32 application called the virtual DOS machine (VDM). Since the VDM is just a user-mode process, it is paged and dispatched like any other Windows 2000 thread. The VDM has an instruction-execution unit to execute or emulate Intel 486 instructions. The VDM also provides routines to emulate the MS-DOS ROM BIOS and .int software interrupt services, and has virtual device drivers for the screen, keyboard, and communication ports. The VDM is based on the MS-DOS 5.0 source code; it gives the application at least 620 kilobytes of memory.

The Windows 2000 command shell is a program that creates a window that looks like an MS-DOS environment. It can run both 16-bit and 32-bit executables. When an MS-DOS application is run, the command shell starts a VDM process to execute the program. If Windows 2000 is running on an x86 processor, MS-DOS graphical applications run in full-screen mode, and character applications can run full screen or in a window. If Windows 2000 is running on different processor architecture, all MS-DOS applications run in windows. Some MS-DOS applications access the disk hardware directly, but they fail to run on Windows 2000 because disk access is privileged to protect the file system. In general, MS-DOS applications that directly access hardware will fail to operate under Windows 2000.

Since MS-DOS is not a multitasking environment, some applications have been written that hog the CPU-for instance, by using busy loops to cause time delays or pauses in execution. The priority mechanism in the Windows 2000 dispatcher detects such delays and automatically throttles the CPU consumption (and causes the offending application to operate incorrectly).

16-Bit Windows Environment

The Win16 execution environment is provided by a VDM that incorporates additional software called Windows on Windows that provides the Windows 3.1 kernel routines and stub routines for window manager and GDI functions. The stub routines call the appropriate Win32 subroutines,

Notes

converting, or thinking, 16-bit addresses into 32-bit ones. Applications that rely on the internal structure of the 16-bit window manager or GDI may not work, because Windows on Windows does not really implement the 16-bit API.

Windows on Windows can multitask with other processes on Windows 2000, but it resembles Windows 3.1 in many ways. Only one Win16 application can run at a time, all applications are single threaded and reside in the same address space, and they all share the same input queue. These features imply that an application that stops receiving input will block all the other Win16 applications, just as in Windows 3.x, and one Win16 application can crash other Win16 applications by corrupting the address space. Multiple Win16 environments can coexist, however, by using the command start/separate win16application from the command line.

Win32 Environment

The main subsystem in Windows 2000 is the Win32 subsystem. It runs Win32 applications, and manages all keyboard, mouse, and screen I/O. Since it is the controlling environment, it is designed to be extremely robust. Several features of Win32 contribute to this robustness. Unlike the Win16 environment, each Win32 process has its own input queue. The window manager dispatches all input on the system to the appropriate process's input queue, so a failed process will not block input to other processes. The Windows 2000 kernel also provides preemptive multitasking, which enables the user to terminate applications that have failed or are no longer needed. Win32 also validates all objects before using them, to prevent crashes that could otherwise occur if an application tried to use an invalid or wrong handle. TheWin32 subsystem verifies the type of the object to which a handle points before using that object. The reference counts kept by the object manager prevent objects from being deleted while they are still being used, and prevents their use after they have been deleted.

POSIX Sub-system

The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard, which is based on the UNIX model. POSIX applications can be started by the Win32 subsystem or by another POSIX application. POSIX applications use the POSIX subsystem server PSXSS.EXE, the POSIX dynamic link library PSXDLL.DLL, and the POSIX console session manager POSIX. EXE.

Although the POSIX standard does not specify printing, POSIX applications can use printers transparently via the Windows 2000 redirection mechanism.

POSIX applications have access to any file system on the Windows 2000 system; the POSIX environment enforces UNIX-like permissions on directory trees.

Several Win32 facilities are not supported by the POSIX subsystem, including memory-mapped files, networking, graphics, and dynamic data exchange.

OS/2 Sub-system

Although Windows 2000 was originally intended to provide a robust OS/2 operating environment, the success of Microsoft Windows led to a change; during the early development of Windows 2000, the Windows environment became the default. Consequently, Windows 2000 provides only limited facilities in the OS/2 environmental subsystem. OS/2 1.x character-based applications can run only on Windows 2000 on Intel x86 computers. Real-mode OS/2 applications can run on all platforms by using the MS-DOS environment. Bound applications, which have dual code for both MS-DOS and OS/2, run in the OS/2 environment unless the OS/2 environment is disabled.

Logon and Security Sub-systems

Notes

Before a user can access objects on Windows 2000, that user must be authenticated by the logon subsystem. To be authenticated, a user must have an account and provide the password for that account.

The security subsystem generates access tokens to represent users on the system. It calls an authentication package to perform authentication using information from the logon subsystem or network server. Typically, the authentication package simply looks up the account information in a local database and checks to see that the password is correct. The security subsystem then generates the access token for the user id containing the appropriate privileges, quota limits, and group ids. Whenever the user attempts to access an object in the system, such as by opening a handle to the object, the access token is passed to the security reference monitor, which checks privileges and quotas.

The default authentication package for Windows 2000 domains is Kerberos.

14.4 File System

Microsoft Windows 2000 supports four types of file systems on readable/writable disks: the NTFS file system and three file allocation table (FAT) file systems: FAT12, FAT16 and FAT32. Windows 2000 also supports two types of file systems on CD-ROM and digital video disk (DVD) media: Compact Disc File System (CDFS) and Universal Disk Format (UDF). The structures of the volumes formatted by each of these file systems, as well as the way each file system organizes data on the disk, are significantly different. The capabilities and limitations of these file systems must be reviewed to determine their comparative features.

In this section 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.

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, ϕ, λε 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.

Notes



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 264 bytes. To get some idea of how big a 264-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 267-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centuri 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.

14.5 Networking

Windows 2000 supports both peer-to-peer and client-server networking. It also has facilities for network management. The networking components in Windows 2000 provide data transport, inter-process communication, file sharing across a network, and the ability to send print jobs to remote printers.

To describe networking in Windows 2000, we will refer to two of the internal networking interfaces, called the Network Device Interface Specification (NDIS) and the Transport Driver Interface (TDI). The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from the transport protocols, so that either could be changed without affecting the other. NDIS resides at the interface between the data-link control and media-access-control layers in the OSI model and enables many protocols to operate over many different network adapters. In terms of the OSI model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport, and has functions to send any type of data.

14.5.1 Protocols

Windows 2000 implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows 2000 comes with several networking protocols.

The server message-block (SMB) protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network.

The SMB protocol has four message types. The Session control messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses File

Notes

messages to access files at the server. The system uses Printer messages to send data to a remote print queue and to receive back status information, and the Message is used to communicate with another workstation.

The Network Basic Input/Output System (NetBIOS) is a hardware-abstraction interface for networks, analogous to the BIOS hardware-abstraction interface devised for PCs running MS-DOS. NetBIOS, developed in the early 1980s, has become a standard network-programming interface. NetBIOS is used to establish logical names on the network, to establish logical connections or sessions between two logical names on the network, and to support reliable data transfer for a session via either NetBIOS or SMB requests.

The NetBIOS Extended User Interface (NetBEUI) was introduced by IBM in 1985 as a simple, efficient networking protocol for up to 254 machines. It is the default protocol for Windows 95 peer networking and for Windows for Workgroups. Windows 2000 uses NetBEUI when it wants to share resources with these networks. Among the limitations of NetBEUI are that it uses the actual name of a computer as the address, and that it does not support routing.

The TCP/IP protocol suite that is used on the Internet has become the de facto standard networking infrastructure; it is widely supported. Windows 2000 uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows 2000 TCP/IP package includes the simple network management protocol (SNMP), dynamic host-configuration protocol (DHCP), Windows Internet name service (WINS), and NetBIOS support. The point-to-point tunneling protocol (PPTP) is a protocol provided by Windows 2000 to communicate between remote-access server modules running on Windows 2000 Server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multi-protocol virtual private networks over the Internet.

The Novell NetWare protocols (IPX datagram service on the SPX transport layer) are widely used for PC LANs. The Windows 2000 NWLink protocol connects the NetBIOS to NetWare networks. In combination with a redirector (such as Microsoft's Client Service for Netware or Novell's NetWare Client for Windows 2000), this protocol enables a Windows 2000 client to connect to a NetWare server.

Windows 2000 uses the data-link control (DLC) protocol to access IBM mainframes and HP printers that are connected directly to the network. This protocol is not otherwise used by Windows 2000 systems.

The AppleTalk protocol was designed as a low-cost connection by Apple to allow Macintosh computers to share files. Windows 2000 systems can share files and printers with Macintosh computers via AppleTalk if a Windows 2000 server on the network is running the Windows 2000 Services for Macintosh package.

14.5.2 Distributed-Processing Mechanisms

Although Windows 2000 is not a distributed operating system, it does support distributed applications. Mechanisms that support distributed processing on Windows 2000 include NetBIOS, named pipes and mailslots, windows sockets, remote procedure calls (RPC), and network dynamic data exchange (NetDDE).

In Windows 2000, NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP. Named pipes are a connection-oriented messaging mechanism. Named pipes were originally developed as a high-level interface to NetBIOS connections over the network. A process can also use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes.

The name of a named pipe has a format called the uniform naming convention (UNC). A UNC name looks like a typical remote file name. The format of a UNC name is __server name_share

Notes

name_x_y_z, where the server name identifies a server on the network; a share name identifies any resource that is made available to network users, such as directories, files, named pipes and printers; and the _x_y_z part is a normal file path name.

Mailslots are a connectionless messaging mechanism. They are unreliable, in that a message sent to a mailslot may be lost before the intended recipient receives it. Mailslots are used for broadcast applications, such as for finding components on the network; they are also used by the Windows 2000 Computer Browser service.

Winsock is the Windows 2000 sockets API. Winsock is a session-layer interface that is largely compatible with UNIX sockets, with some Windows 2000 extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack.

A remote procedure call (RPC) is a client-server mechanism that enables an application on one machine to make a procedure call to code on another machine. The client calls a local procedure-a stub routine-that packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called marshaling.

The Windows 2000 RPC mechanism follows the widely used distributed computing environment standard for RPC messages, so programs written to use Windows 2000 RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences between computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

Windows 2000 can send RPC messages using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks. The LPC facility, discussed earlier, is similar to RPC, except that in the LPC case the messages are passed between two processes running on the same computer.

It is tedious and error-prone to write the code to marshal and transmit arguments in the standard format, to un-marshal and execute the remote procedure, to marshal and send the return results, and to un-marshal and return them to the caller. Fortunately, however, much of this code can be generated automatically from a simple description of the arguments and return results.

Windows 2000 provides the Microsoft Interface Definition Language to describe the remote procedure names, arguments, and results. The compiler for this language generates header files that declare the stubs for the remote procedures, and the data types for the argument and return-value messages.

It also generates source code for the stub routines used at the client side, and for an un-marshaller and dispatcher at the server side. When the application is linked, the stub routines are included. When the application executes the RPC stub, the generated code handles the rest.

DCOM (COM) is a mechanism for inter-process communication that was developed for Windows. COM objects provide a well defined interface to manipulate the data in the object. Windows 2000 has an extension called DCOM that can be used over a network utilizing the RPC mechanism to provide a transparent method of developing distributed applications.

14.5.3 Redirectors and Servers

In Windows 2000, an application can use the Windows 2000 I/O API to access files from a remote computer as though they were local, provided that the remote computer is running an MS-NET server, such as is provided by Windows 2000 or Windows for Workgroups. A redirector is the

client-side object that forwards I/O requests to remote files, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

Notes

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet.
3. The I/O manager recognizes that the access is for a remote file, and calls a driver called a multiple universal-naming-convention provider (MUP).
4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a multi-provider router is used, instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol. The list of redirectors is maintained in the system registry database.

14.5.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school, or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows 2000 uses the concept of a domain. Previously, these domains had no relationship whatsoever to the Domain Name System that maps Internet host names to IP addresses; now, however, they are closely related. Specifically, a Windows 2000 domain is a group of Windows 2000 workstations and servers that shares a common security policy and user database. Since Windows 2000 now uses the Kerberos protocol for trust and authentication, a Windows 2000 domain is the same thing as a Kerberos realm. Previous versions of NT used the idea of primary and backup domain controllers; now all servers in a domain are domain controllers.

In addition, previous versions required the setup of one-way trusts between domains. Windows 2000 utilizes uses a hierarchical approach based on DNS, and allows transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for n domains from n _ (n _ 1) to O(n). The workstations in the domain trust the domain controller to give correct information about the access rights of each user (via the user's access token). All users retain the ability to restrict access to their own workstations, no matter what any domain controller may say.

Because a business may have many departments, and a school may have many classes, it is often necessary to manage multiple domains within a single organization. A domain tree is a

Notes

contiguous DNS naming hierarchy. For example, bell-labs.com might be the root of the tree, with research.bell-labs.com and pez.bell-labs.com as children -(domains research and pez). A forest is a set of non-contiguous names. An example would be the trees bell-labs.com and/or lucent.com. A forest may be comprised of only one domain tree, however.

Trust relationships may be set up between domains in three ways: one-way, transitive, and cross-link. Versions of NT through version 4.0 allowed only one-way trusts to be set up. A one-way trust is exactly what its name implies: Domain A is told it can trust domain B. However, B would not trust A unless another relationship is configured. Under a transitive trust, if A trusts B and B trusts C, then A, B, and C all trust each other since transitive trusts are two-way by default. Transitive trusts are enabled by default for new domains in a tree and can only be configured among domains within a forest. The third type, a cross-link trust, is useful to cut down on authentication traffic. Suppose that domains A and B are leaf nodes, and that users in A often use resources in B. If a standard transitive trust is used, authentication requests must traverse up to the common ancestor of the two leaf nodes; but if A and B have a cross-linking trust established, the authentications would be sent directly to the other node.

14.5.5 Name Resolution in TCP/IP Networks

On an IP network, name resolution is the process of converting a computer name to an IP address, such as resolving www.bell-labs.com to 135.104.1.14.

Windows 2000 provides several methods of name resolution, including Windows Internet Name Service (WINS), broadcast name resolution, domain name system (DNS), a hosts file, and an LMHOSTS file. Most of these methods are used by many operating systems, so we describe only WINS here.

Under WINS, two or more WINS servers maintain a dynamic database of name-to-IP address bindings, and client software to query the servers. At least two servers are used, so that the WINS service can survive a server failure, and so that the name-resolution workload can be spread over multiple machines.

WINS uses the dynamic host-configuration protocol (DHCP). DHCP updates address configurations automatically in the WINS database, without user or administrator intervention, as follows. When a DHCP client starts up, it broadcasts a discover message. Each DHCP server that receives the message replies with an offer message that contains an IP address and configuration information for the client. The client then chooses one of the configurations and sends a request message to the selected DHCP server. The DHCP server responds with the IP address and configuration information it gave previously, and with a lease for that address. The lease gives the client the right to use that IP address for a specified period of time. When the lease time is half expired, the client will attempt to renew the lease for that address. If the lease is not renewed, the client must get a new one.

14.6 Programmer Interface

The Win32 API is the fundamental interface to the capabilities of Windows 2000. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

14.6.1 Access to Kernel Objects

The Windows 2000 kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named XXX by calling the CreateXXX function to open a handle to XXX. This handle is unique to that process. Depending on which object is being opened, if the create function fails, it

Notes

may return 0, or it may return a special constant named INVALID HANDLE VALUE. A process can close any handle by calling the CloseHandle function, and the system may delete the object if the count of processes using the object drops to 0.

Windows 2000 provides three ways to share objects between processes. The First way is for a child process to inherit a handle to the object. When the parent calls the CreateXXX function, the parent supplies a SECURITIES ATTRIBUTES structure with the bInheritHandle FIeld set to TRUE. This field creates an inheritable handle. Then, the child process can be created, passing a value of TRUE to the CreateProcess function's bInheritHandle argument. Assuming that the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects.

The second way to share objects is for one process to give the object a name when that object is created, and for the second process to open that name. This method has two drawbacks. One is that Windows 2000 does not provide a way to check whether an object with the chosen name already exists. A second drawback is that the object name space is global, without regard to the object type. For instance, two applications may create an object named .pipe. when two distinct (and possibly different) objects are desired.

Named objects have the advantage that unrelated processes can share them readily. The First process would call one of the CreateXXX functions and supply a name in the lpszName parameter. The second process can get a handle to share this object by calling OpenXXX (or CreateXXX) with the same name,

The third way to share objects is via the DuplicateHandle function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process, and the value of a handle within that process, a second process can get a handle to the same object, and thus share it.

In Windows 2000, a process is an executing instance of an application, and a thread is a unit of code that can be scheduled by the operating system. Thus, a process contains one or more threads. A process is started when some other process calls the CreateProcess routine. This routine loads any dynamic link libraries that are used by the process, and creates a primary thread. Additional threads can be created by the CreateThread function. Each thread is created with its own stack, which defaults to one MB unless specified otherwise in an argument to CreateThread. Because some C run-time functions maintain state in static variables, such as errno, a multithread application needs to guard against unsynchronized access. The wrapper function beginthreadex provides appropriate synchronization.

Every dynamic link library or executable file that is loaded into the address space of a process is identified by an instance handle. The value of the instance handle is actually the virtual address where the file is loaded. An application can get the handle to a module in its address space by passing the name of the module to GetModuleHandle. If NULL is passed as the name, the base address of the process is returned. The lowest 64 kilobytes of the address space are not used, so a faulty program that tries to dereference a NULL pointer will get an access violation.

Priorities in the Win32 environment are based on the Windows 2000 scheduling model, but not all priority values may be chosen. Win32 uses four priority classes: IDLE PRIORITY CLASS (priority level 4), NORMAL PRIORITY CLASS (level 8), HIGH PRIORITY CLASS (level 13) and REALTIME PRIORITY CLASS (level 24). Processes are typically members of the NORMAL PRIORITY CLASS unless the parent of the process was of the IDLE PRIORITY CLASS, or another class was specified when CreateProcess was called. The priority class of a process can be changed with the SetPriorityClass function, or by an argument being passed to the START command. For example, the command START/REALTIME observer.exe would run the observer program in the REALTIME PRIORITY CLASS. Note that only users with the increase scheduling priority privilege can move a process into the REALTIME PRIORITY CLASS.

Administrators and power users have this privilege by default.

Notes

When a user is running an interactive program, the system needs to provide especially good performance for that process. For this reason, Windows 2000 has a special scheduling rule for processes in the NORMAL PRIORITY CLASS.

Windows 2000 distinguishes between the foreground process that is currently selected on the screen, and the background processes that are not currently selected. When a process moves into the foreground, Windows 2000 increases the scheduling quantum by some factor-typically by 3. (This factor can be changed via the performance option in the system section of the control panel).

This increase gives the foreground process three times longer to run before a time sharing preemption occurs.

A thread can be created in a suspended state: The thread will not execute until another thread makes it eligible via the ResumeThread function. The SuspendThread function does the opposite. These functions set a counter, so if a thread is suspended twice, it must be resumed twice before it can run.

To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.

In addition, synchronization of threads can be achieved by using the WaitForSingleObject or WaitForMultipleObjects functions. Another method of synchronization in the Win32 API is the critical section. A critical section is a synchronized region of code that can be executed by only one thread at a time. A thread establishes a critical section by calling InitializeCriticalSection.

The application must call EnterCriticalSection before entering the critical section, and LeaveCriticalSection after exiting the critical section. These two routines guarantee that, if multiple threads attempt to enter the critical section concurrently, only one thread at a time will be permitted to proceed, and the others will wait in the EnterCriticalSection routine. The critical-section mechanism is slightly faster than the kernel-synchronization objects.

A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm. A process may have multiple fibers in it, just as it can have multiple threads. A major difference between threads and fibers is that threads can execute concurrently, but only one fiber at a time is permitted to execute, even on multiprocessor hardware. This mechanism is included in Windows 2000 to facilitate the porting of those legacy UNIX applications that were written for a fiber-execution model.

The system creates a fiber by calling either ConvertThreadToFiber or CreateFiber. The primary difference between these functions is that CreateFiber does not begin executing the fiber that was created. To begin execution, the application must call SwitchToFiber. The application can terminate a fiber by calling DeleteFiber.

14.6.2 Inter-process Communication

One way that Win32 applications can do inter-process communication is by sharing kernel objects. Another way is by passing messages, an approach that is particularly popular for Windows GUI applications.

One thread can send a message to another thread or to a window by calling PostMessage, PostThreadMessage, SendMessage, SendThreadMessage, or SendMessageCallback. The difference between posting a message and sending a message is that the post routines are asynchronous: They return immediately, and the calling thread does not know when the message is actually delivered.

The send routines are synchronous-they block the caller until the message has been delivered and processed.

Notes

In addition to sending a message, a thread can also send data with the message. Since processes have separate address spaces, the data must be copied. The system copies them by calling endMessage to send a message of type WM COPYDATA with a COPYDATASTRUCT data structure that contains the length and address of the data to be transferred. When the message is sent, Windows 2000 copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Unlike the 16-bit windows environment, every Win32 thread has its own input queue from which the thread receives messages. (All input is received via messages.) This structure is more reliable than the shared input queue of 16-bit windows, because, with separate queues, one stuck application cannot block input to the other applications.

14.6.3 Memory Management

TheWin32 API provides several ways for an application to use memory: virtual memory, memory-mapped FIles, heaps, and thread-local storage.

An application calls VirtualAlloc to reserve or commit virtual memory, and VirtualFree to decommit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated.

They operate on multiples of the memory pagesize, and the starting address of an allocated region must be greater than 0_10000.

A process may lock some of its committed pages into physical memory by calling VirtualLock. The maximum number of pages that a process can lock is 30, unless the process FIrst calls SetProcessWorkingSetSize to increase the minimum working-set size.

Another way for an application to use memory is by memory mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: Both processes map the same file into their virtual memory. Memory mapping is a multistage process.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process can call CreateFileMapping with a file handle of off 0×ffffffff and a particular size. The resulting file-mapping object can be shared by inheritance, by name lookup, or by duplication.

The third way for applications to use memory is a heap. A heap in the Win32 environment is just a region of reserved address space. When a Win32 process is initialized, it is created with a 1-MB default heap. Since many Win32 functions use the default heap, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads. Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are HeapCreate, HeapAlloc, HeapRealloc, HeapSize, HeapFree, and HeapDestroy.

TheWin32 API also provides the HeapLock and HeapUnlock functions to enable a thread to gain exclusive access to a heap. Unlike VirtualLock, these functions perform only synchronization; they do not lock pages into physical memory.

The fourth way for applications to use memory is a thread-local storage mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function strtok uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute strtok correctly, they need separate .current position. variables. The thread-local storage mechanism allocates global storage on a per-thread basis. It provides both dynamic and static methods of creating thread-local storage.

Notes

14.7 Summary

Microsoft designed Windows 2000 to be an extensible, portable operating system - one able to take advantage of new techniques and hardware. Windows 2000 supports multiple operating environments and symmetric multiprocessing. The use of kernel objects to provide basic services, and the support for client-server computing, enable Windows 2000 to support a wide variety of application environments. For instance, Windows 2000 can run programs compiled for MS-DOS, Win16, Windows 95, Windows 2000, and/or POSIX. It provides virtual memory, integrated caching, and preemptive scheduling. Windows 2000 supports a security model stronger than those of previous Microsoft operating systems, and includes internationalization features. Windows 2000 runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements, without needing to alter the applications that they run.

14.8 Keywords

I/O Manager: It allows devices to communicate with user-mode subsystems by translating user-mode read and write commands and passing them to device drivers.

IPC Manager: Short form of Inter-Process Communication Manager, manages the communication between clients (the environment subsystem) and servers (components of the executive).

Object manager: It is a special executive subsystem that all other executive subsystems must pass through to gain access to Windows 2000 resources.

PnP Manager: It handles Plug and Play and supports device detection and installation at boot time.

Power Manager: The power manager coordinates power events and generates power IRPs.

Process Manager: Handles process and thread creation and termination

Security Reference Monitor (SRM): The primary authority for enforcing the security rules of the security integral subsystem.

Virtual Memory Manager: It manages virtual memory, allowing Windows 2000 to use the hard disk as a primary storage device (although strictly speaking it is secondary storage).

Windows 2000 Advanced Server: It is an operating system which supports up to eight processors and up to 8GB of RAM. It is used in an enterprise network and very useful as an SQL server.

Windows 2000 Datacenter Server: It is an operating system which upports up to 32 processors and up to 64GB of RAM. It is used in an enterprise network to support extremely large databases and real time processing.

Windows 2000 Professional: It is an operating system which supports up to two processors and up to 4GB of RAM. Used as a workstation or client computer and it is the replacement for Windows NT Workstation.

Windows 2000 Server: It is an operating system which supports up to four processors and up to 4GB of RAM. It is used for web, application, print and file servers.

14.9 Review Questions

1. What are the differences between Windows 2000 Professional, Server, Advanced Server, and DataCenter?
2. Write short notes on:
   (a) NTFS

(b) POSIX subsystem

(c) Distributed-Processing Mechanisms

3. How Windows 2000 accesses the remote file?

4. What is IPC? How does it occur in Windows 2000?

Notes

14.10 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "*Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "*Operating System Concepts*", John Wiley & Sons, Seventh Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com