OPERATING SYSTEM

INDEX

LESSON 1: INTRODUCTION TO OPERATING SYSTEM
LESSON 2: FILE SYSTEM – I
LESSON 3: FILE SYSTEM – II
LESSON 4: CPU SCHEDULING
LESSON 5: MEMORY MANAGEMENT – I
LESSON 6: MEMORY MANAGEMENT – II
LESSON 7: DISK SCHEDULING
LESSON 8: PROCESS MANAGEMENT
LESSON 9: DEADLOCKS
LESSON 10: CASE STUDY OF UNIX
LESSON 11: CASE STUDY OF MS-DOS
LESSON 12: CASE STUDY OF MS-WINDOWS NT

Lesson Number: 1 ***Writer: Dr. Rakesh Kumar***

Introduction to Operating System ***Vetter: Prof. Dharminder Kr.***

1.0 OBJECTIVE

The objective of this lesson is to make the students familiar with the basics of operating system. After studying this lesson they will be familiar with:

1. What is an operating system?
2. Important functions performed by an operating system.
3. Different types of operating systems.

1. 1 INTRODUCTION

Operating system (OS) is a program or set of programs, which acts as an interface between a user of the computer & the computer hardware. The main purpose of an OS is to provide an environment in which we can execute programs. The main goals of the OS are (i) To make the computer system convenient to use, (ii) To make the use of computer hardware in efficient way.

Operating System is system software, which may be viewed as collection of software consisting of procedures for operating the computer & providing an environment for execution of programs. It's an interface between user & computer. So an OS makes everything in the computer to work together smoothly & efficiently.

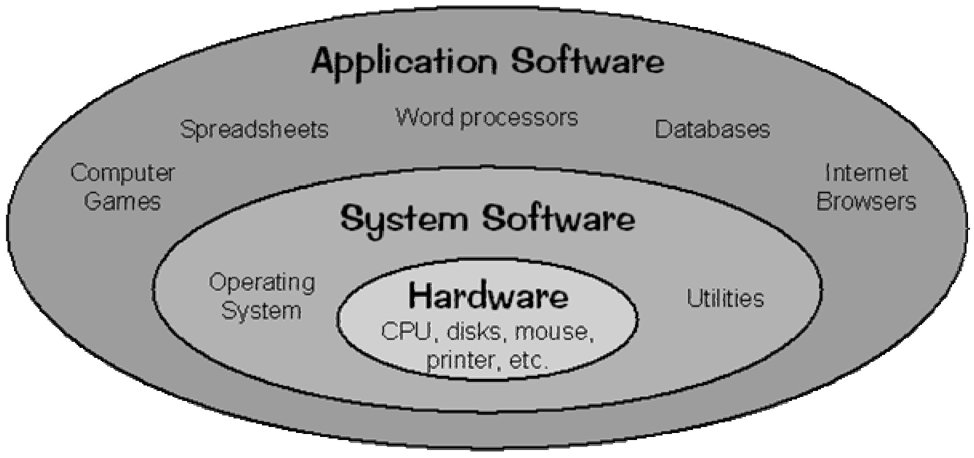


Figure 1: The relationship between application & system software

Basically, an OS has three main responsibilities: (a) Perform basic tasks such as recognizing input from the keyboard, sending output to the display screen, keeping track of files & directories on the disk, & controlling peripheral devices such as disk drives & printers (b) Ensure that different programs & users running at the same time do not interfere with each other; & (c) Provide a software platform on top of which other programs can run. The OS is also responsible for security, ensuring that unauthorized users do not access the system. Figure 1 illustrates the relationship between application software & system software.

The first two responsibilities address the need for managing the computer hardware & the application programs that use the hardware. The third responsibility focuses on providing an interface between application software & hardware so that application software can be efficiently developed. Since the OS is already responsible for managing the hardware, it should provide a programming interface for application developers. As a user, we normally interact with the OS through a set of commands. The commands are accepted & executed by a part of the OS called the command processor or command line interpreter.

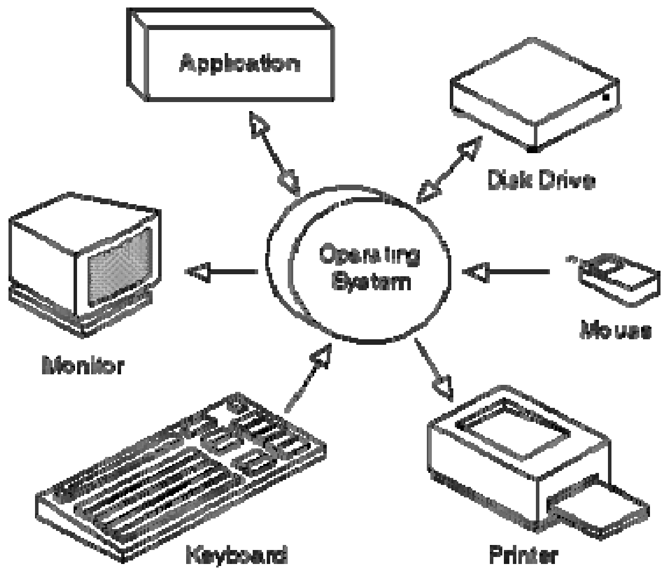


Figure 2: The interface of various devices to an operating system

In order to understand operating systems we must understand the computer hardware & the development of OS from beginning. Hardware means the physical machine & its electronic components including memory chips, input/output devices, storage devices & the central processing unit. Software are the programs written for these computer systems. Main memory is where the data & instructions are stored to be processed. Input/Output devices are the peripherals attached to the system, such as keyboard, printers, disk drives, CD drives, magnetic tape drives, modem, monitor, etc. The central processing unit is the brain of the computer system; it has circuitry to control the interpretation & execution of instructions. It controls the operation of entire computer system. All of the storage references, data manipulations & I/O operations are performed by the CPU. The entire computer systems can be divided into four parts or components (1) The hardware (2) The OS (3) The application programs & system programs (4) The users.

The hardware provides the basic computing power. The system programs the way in which these resources are used to solve the computing problems of the users. There may be many different users trying to solve different problems. The OS controls & coordinates the use of the hardware among the various users & the application programs.

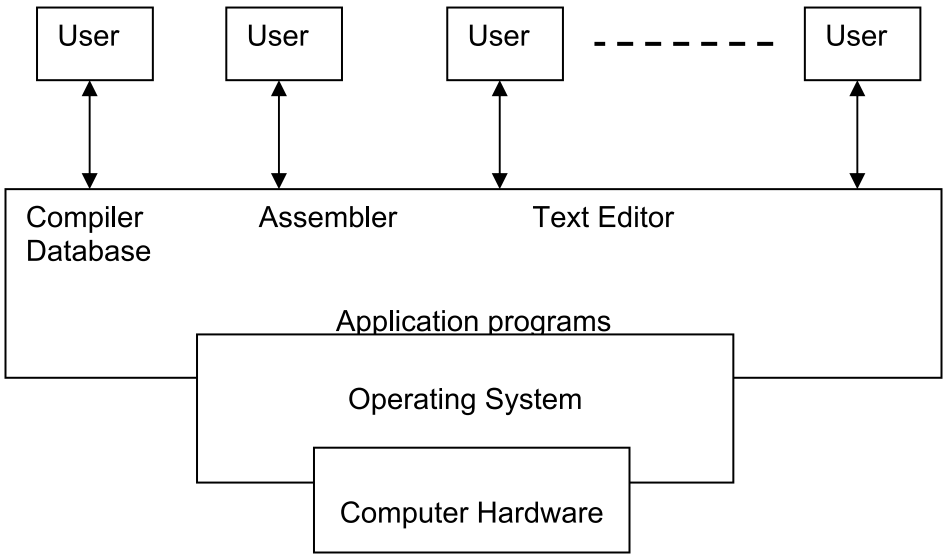
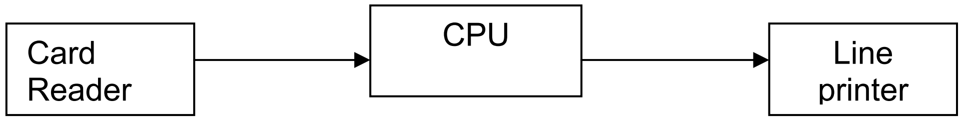


Figure 3. Basic components of a computer system

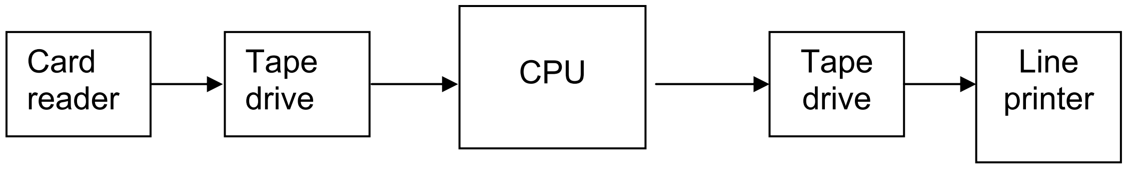
We can view an OS as a resource allocator. A computer system has many resources, which are to be required to solve a computing problem. These

resources are the CPU time, memory space, files storage space, input/output devices & so on. The OS acts as a manager of all of these resources & allocates them to the specific programs & users as needed by their tasks. Since there can be many conflicting requests for the resources, the OS must decide which requests are to be allocated resources to operate the computer system fairly & efficiently.
An OS can also be viewed as a control program, used to control the various I/O devices & the users programs. A control program controls the execution of the user programs to prevent errors & improper use of the computer resources. It is especially concerned with the operation & control of I/O devices. As stated above the fundamental goal of computer system is to execute user programs & solve user problems. For this goal computer hardware is constructed. But the bare hardware is not easy to use & for this purpose application/system programs are developed. These various programs require some common operations, such as controlling/use of some input/output devices & the use of CPU time for execution. The common functions of controlling & allocation of resources between different users & application programs is brought together into one piece of software called operating system. It is easy to define operating systems by what they do rather than what they are. The primary goal of the operating systems is convenience for the user to use the computer. Operating systems makes it easier to compute. A secondary goal is efficient operation of the computer system. The large computer systems are very expensive, & so it is desirable to make them as efficient as possible. Operating systems thus makes the optimal use of computer resources. In order to understand what operating systems are & what they do, we have to study how they are developed. Operating systems & the computer architecture have a great influence on each other. To facilitate the use of the hardware operating systems were developed.
First, professional computer operators were used to operate the computer. The programmers no longer operated the machine. As soon as one job was finished, an operator could start the next one & if some errors came in the program, the operator takes a dump of memory & registers, & from this the programmer have

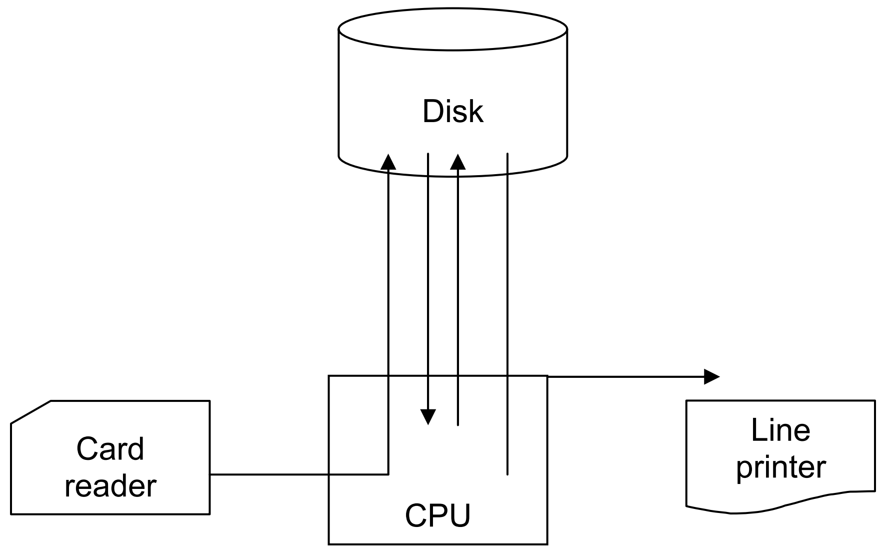
to debug their programs. The second major solution to reduce the setup time was to batch together jobs of similar needs & run through the computer as a group. But there were still problems. For example, when a job stopped, the operator would have to notice it by observing the console, determining why the program stopped, takes a dump if necessary & start with the next job. To overcome this idle time, automatic job sequencing was introduced. But even with batching technique, the faster computers allowed expensive time lags between the CPU & the I/O devices. Eventually several factors helped improve the performance of CPU. First, the speed of I/O devices became faster. Second, to use more of the available storage area in these devices, records were blocked before they were retrieved. Third, to reduce the gap in speed between the I/O devices & the CPU, an interface called the control unit was placed between them to perform the function of buffering. A buffer is an interim storage area that works like this: as the slow input device reads a record, the control unit places each character of the record into the buffer. When the buffer is full, the entire record is transmitted to the CPU. The process is just opposite to the output devices. Fourth, in addition to buffering, an early form of spooling was developed by moving off-line the operations of card reading, printing etc. SPOOL is an acronym that stands for the simultaneous peripherals operations on-line. Foe example, incoming jobs would be transferred from the card decks to tape/disks off-line. Then they would be read into the CPU from the tape/disks at a speed much faster than the card reader.



On-line



Off-line



SPOOLING

Figure 4: the on-line, off-line & spooling processes

Moreover, the range & extent of services provided by an OS depends on a number of factors. Among other things, the needs & characteristics of the target environmental that the OS is intended to support largely determine user- visible functions of an operating system. For example, an OS intended for program development in an interactive environment may have a quite different set of system calls & commands than the OS designed for run-time support of a car engine.

1.2 PRESENTATION OF CONTENTS

1.2.1 Operating System as a Resource Manager
1.2.1.1 Memory Management Functions
1.2.1.2 Processor / Process Management Functions
1.2.1.3 Device Management Functions
1.2.1.4 Information Management Functions
1.2.2 Extended Machine View of an Operating System
1.2.3 Hierarchical Structure of an Operating System
1.2.4 Evolution of Processing Trends
1.2.4.1 Serial Processing
1.2.4.2 Batch Processing

1.2.4.3 Multi Programming
1.2.5 Types Of Operating Systems
1.2.5.1 Batch Operating System
1.2.5.2 Multi Programming Operating System
1.2.5.3 Multitasking Operating System
1.2.5.4 Multi-user Operating System
1.2.5.5 Multithreading
1.2.5.6 Time Sharing System
1.2.5.7 Real Time Systems
1.2.5.8 Combination Operating Systems
1.2.5.9 Distributed Operating Systems
1.2.6 System Calls

1.2.1 OPERATING SYSTEM AS A RESOURCE MANAGER

The OS is a manager of system resources. A computer system has many resources as stated above. Since there can be many conflicting requests for the resources, the OS must decide which requests are to be allocated resources to operate the computer system fairly & efficiently. Here we present a framework of the study of OS based on the view that the OS is manager of resources. The OS as a resources manager can be classified in to the following three popular views: primary view, hierarchical view, & extended machine view. The primary view is that the OS is a collection of programs designed to manage the system's resources, namely, memory, processors, peripheral devices, & information. It is the function of OS to see that they are used efficiently & to resolve conflicts arising from competition among the various users. The OS must keep track of status of each resource; decide which process is to get the resource, allocate it, & eventually reclaim it.

The major functions of each category of OS are.

1.2.1.1 Memory Management Functions

To execute a program, it must be mapped to absolute addresses & loaded into memory. As the program executes, it accesses instructions & data from memory by generating these absolute addresses. In multiprogramming environment, multiple programs are maintained in the memory simultaneously. The OS is responsible for the following memory management functions:

- Keep track of which segment of memory is in use & by whom.
- Deciding which processes are to be loaded into memory when space becomes available. In multiprogramming environment it decides which process gets the available memory, when it gets it, where does it get it, & how much.
- Allocation or de-allocation the contents of memory when the process request for it otherwise reclaim the memory when the process does not require it or has been terminated.

1.2.1.2 Processor/Process Management Functions

A process is an instance of a program in execution. While a program is just a passive entity, process is an active entity performing the intended functions of its related program. To accomplish its task, a process needs certain resources like CPU, memory, files & I/O devices. In multiprogramming environment, there will a number of simultaneous processes existing in the system. The OS is responsible for the following processor/ process management functions:

- Provides mechanisms for process synchronization for sharing of resources amongst concurrent processes.
- Keeps track of processor & status of processes. The program that does this has been called the traffic controller.
- Decide which process will have a chance to use the processor; the job scheduler chooses from all the submitted jobs & decides which one will be allowed into the system. If multiprogramming, decide which process gets the processor, when, for how much of time. The module that does this is called a process scheduler.
- Allocate the processor to a process by setting up the necessary hardware registers. This module is widely known as the dispatcher.
- Providing mechanisms for deadlock handling.
- Reclaim processor when process ceases to use a processor, or exceeds the allowed amount of usage.

1.2.1.3 I/O Device Management Functions

An OS will have device drivers to facilitate I/O functions involving I/O devices. These device drivers are software routines that control respective I/O devices through their controllers. The OS is responsible for the following I/O Device Management Functions:

- Keep track of the I/O devices, I/O channels, etc. This module is typically called I/O traffic controller.
- Decide what is an efficient way to allocate the I/O resource. If it is to be shared, then decide who gets it, how much of it is to be allocated, & for how long. This is called I/O scheduling.
- Allocate the I/O device & initiate the I/O operation.
- Reclaim device as & when its use is through. In most cases I/O terminates automatically.

1.2.1.4 Information Management Functions

- Keeps track of the information, its location, its usage, status, etc. The module called a file system provides these facilities.
- Decides who gets hold of information, enforce protection mechanism, & provides for information access mechanism, etc.
- Allocate the information to a requesting process, e.g., open a file.
- De-allocate the resource, e.g., close a file.

1.2.2 Network Management Functions

An OS is responsible for the computer system networking via a distributed environment. A distributed system is a collection of processors, which do not share memory, clock pulse or any peripheral devices. Instead, each processor is having its own clock pulse, & RAM & they communicate through network. Access to shared resource permits increased speed, increased functionality & enhanced reliability. Various networking protocols are TCP/IP (Transmission Control Protocol/ Internet Protocol), UDP (User Datagram Protocol), FTP (File Transfer Protocol), HTTP (Hyper Text Transfer protocol), NFS (Network File System) etc.

1.2.3 EXTENDED MACHINE VIEW OF AN OPERATING SYSTEM

As discussed in previous section, there arises a need to identify the system resources that must be managed by the OS & using the process viewpoint, we indicate when the corresponding resource manager comes into play. We now answer the question, “How are these resource managers activated, & where do they reside?” Does memory manager ever invoke the process scheduler? Does scheduler ever call upon the services of memory manager? Is the process concept only for the user or is it used by OS also?

The OS provides many instructions in addition to the Bare machine instructions (A Bare machine is a machine without its software clothing, & it does not provide the environment which most programmers are desired for). Instructions that form a part of Bare machine plus those provided by the OS constitute the instruction set of the extended machine. The situation is pictorially represented in figure 5. The OS kernel runs on the bare machine; user programs run on the extended machine. This means that the kernel of OS is written by using the instructions of bare machine only; whereas the users can write their programs by making use of instructions provided by the extended machine.

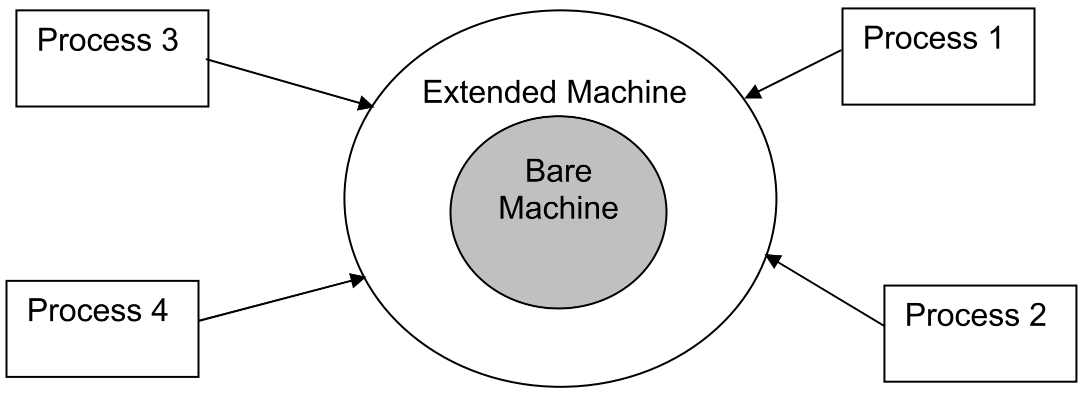


Figure 5. Extended Machine View

The OS kernel runs on the bare machine; user programs run on the extended machine. This means that the kernel of OS is written by using the instructions of bare machine only; whereas the users can write their programs by making use of instructions provided by the extended machine.

1.2.4 EVOLUTION OF PROCESSING TRENDS

Starting from the bare machine approach to its present forms, the OS has evolved through a number of stages of its development like serial processing, batch processing multiprocessing etc. as mentioned below:

1.2.4.1 Serial Processing

In theory, every computer system may be programmed in its machine language, with no systems software support. Programming of the bare machine was customary for early computer systems. A slightly more advanced version of this mode of operation is common for the simple evaluation boards that are sometimes used in introductory microprocessor design & 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 & data are then entered into the computer by means of console switches, or perhaps through a hexadecimal keyboard. Loading the program counter with the address of the first instruction starts programs. Results of execution are obtained by examining the contents of the relevant registers & memory locations. The executing program, if any, must control Input/output devices, directly, say, by reading & writing the related I/O ports. Evidently, programming of the bare machine results in low productivity of both users & hardware. The long & tedious process of program & 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 & paper tape, & of language translators. Programs, now coded in a programming language, are translated into executable form by a computer program, such as a compiler or an interpreter. Another program, called the loader, automates the process of loading executable programs into memory. The user places a program & its input data on an input device, & the loader transfers information from that input device into memory. After transferring control to the loader program by manual or automatic means, execution of the program commences. The executing program reads its input from the designated input device & may produce some output on an output device. Once in memory, the program may be rerun with a different set of input data.

The mechanics of development & preparation of programs in such environments are quite slow & cumbersome due to serial execution of programs & to 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 & execute the language translator & to provide it with the source code of the user program. When serial input devices, such as card reader, 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 & executed. If run-time errors are detected, the state of the machine can be examined & modified by means of console switches, or with the assistance of a program called a debugger.

1.2.4.2 Batch Processing

With the invention of hard disk drive, the things were much better. The batch processing was relied on punched cards or tape for the input when assembling the cards into a deck & running the entire deck of cards through a card reader as a batch. Present batch systems aren't limited to cards or tapes, but the jobs are still processed serially, without the interaction of the user. The efficiency of these systems was measured in the number of jobs completed in a given amount of time called as throughput. Today's operating systems are not limited to batch programs. This was the next logical step in the evolution of operating systems to automate the sequencing of operations involved in program execution & in the mechanical aspects of program development. The intent was to increase system resource utilization & programmer productivity by reducing or eliminating component idle times caused by comparatively lengthy manual operations.

Furthermore, even when automated, housekeeping operations such as mounting of tapes & filling out log forms take a long time relative to processors & memory speeds. Since there is not much that can be done to reduce these operations, system performance may be increased by dividing this overhead among a number of programs. More specifically, if several programs are batched together on a single input tape for which housekeeping operations are performed only once, the overhead per program is reduced accordingly. A related concept, sometimes called phasing, is to prearrange submitted jobs so that similar ones are placed in the same batch. For example, by batching several Fortran compilation jobs together, the Fortran compiler can be loaded only once to process all of them in a row. To realize the resource-utilization potential of batch processing, a mounted batch of jobs must be executed automatically, without slow human intervention. Generally, OS commands are statements written in Job Control Language (JCL). These commands are embedded in the job stream, together with user programs & data. A memory-resident portion of the batch operating system- sometimes called the batch monitor- reads, interprets, & executes these commands.

Moreover, the sequencing of program execution mostly automated by batch operating systems, the speed discrepancy between fast processors & comparatively slow I/O devices, such as card readers & printers, emerged as a major performance bottleneck. Further improvements in batch processing were mostly along the lines of increasing the throughput & resource utilization by overlapping input & output operations. These developments have coincided with the introduction of direct memory access (DMA) channels, peripheral controllers, & later dedicated input/output processors. As a result, satellite computers for offline processing were often replaced by sophisticated input/output programs executed on the same computer with the batch monitor.

Many single-user operating systems for personal computers basically provide for serial processing. User programs are commonly loaded into memory & executed in response to user commands typed on the console. A file management system is often provided for program & data storage. A form of batch processing is made possible by means of files consisting of commands to the OS that are executed in sequence. Command files are primarily used to automate complicated customization & operational sequences of frequent operations.

1.2.4.3 Multiprogramming

In multiprogramming, many processes are simultaneously resident in memory, & execution switches between processes. The advantages of multiprogramming are the same as the commonsense reasons that in life you don't always wait until one thing has finished before starting the next thing. Specifically:

- More efficient use of computer time. If the computer is running a single process, & the process does a lot of I/O, then the CPU is idle most of the time. This is a gain as long as some of the jobs are I/O bound -- spend most of their time waiting for I/O.
- Faster turnaround if there are jobs of different lengths. Consideration (1) applies only if some jobs are I/O bound. Consideration (2) applies even if all jobs are CPU bound. For instance, suppose that first job A, which takes an hour, starts to run, & then immediately afterward job B, which takes 1 minute, is submitted. If the computer has to wait until it finishes A before it starts B, then user A must wait an hour; user B must wait 61 minutes; so the average waiting time is 60-1/2 minutes. If the computer can switch back & forth between A & B until B is complete, then B will complete after 2 minutes; A will complete after 61 minutes; so the average waiting time will be 31-1/2 minutes. If all jobs are CPU bound & the same length, then there is no advantage in multiprogramming; you do better to run a batch system. The multiprogramming environment is supposed to be invisible to the user processes; that is, the actions carried out by each process should proceed in the same was as if the process had the entire machine to itself.

This raises the following issues:

- Process model: The state of an inactive process has to be encoded & saved in a process table so that the process can be resumed when made active.
- Context switching: How does one carry out the change from one process to another?

- Memory translation: Each process treats the computer's memory as its own private playground. How can we give each process the illusion that it can reference addresses in memory as it wants, but not have them step on each other's toes? The trick is by distinguishing between virtual addresses -- the addresses used in the process code -- & physical addresses -- the actual addresses in memory. Each process is actually given a fraction of physical memory. The memory management unit translates the virtual address in the code to a physical address within the user's space. This translation is invisible to the process.
- Memory management: How does the OS assign sections of physical memory to each process?
- Scheduling: How does the OS choose which process to run when?

Let us briefly review some aspects of program behavior in order to motivate the basic idea of multiprogramming. This is illustrated in Figure 6, indicated by dashed boxes. Idealized serial execution of two programs, with no inter-program idle times, is depicted in Figure 6(a). For comparison purposes, both programs are assumed to have identical behavior with regard to processor & I/O times & their relative distributions. As Figure 6(a) suggests, serial execution of programs causes either the processor or the I/O devices to be idle at some time even if the input job stream is never empty. One way to attack this problem is to assign some other work to the processor & I/O devices when they would otherwise be idling.

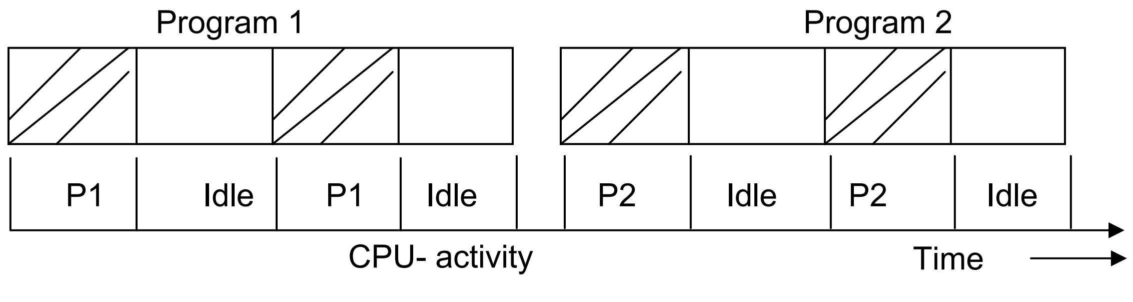


Figure 6 (a) Sequential execution

Figure 6(b) illustrates a possible scenario of concurrent execution of the two programs introduced in Figure 6(a). It starts with the processor executing the first computational sequence of Program 1. Instead of idling during the subsequent I/O sequence of Program 1, the processor is assigned to the first computational sequence of the Program 2, which is assumed to be in memory & awaiting execution. When this work is done, the processor is assigned to Program 1 again, then to Program 2, & so forth.

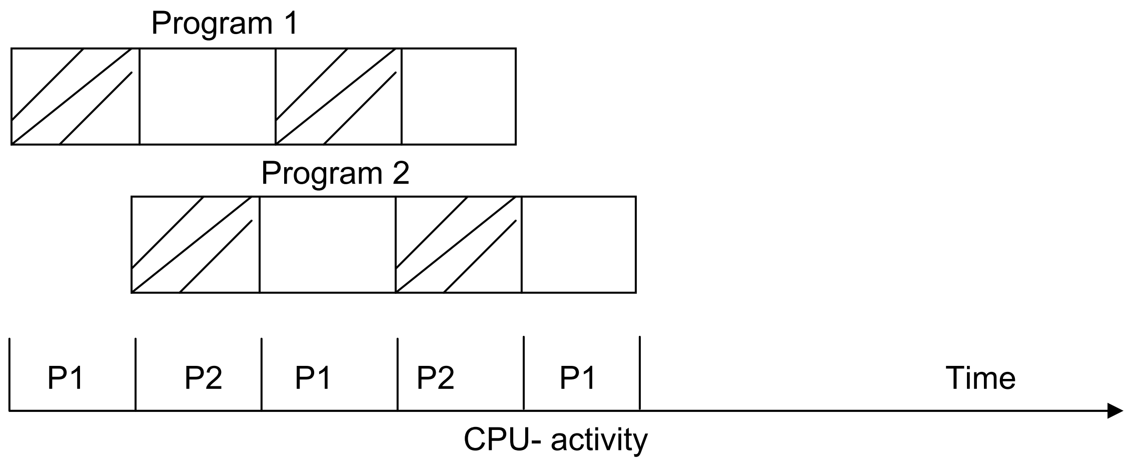


Figure 6(b) Multiprogrammed execution

As Figure 6 suggests, significant performance gains may be achieved by interleaved executing of programs, or multiprogramming, as this mode of operation is usually called. With a single processor, parallel execution of programs is not possible, & at most one program can be in control of the processor at any time. The example presented in Figure 6(b) achieves 100% processor utilization with only two active programs. The number of programs actively competing for resources of a multi-programmed computer system is called the degree of multiprogramming. In principle, higher degrees of multiprogramming should result in higher resource utilization. Time-sharing systems found in many university computer centers provide a typical example of a multiprogramming system.

1.2.5 TYPES OF OPERATING SYSTEMS

Operating system can be classified into various categories on the basis of several criteria, viz. number of simultaneously active programs, number of users working simultaneously, number of processors in the computer system, etc. In the following discussion several types of operating systems are discussed**.**

1.2.5.1 Batch Operating System

Batch processing is the most primitive type of operating system. Batch processing generally requires the program, data, & appropriate system commands to be submitted together in the form of a job. Batch operating systems usually allow little or no interaction between users & executing programs. Batch processing has a greater potential for resource utilization than simple serial processing in computer systems serving multiple users. Due to turnaround delays & offline debugging, batch is not very convenient for program development. Programs that do not require interaction & programs with long execution times may be served well by a batch operating system. Examples of such programs include payroll, forecasting, statistical analysis, & large scientific number-crunching programs. Serial processing combined with batch like command files is also found on many personal computers. Scheduling in batch is very simple. Jobs are typically processed in order of their submission, that is, first-come first-served fashion.

Memory management in batch systems is also very simple. Memory is usually divided into two areas. The resident portion of the OS permanently occupies one of them, & the other is used to load transient programs for execution. When a transient program terminates, a new program is loaded into the same area of memory. Since at most one program is in execution at any time, batch systems do not require any time-critical device management. For this reason, many serial & I/O & ordinary batch operating systems use simple, program controlled method of I/O. The lack of contention for I/O devices makes their allocation & deallocation trivial.

Batch systems often provide simple forms of file management. Since access to files is also serial, little protection & no concurrency control of file access in required.

1.2.5.2 Multiprogramming Operating System

A multiprogramming system permits multiple programs to be loaded into memory & execute the programs concurrently. Concurrent execution of programs has a significant potential for improving system throughput & resource utilization relative to batch & serial processing. This potential is realized by a class of operating systems that multiplex resources of a computer system among a multitude of active programs. Such operating systems usually have the prefix multi in their names, such as multitasking or multiprogramming.

1.2.5.3 Multitasking Operating System

An instance of a program in execution is called a process or a task. A multitasking OS is distinguished by its ability to support concurrent execution of two or more active processes. Multitasking is usually implemented by maintaining code & data of several processes in memory simultaneously, & by multiplexing processor & I/O devices among them. Multitasking is often coupled with hardware & software support for memory protection in order to prevent erroneous processes from corrupting address spaces & behavior of other resident processes. Allows more than one program to run concurrently. The ability to execute more than one task at the same time, a task being a program is called as multitasking. The terms multitasking & multiprocessing are often used interchangeably, although multiprocessing sometimes implies that more than one CPU is involved. In multitasking, only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same time. There are two basic types of multitasking: preemptive & cooperative. In preemptive multitasking, the OS parcels out CPU time slices to each program. In cooperative multitasking, each program can control the CPU for as long as it needs it. If a program is not using the CPU, however, it can allow another program to use it temporarily. OS/2, Windows 95, Windows NT, & UNIX use preemptive multitasking, whereas Microsoft Windows 3.x & the MultiFinder use cooperative multitasking.

1.2.5.4 Multi-user Operating System

Multiprogramming operating systems usually support multiple users, in which case they are also called multi-user systems. Multi-user operating systems provide facilities for maintenance of individual user environments & therefore require user accounting. In general, multiprogramming implies multitasking, but multitasking does not imply multi-programming. In effect, multitasking operation is one of the mechanisms that a multiprogramming OS employs in managing the totality of computer-system resources, including processor, memory, & I/O devices. Multitasking operation without multi-user support can be found in operating systems of some advanced personal computers & in real-time systems. Multi-access operating systems allow simultaneous access to a computer system through two or more terminals. In general, multi-access operation does not necessarily imply multiprogramming. An example is provided by some dedicated transaction-processing systems, such as airline ticket reservation systems, that support hundreds of active terminals under control of a single program.

In general, the multiprocessing or multiprocessor operating systems manage the operation of computer systems that incorporate multiple processors. Multiprocessor operating systems are multitasking operating systems by definition because they support simultaneous execution of multiple tasks (processes) on different processors. Depending on implementation, multitasking may or may not be allowed on individual processors. Except for management & scheduling of multiple processors, multiprocessor operating systems provide the usual complement of other system services that may qualify them as time-sharing, real-time, or a combination operating system.

1.2.5.5 Multithreading

Allows different parts of a single program to run concurrently. The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other.

1.2.5.6 Time-sharing system

Time-sharing is a popular representative of multi-programmed, multi-user systems. In addition to general program-development environments, many large computer-aided design & text-processing systems belong to this category. One of the primary objectives of multi-user systems in general, & time-sharing in particular, is good terminal response time. Giving the illusion to each user of having a machine to oneself, time-sharing systems often attempt to provide equitable sharing of common resources. For example, when the system is

loaded, users with more demanding processing requirements are made to wait longer.
This philosophy is reflected in the choice of scheduling algorithm. Most time-sharing systems use time-slicing scheduling. In this approach, programs are executed with rotating priority that increases during waiting & drops after the service is granted. In order to prevent programs from monopolizing the processor, a program executing longer than the system-defined time slice is interrupted by the OS & placed at the end of the queue of waiting programs. This mode of operation generally provides quick response time to interactive programs. Memory management in time-sharing systems provides for isolation & protection of co-resident programs. Some forms of controlled sharing are sometimes provided to conserve memory & possibly to exchange data between programs. Being executed on behalf of different users, programs in time-sharing systems generally do not have much need to communicate with each other.

As in most multi-user environments, allocation & de-allocation of devices must be done in a manner that preserves system integrity & provides for good performance.

1.2.5.7 Real-time systems

Real time systems are used in time critical environments where data must be processed extremely quickly because the output influences immediate decisions. Real time systems are used for space flights, airport traffic control, industrial processes, sophisticated medical equipments, telephone switching etc. A real time system must be 100 percent responsive in time. Response time is measured in fractions of seconds. In real time systems the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the results is produced. If the timing constraints of the system are not met, system failure is said to have occurred. Real-time operating systems are used in environments where a large number of events, mostly external to the computer system, must be accepted & processed in a short time or within certain deadlines.

A primary objective of real-time systems is to provide quick event-response times, & thus meet the scheduling deadlines. User convenience & resource utilization are of secondary concern to real-time system designers. It is not uncommon for a real-time system to be expected to process bursts of thousands of interrupts per second without missing a single event. Such requirements usually cannot be met by multi-programming alone, & real-time operating systems usually rely on some specific policies & techniques for doing their job.

The Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is normally allocated to the highest-priority process among those that are ready to execute. Higher-priority processes usually preempt execution of the lower-priority processes. This form of scheduling, called priority-based preemptive scheduling, is used by a majority of real-time systems. Unlike, say, time-sharing, the process population in real-time systems is fairly static, & there is comparatively little moving of programs between primary & secondary storage. On the other hand, processes in real-time systems tend to cooperate closely, thus necessitating support for both separation & sharing of memory. Moreover, as already suggested, time-critical device management is one of the main characteristics of real-time systems. In addition to providing sophisticated forms of interrupt management & I/O buffering, real-time operating systems often provide system calls to allow user processes to connect themselves to interrupt vectors & to service events directly. File management is usually found only in larger installations of real-time systems. In fact, some embedded real-time systems, such as an onboard automotive controller, may not even have any secondary storage. The primary objective of file management in real-time systems is usually speed of access, rather then efficient utilization of secondary storage.

1.2.5.8 Combination of operating systems

Different types of OS are optimized or geared up to serve the needs of specific environments. In practice, however, a given environment may not exactly fit any of the described molds. For instance, both interactive program development & lengthy simulations are often encountered in university computing centers. For this reason, some commercial operating systems provide a combination of described services. For example, a time-sharing system may support interactive users & also incorporate a full-fledged batch monitor. This allows computationally intensive non-interactive programs to be run concurrently with interactive programs. The common practice is to assign low priority to batch jobs & thus execute batched programs only when the processor would otherwise be idle. In other words, batch may be used as a filler to improve processor utilization while accomplishing a useful service of its own. Similarly, some time-critical events, such as receipt & transmission of network data packets, may be handled in real-time fashion on systems that otherwise provide time-sharing services to their terminal users.

1.2.5.9 Distributed Operating Systems

A distributed computer system is a collection of autonomous computer systems capable of communication & cooperation via their hardware & software interconnections. Historically, distributed computer systems evolved from computer networks in which a number of largely independent hosts are connected by communication links & protocols. A distributed OS governs the operation of a distributed computer system & provides a virtual machine abstraction to its users. The key objective of a distributed OS is transparency. Ideally, component & resource distribution should be hidden from users & application programs unless they explicitly demand otherwise. Distributed operating systems usually provide the means for system-wide sharing of resources, such as computational capacity, files, & I/O devices. In addition to typical operating-system services provided at each node for the benefit of local clients, a distributed OS may facilitate access to remote resources, communication with remote processes, & distribution of computations. The added services necessary for pooling of shared system resources include global naming, distributed file system, & facilities for distribution.

1.2.5.6 SYSTEM CALLS

System calls are kernel level service routines for implementing basic operations performed by the operating system. Below are mentioned some of several generic system calls that most operating systems provide.

CREATE (processID, attributes);

In response to the CREATE call, the OS creates a new process with the specified or default attributes & identifier. A process cannot create itself-because it would have to be running in order to invoke the OS, & it cannot run before being created. So a process must be created by another process. In response to the CREATE call, the OS obtains a new PCB from the pool of free memory, fills the fields with provided and/or default parameters, & inserts the PCB into the ready list-thus making the specified process eligible to run. Some of the parameters definable at the process-creation time include: (a) Level of privilege, such as system or user (b) Priority (c) Size & memory requirements (d) Maximum data area and/or stack size (e) Memory protection information & access rights (f) Other system-dependent data

Typical error returns, implying that the process was not created as a result of this call, include: wrongID (illegal, or process already active), no space for PCB (usually transient; the call may be retries later), & calling process not authorized to invoke this function.

DELETE (process ID);

DELETE invocation causes the OS to destroy the designated process & remove it from the system. A process may delete itself or another process. The OS reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process, & performing whatever other housekeeping is necessary. Following this process, the PCB is removed from its place of residence in the list & is returned to the free pool. This makes the designated process dormant. The DELETE service is normally invoked as a part of orderly program termination.

To relieve users of the burden & to enhance probability of programs across different environments, many compilers compile the last END statement of a main program into a DELETE system call.

Almost all multiprogramming operating systems allow processes to terminate themselves, provided none of their spawned processes is active. OS designers differ in their attitude toward allowing one process to terminate others. The issue here is none of convenience & efficiency versus system integrity. Allowing uncontrolled use of this function provides a malfunctioning or a malevolent process with the means of wiping out all other processes in the system. On the other hand, terminating a hierarchy of processes in a strictly guarded system where each process can only delete itself, & where the parent must wait for children to terminate first, could be a lengthy operation indeed. The usual compromise is to permit deletion of other processes but to restrict the range to the members of the family, to lower-priority processes only, or to some other subclass of processes.

Possible error returns from the DELETE call include: a child of this process is active (should terminate first), wrongID (the process does not exist), & calling process not authorized to invoke this function.

Abort (processID);

ABORT is a forced termination of a process. Although a process could conceivably abort itself, the most frequent use of this call is for involuntary terminations, such as removal of a malfunctioning process from the system. The OS performs much the same actions as in DELETE, except that it usually furnishes a register & memory dump, together with some information about the identity of the aborting process & the reason for the action. This information may be provided in a file, as a message on a terminal, or as an input to the system crash-dump analyzer utility. Obviously, the issue of restricting the authority to abort other processes, discussed in relation to the DELETE, is even more pronounced in relation to the ABORT call.

Error returns for ABORT are practically the same as those listed in the discussion of the DELETE call.

FORK/JOIN

Another method of process creation & termination is by means of the FORK/JOIN pair, originally introduced as primitives for multiprocessor systems. The FORK operation is used to split a sequence of instructions into two concurrently executable sequences. After reaching the identifier specified in FORK, a new process (child) is created to execute one branch of the forked code while the creating (parent) process continues to execute the other. FORK usually returns the identity of the child to the parent process, & the parent can use that identifier to designate the identity of the child whose termination it wishes to await before invoking a JOIN operation. JOIN is used to merge the two sequences of code divided by the FORK, & it is available to a parent process for synchronization with a child.

The relationship between processes created by FORK is rather symbiotic in the sense that they execute from a single segment of code, & that a child usually initially obtains a copy of the variables of its parent.

SUSPEND (processKD);

The SUSPEND service is called SLEEP or BLOCK in some systems. The designated process is suspended indefinitely & placed in the suspended state. It does, however, remain in the system. A process may suspend itself or another process when authorized to do so by virtue of its level of privilege, priority, or family membership. When the running process suspends itself, it in effect voluntarily surrenders control to the operating system. The OS responds by inserting the target process's PCB into the suspended list & updating the PCB state field accordingly.

Suspending a suspended process usually has no effect, except in systems that keep track of the depth of suspension. In such systems, a process must be resumed at least as many times as if was suspended in order to become ready. To implement this feature, a suspend-count field has to be maintained in each PCB. Typical error returns include: process already suspended, wrongID, & caller not authorized.

RESUME (processID)

The RESUME service is called WAKEUP is some systems. This call resumes the target process, which is presumably suspended. Obviously, a suspended process cannot resume itself, because a process must be running to have its OS call processed. So a suspended process depends on a partner process to issue the RESUME. The OS responds by inserting the target process's PCB into the ready list, with the state updated. In systems that keep track of the depth of suspension, the OS first increments the suspend count, moving the PCB only when the count reaches zero.

The SUSPEND/RESUME mechanism is convenient for relatively primitive & unstructured form of inter-process synchronization. It is often used in systems that do not support exchange of signals. Error returns include: process already active, wrongID, & caller not authorized.

DELAY (processID, time);

The system call DELAY is also known as SLEEP. The target process is suspended for the duration of the specified time period. The time may be expressed in terms of system clock ticks that are system-dependent & not portable, or in standard time units such as seconds & minutes. A process may delay itself or, optionally, delay some other process.

The actions of the OS in handling this call depend on processing interrupts from the programmable interval timer. The timed delay is a very useful system call for implementing time-outs. In this application a process initiates an action & puts itself to sleep for the duration of the time-out. When the delay (time-out) expires, control is given back to the calling process, which tests the outcome of the initiated action. Two other varieties of timed delay are cyclic rescheduling of a process at given intervals (e.g,. running it once every 5 minutes) & time-of-day scheduling, where a process is run at a specific time of the day. Examples of the latter are printing a shift log in a process-control system when a new crew is scheduled to take over, & backing up a database at midnight.

The error returns include: illegal time interval or unit, wrongID, & called not authorized. In Ada, a task may delay itself for a number of system clock ticks (system-dependent) or for a specified time period using the pre-declared floating-point type TIME. The DELAY statement is used for this purpose.

GET_ATTRIBUTES (processID, attribute_set);

GET_ATTRIBUTES is an inquiry to which the OS responds by providing the current values of the process attributes, or their specified subset, from the PCB. This is normally the only way for a process to find out what its current attributes are, because it neither knows where its PCB is nor can access the protected OS space where the PCBs are usually kept.

This call may be used to monitor the status of a process, its resource usage & accounting information, or other public data stored in a PCB. The error returns include: no such attribute, wrongID, & caller not authorized. In Ada, a task may examine the values of certain task attributes by means of reading the pre-declared task attribute variables, such as T'ACTIVE, T'CALLABLE, T'PRIORITY, & T'TERMINATED, where T is the identity of the target task.

CHANGE_PRIORITY (processID, new_priority);

CHANGE_PRIORITY is an instance of a more general SET_PROCESS_ATTRIBUTES system call. Obviously, this call is not implemented in systems where process priority is static.

Run-time modifications of a process's priority may be used to increase or decrease a process's ability to compete for system resources. The idea is that priority of a process should rise & fall according to the relative importance of its momentary activity, thus making scheduling more responsive to changes of the global system state. Low-priority processes may abuse this call, & processes competing with the OS itself may corrupt the whole system. For these reasons, the authority to increase priority is usually restricted to changes within a certain range. For example, maximum may be specified, or the process may not exceed its parent's or group priority. Although changing priorities of other processes could be useful, most implementations restrict the calling process to manipulate its own priority only.

The error returns include: caller not authorized for the requested change & wrong ID. In Ada, a task may change its own priority by calling the SET_PRIORITY procedure, which is pre-declared in the language.

1.4 SUMMARY

Operating system is also known as resource manager because its prime responsibility is to manage the resources of the computer system i.e. memory, processor, devices and files. In addition to these, operating system provides an interface between the user and the bare machine. Following the course of the conceptual evolution of operating systems, we have identified the main characteristics of the program-execution & development environments provided by the bare machine, serial processing, including batch & multiprogramming.

On the basis of their attributes & design objectives, different types of operating systems were defined & characterized with respect to scheduling & management of memory, devices, & files. The primary concerns of a time-sharing system are equitable sharing of resources & responsiveness to interactive requests. Real-time operating systems are mostly concerned with responsive handling of external events generated by the controlled system. Distributed operating systems provide facilities for global naming & accessing of resources, for resource migration, & for distribution of computation.

Typical services provided by an OS to its users were presented from the point of view of command-language users & system-call users. In general, system calls provide functions similar to those of the command language but allow finer gradation of control.

1.5. SELF ASSESMENT QUESTIONS (SAQ)

1. What are the objectives of an operating system? Discuss.
2. Discuss modular approach of development of an operating system.
3. Present a hierarchical structure of an operating system.
4. What is an extended machine view of an operating system?
5. Discuss whether there are any advantages of using a multitasking operating system, as opposed to a serial processing one.
6. What are the major functions performed by an operating system? Explain.

1.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 2 ***Writer: Dr. Rakesh Kumar***

File System - I ***Vetter: Prof. Dharminder Kr.***

2.0 Objectives

A file is a logical collection of information and file system is a collection of files. The objective of this lesson is to discuss the various concepts of file system and make the students familiar with the different techniques of file allocation and access methods. We also discuss the ways to handle file protection, which is necessary in an environment where multiple users have access to files and where it is usually desirable to control by whom and in what ways files may be accessed.

2.1 Introduction

The file system is the most visible aspect of an operating system. 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). It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists to two distinct parts: a collection of files, each storing related data and a directory structure, which organizes and provides information about all the files in the system. Some file systems have a third part, partitions, which are used to separate physically or logically large collections of directories.

Nutt describes the responsibility of the file manager and defines the file, the fundamental abstraction of secondary storage:

"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."

2.2 Presentation of Contents

2.2.1 File Concepts

2.2.1.1 File Operations

2.2.1.2 File Naming

2.2.1.3 File Types

2.2.1.4 Symbolic Link

2.2.1.5 File Sharing & Locking

2.2.1.6 File-System Structure

2.2.1.7 File-System Mounting

2.2.1.8 File Space Allocations

2.2.1.8.1 Contagious Space Allocation

2.2.1.8.2 Linked Allocation

2.2.1.8.3 Indexed Allocation

2.2.1.8.4 Performance

2.2.1.9 File Attributes

2.2.2 Access Methods

2.2.2.1 Sequential Access

2.2.2.2 Index-sequential

2.2.2.3 Direct Access

2.2 PRESENTATION OF CONTENTS

2.2.1 FILE CONCEPTS

The most important function of an operating system is the effective management of information. The modules of the operating system dealing with the management of information are known as file system. The file system provides the mechanism for online storage and access to both data and programs. The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently. The desirable features of a file system are:

4. Minimal I/O operations.
5. Flexible file naming facilities.

6. Automatic allocation of file space.
7. Dynamic allocation of file space.
8. Unrestricted flexibility between logical record size and physical block size.
9. Protection of files against illegal forms of access.
10. Static and dynamic sharing of files.
11. Reliable storage of files.

This lesson is primarily concerned with issues concerning file storage and access on the most common secondary storage medium, the disk.

A file is a collection of related information units (records) treated as a unit. A record is itself a collection of related data elements (fields) treated as a unit. A field contains a single data item.

So file processing refers to reading/writing of records in a file and processing of the information in the fields of a record.

2.2.1.1 File operations

Major file operations are performed are as follows:

- Read operation: Read information contained in the file.
- Write operation: Write new information into a file at any point or overwriting existing information in a file.
- Deleting file: Delete a file and release its storage space for use in other files.
- Appending file: Write new information at the end of a file.
- Execute
- Coping file
- Renaming file
- Moving file
- Creating file
- Merging files
- Sorting file
- Comparing file

2.2.1.2 File Naming

Each file is a distinct entity and therefore a naming convention is required to distinguish one from another. The operating systems generally employ a naming system for this purpose. In fact, there is a naming convention to identify each resource in the computer system and not files alone.

2.2.1.3 File Types

The files under UNIX can be categorized as follows:

- Ordinary files.
- Directory files.
- Special files.
- FIFO files.

Ordinary Files

Ordinary files are the one, with which we all are familiar. They may contain executable programs, text or databases. You can add, modify or delete them or remove the file entirely.

Directory Files

Directory files, as discussed earlier also represent a group of files. They contain list of file names and other information related to these files. Some of the commands, which manipulate these directory files, differ from those for ordinary files.

Special Files

Special files are also referred to as device files. These files represent physical devices such as terminals, disks, printers and tape-drives etc. These files are read from or written into just like ordinary files, except that operation on these files activates some physical devices. These files can be of two types (i) character device files and (ii) block device file. In character device files data are handled character by character, as in case of terminals and printers. In block device files, data are handled in large chunks of blocks, as in the case of disks and tapes.

FIFO Files

FIFO (first-in-first-out) are files that allow unrelated processes to communicate with each other. They are generally used in applications where the communication path is in only one direction, and several processes need to communicate with a single process. For an example of FIFO file, take the pipe in UNIX. This allows transfer of data between processes in a first-in-first-out manner. A pipe takes the output of the first process as the input to the next process, and so on.

2.2.1.4 Symbolic Link

A link is effectively a pointer or an alias to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name (a symbolic link). When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We resolve the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry (or by their having a special type on systems that support types), and are effectively named indirect pointers.

A symbolic link can be deleted without deleting the actual file it links. There can be any number of symbolic links attached to a single file.

Symbolic links are helpful in sharing a single file called by different names. Each time a link is created, the reference count in its inode is incremented by one. Whereas deletion of link decreases file the count by one. The operating system denies deletion of such files whose reference count is not 0, thereby meaning that the file is in use.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated link is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list - we need to keep only a count of the number of references. A new link or directory entry increments the reference counts; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for non-symbolic links, or hard links, keeping a reference count in the file information block or inode). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories links. For example, in MS-DOS, the directory structure is a tree structure.

2.2.1.5 File Sharing and Locking

The owner of a file uses the access control list of the file to authorize some other users to access the file. In a multi-user environment a file is required to be shared among more than one user. There are several techniques and approaches to affect this operation. File sharing can occur in two modes (i) sequential sharing and (ii) concurrent sharing. Sequential sharing occurs when authorized users access a shared file one after another. So any change made by a user is reflected to other users also. Concurrent sharing occurs when two or more users access a file over the same period of time. Concurrent sharing may be implemented in one of the following three forms:

(a) Concurrent sharing using immutable files: In it any program cannot modify the file being shared.

(b) Concurrent sharing using single image mutable files: An image is a view of a file. All programs concurrently sharing the file see the same image of the file. So changes made by one program are also visible to other programs sharing the file.

(c) Concurrent sharing using multiple image mutable files: Each program accessing the file has its own image of the file. So many versions of the file at a time may exist and updates made by a user may not be visible to some concurrent user.

There are three different modes to share a file:

- Read only: In this mode the user can only read or copy the file.
- Linked shared: In this mode all the users sharing the file can make changes in this file but the changes are reflected in the order determined by the operating

systems.

- Exclusive mode: In this mode a single user who can make the changes (while others can only read or copy it) acquires the file.

Another approach is to share a file through symbolic links. This approach poses a couple of problems - concurrent updation problem, deletion problem. If two users try to update the same file, the updating of one of them will be reflected at a time. Besides, another user must not delete a file while it is in use.

File locking gives processes the ability to implement mutually exclusive access to a file. Locking is mechanism through which operating systems ensure that the user making changes to the file is the one who has the lock on the file. As long as the lock remains with this user, no other user can alter the file. Locking can be limited to files as a whole or parts of a file. Locking may apply to any access or different levels of locks may exist such as read/write locks etc.

2.2.1.6 File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. To improve I/O efficiency, I/O transfers between memory and disks are performed in units of blocks. Each block is one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4096 bytes; usually, they are 512 bytes. The blocking method determines how a file's records are allocated into blocks:

Fixed blocking: An integral number of fixed-size records are stored in each block. No record may be larger than a block.

Unspanned blocking: Multiple variable size records can be stored in each block but no record may span multiple blocks.

Spanned blocking: Records may be stored in multiple blocks. There is no limit on the size of a record.

Disks have two important characteristics that make them a convenient medium for storing multiple files:

(a) They can be rewritten in place; it is possible to read a block from the disk, to modify the block, and to write it back into the same place.

(b) One can access directly any given block of information on the disk. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another added requires only moving the read-write heads and waiting for the disk to rotate.

To provide an efficient and convenient access to the disk, the operating system imposes a file system to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems.

(a) How the file system should look to the user? This task involves the definition of a file and its attributes, operations allowed on a file and the directory structure for organizing the files.

(b) Algorithms and data structure must be created to map the logical file system onto the physical secondary storage devices.

2.2.1.7 File-System Mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. The mount procedure is straightforward. The operating system is given the name of the device and the location within the file structure at which to attach the file system (called the mount point). For instance, on the UNIX system, a file system containing user's home directory might be mounted as /home; then, to access the directory structure within that file system, one could precede the directory names with /home, as in /home/jane. Mounting that file system under /users would result in the path name /users/jane to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate. Consider the actions of the Macintosh Operating System.

Whenever the system encounters a disk for the first time (hard disks are found at boot time, floppy disks ate seen when they are inserted into the drive), the Macintosh Operating System searches for a file system on the device. If it finds one, it automatically mounts the file system at the boot-level, adds a folder icon to the screen labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus to display the newly mounted file system.

2.2.1.8 File space allocations

The direct-access nature of disks allows flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. There are three major methods of allocating disk space:

(a) Contiguous space allocation

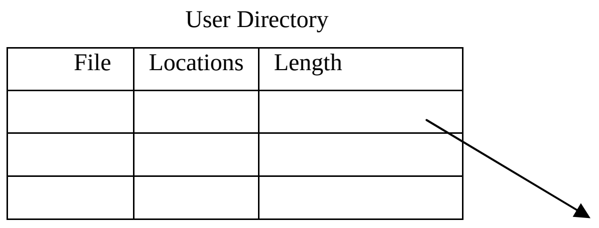
(b) Linked allocation

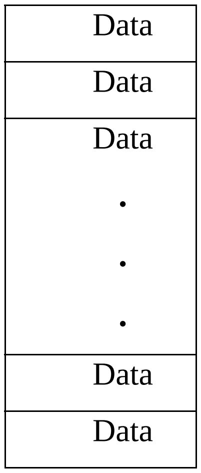
(c) Indexed allocation

Each method has its advantages and disadvantages. Accordingly some systems support all three. More common system will use one particular method for all files.

2.2.1.8.1 Contiguous space Allocation

The simplest scheme is contiguous allocation. The logical blocks pf a file are stored in a partition of contiguous physical blocks. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b+1 after block b normally requites 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 movement. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal. The disk address and length (in block units) of the first block define contiguous allocation of the file. If the file is n blocks long, and starts at location b, then it occupies block 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. Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b+i. So contiguous space allocation easily supports both sequential and direct access.





The major problem with contiguous allocation is locating the space for a new file. The contiguous disk space-allocation problem can be seen to be particular application of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes. First-fit (This strategy allocates the first available space that is big enough to accommodate file. Search may start at beginning of set of holes or where previous first-fit ended. Searching stops as soon as it finds a free hole that is large enough) and best-fit (This strategy allocates the smallest hole that is big enough to accommodate file. Entire list ordered by size is searched & matching smallest left over hole is chosen) are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are more efficient than worst-fit (This strategy allocates the largest hole. Entire list is searched. It chooses largest left over hole) in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms suffer from the problem of external fragmentation i.e. the tendency to develop a large number of small holes. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

Some older microcomputer systems used contiguous allocation on floppy disks. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run a repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole.

The scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

This is not all, there are other problems with contiguous allocation. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found 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 (copying an existing file, for example); in general, however, the size of an output file may be difficult to estimate.

If too little space is allocated to a file, it may be found that file cannot be extended. Especially with only a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in space. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may prove costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.

The other possibility is to find a larger hole, to copy the contents of the file to the new space and release the previous space. This series of actions may be repeated as long as space exists, although it can also be time-consuming. Notice, however, that in this case the user never needs to be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly.

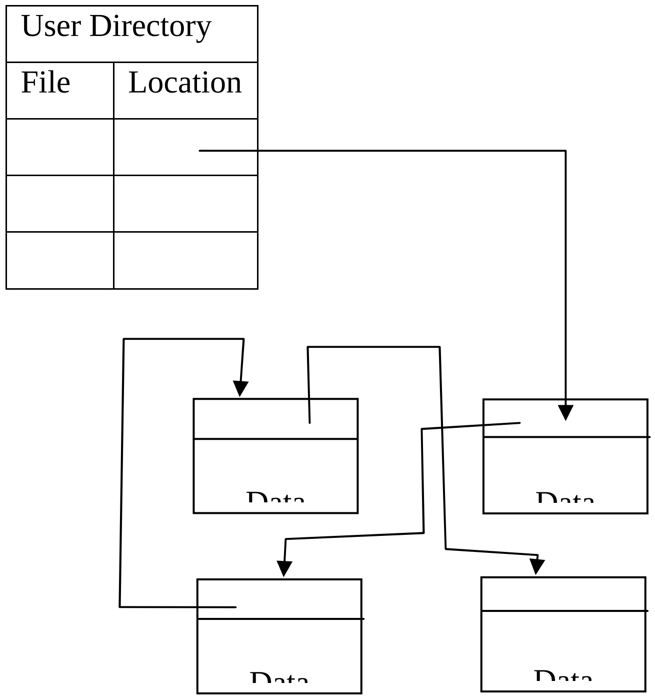
Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time. The file, therefore, has a large amount of internal fragmentation.

To avoid several of these drawbacks, some operating systems use a modified contiguous allocation scheme, in which a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, called an extent, is added to the initial allocation. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and de-allocated in turn.

2.2.1.8.2 Linked Allocation

In linked allocation, file is not stored on a contiguous set of blocks, rather the physical blocks in which a file is stored may be scattered throughout the secondary storage devices. Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes. To create anew file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialised to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. Notice also that there is no need to declare the size of a file when a file is created. A file can continue to grow as long as there are free blocks. Consequently, it is never necessary to compact disk space. Linked allocation suffers from some disadvantages, however. The major problem is that it can be used effectively for only sequential-access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read, and sometimes a disk seek also. Consequently, it is inefficient to support a direct-access capability for linked allocation files.



Space required for the pointers is another disadvantage to linked allocation. If a pointer requires 4 bytes out of a 512-byte block, then ((4 / 512) * 100 = 0.78) percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it otherwise would. The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head-seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially fully than when a block is partially full. Clusters can be used to improve the disk access time for many other algorithms, so they are used in most operating systems.

Yet another problem is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged. A bug in the operating- system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or, to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation on the linked allocation method is the use of a file-allocation table (FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition is reserved to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value.

Note that the FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the partition to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

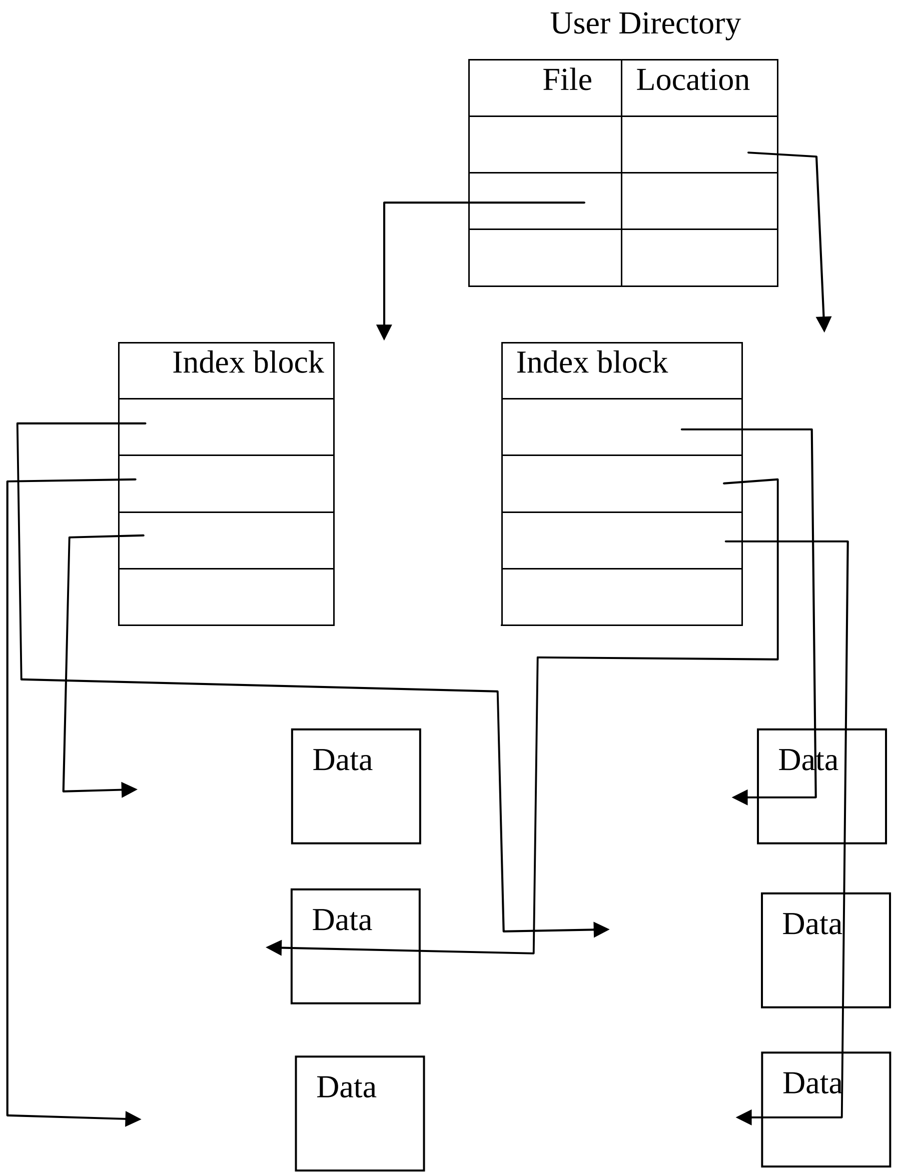
2.2.1.8.3 Indexed Allocation

Although linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together into one location, called the index block. Indexed allocation is a variant of linked allocation.

Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block (See following figure). To read the i^{th} block, we use the pointer in the i^{th} index-block entry to find and read the desired block.

When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.



Indexed allocation of disk space

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Note that indexed allocation schemes suffer from some of the same performance problems, as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

2.2.1.8.4 Performance

To evaluate the performance of allocation methods, two important criteria are storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

One difficulty in comparing in performance of the various systems is determining how the systems will be used – in a sequential access manner or random access. A system with mostly sequential access should use a method different from that for a system with mostly random access. For any type of access, contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the i^{th} block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the i^{th} block might require i disk reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential access by linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared with it.

2.2.1.9 File attributes

Attributes are properties of a file. The operating system treats a file according to its attributes. Following are a few common attributes of a file:

- H for hidden
- A for archive
- D for directory
- X for executable
- R for read only

These attributes can be used in combination also.

2.2.2 ACCESS METHODS

Files store information, which is when required, may be read into the main memory. There are several different ways in which the data stored in a file may be accessed for reading and writing. The operating system is responsible for supporting these file access methods. The fundamental methods for accessing information in the file are (a) sequential access: in it information in the file must be accessed in the order it is stored in the file, (b) direct access, and (c) index sequential access.

2.2.2.1 Sequential access

A sequential file is the most primitive of all file structures. It has no directory and no linking pointers. The records are generally organized in a specific sequence according to the key field. In other words, a particular attribute is chosen whose value will determine the order of the records. Access proceeds sequentially from start to finish. Operations to read or write the file need not specify the logical location within the file, because operating system maintains a file pointer that determines the location of the next access. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate. Use of sequential file requires data to be sorted in a desired sequence according to the key field before storing or processing them.

Its main advantages are:

- It is easy to implement
- It provides fast access to the next record if the records are to be accessed using lexicographic order.

Its disadvantages are:

- It is difficult to update and insertion of a new record may require moving a large proportion of the file
- Random access is extremely slow.

Sometimes a file is considered to be sequentially organised despite the fact that it is not ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file and therefore pose no difficulty to updating. Sequential files are advisable if the applications are sequential by nature.

2.2.2.2 Index-sequential

An index-sequential file each record is supposed to have a unique key and the set of records may be ordered sequentially by a key. An index is maintained to determine the location of a record from its key value. Each key value appears in the index with the associated address of its record. To access a record with key k, the index entry containing k is found by searching the index and the disk address mentioned in the entry is used to access the record.

In the following figure an employee file is illustrated where records are arranged in ascending order according to the employee #.

Track #	
1	1 2 5 8 16 20 25 30 32 36
2	38 40 41 43 44 45 50 52
3	53 57 59 60 62 64 67 70

A track index is maintained as shown in the following figure to speed up the search:

Track	Low	High
1	1	36
2	38	52
3	53	70

For example, to locate the record of employee # 41, index is searched. It is evident from the index that the record of employee #41 will be on track no.2 because it has the lowest key value 48 and highest key value 52.

In the literature an index-sequential file is usually thought of as a sequential file with a hierarchy of indices. For example, there might be three levels of indexing: track, cylinder and master. Each entry in the track index will contain enough information to locate the start of the track, and the key of the last record in the track, which is also normally the highest value on that track. There is a track index for each cylinder. Each entry in the cylinder index gives the last record on each cylinder and the address of the track index for that cylinder. If the cylinder index itself is stored on tracks, then the master index will give the highest key referenced for each track of the cylinder index and the starting address of that track. No mention has been made of the possibility of overflow during an updating process. Normally provision is made in the directory to administer an overflow area. This of course increases the number of book-keeping entries in each entry of the index.

2.2.2.3 Direct access

In direct access file organization, any records can be accessed irrespective of the current position in the file. Direct access files are created on direct access storage devices. Whenever a record is to be inserted, its key value is mapped into an address using a hashing function. On that address record is stored. The advantage of direct access file organization is realized when the records are to be accessed randomly (not sequentially). Otherwise this organization has a number of limitations such as (a) poor utilization of the I/O medium and (b) Time consumption during record address calculation.

2.3 Key words

Contiguous space Allocation: The logical blocks pf a file are stored in a partition of contiguous physical blocks.

Linked allocation: In it each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

Indexed Allocation: In Indexed allocation all the pointers together are stored into one location, called the index block. Each file has its own index block, which is an array of disk-block addresses.

Sequential access: In it information in the file must be accessed in the order it is stored in the file.

Index-sequential: In index-sequential file each record is supposed to have a unique key and the set of records may be ordered sequentially by a key. An index is maintained to determine the location of a record from its key value.

Direct access: In direct access file organization, records can be accessed randomly. The key value of the record is mapped into an address using a hashing function. On that address record is stored.

2.4 SUMMARY

The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently.

The various files can be allocated space on the disk in three ways: through contagious, linked or indexed allocation. Contagious allocation can suffer from external fragmentation. Direct-access is very inefficient with linked-allocation. Indexed allocation may require substantial overhead for its index block. There are many ways in which these algorithms can be optimised.

Free space allocation methods also influence the efficiency of the use of disk space, the performance of the file system and the reliability of secondary storage.

2.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. What do you understand by a file? What is a file system?
2. What are the different modes to share a file?
3. What are the different methods to access the information from a file? Discuss their advantages and disadvantages.
4. What are the advantages of indexed allocation over linked allocation and contiguous space allocation? Explain.
5. Differentiate between first fit, best fit and worst fit storage allocation strategies.

2.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New

Delhi, 2002.

Lesson number: 3 ***Writer: Dr. Rakesh Kumar***

File System - II ***Vetter: Prof. Dharminder Kr.***

3.0 Objectives

The objectives of this lesson are to make the students familiar with directory system and file protection mechanism. After studying this lesson students will become familiar with:

(a) Different types of directory structures.

(b) Different protection structures such as:

- Access Control Matrix
- Access Control Lists

3.1 Introduction

A file system provides the following facilities to its users: (a) Directory structure and file naming facilities, (b) Protection of files against illegal form of access, (c) Static and dynamic sharing of files, and (d) Reliable storage of files. A file system helps the user in organizing the files through the use of directories. A directory may be defined as an object that contains the names of the file system objects. Entries in the directory determine the names associated with a file system object. A directory contains information about a group of files. A typical structure of a directory entry is as under:

File name – Locations Information – Protection Information – Flags

The presences of directories enable file system to support file sharing and protection. Sharing is simply a matter of permitting a user to access the files of other user stored in some other directory. Protection is implemented by permitting the owner of a file to specify which other users may access his files and in what manner. All these issues are discussed in detail in this lesson.

3.2 Presentation of Contents

3.2.1 Hierarchical Directory Systems

3.2.1.1 Directory Structure

3.2.1.2 The Logical Structure of a Directory

3.2.1.2.1 Single-level Directory

3.2.1.2.2 Two-level Directory

3.2.1.2.3 Tree-structured Directories

3.2.1.2.4 Acyclic-Graph Directories
3.2.1.2.5 General Graph Directory
3.2.1.3 Directory Operations
3.2.2 File Protection and Security
3.2.2.1 Type of Access
3.2.2.2 Protection Structure
3.2.2.2.1 Access Control Matrix
3.2.2.2.2 Access Lists and Groups
3.2.2.2.3 Other Protection Approaches

3.2.1 HIERARCHICAL DIRECTORY SYSTEMS

Files are generally stored on secondary storage devices. Numerous files are to be stored on storage of giga-byte capacity. To handle such a huge size of data, there is a need to properly organize the files. The organization, usually, done in two parts. In the first part, a file system may incorporate the notion of a partition, which determines on which device a file will be stored. The file system is broken into partitions, also known as minidisks or volumes. Typically, a disk contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, there may be more than one partition on a disk, each partition acting as a virtual disk. The users do not have to concern themselves with the translating the physical address; the system does the required job.

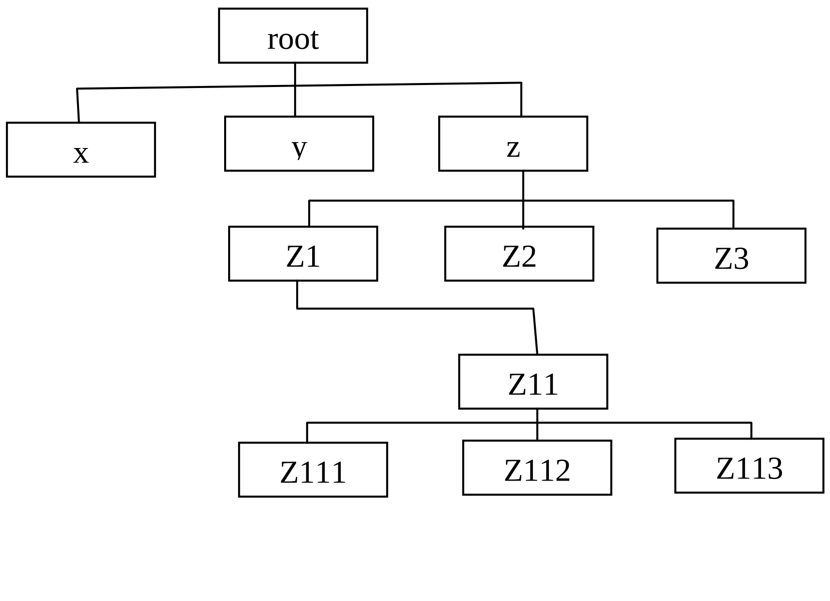


Figure 1 Directory Hierarchy

Partitions contain information about itself in a file called partition table. It also contains information about files and directories on it. Typical file information is name, size, type, location etc. The entries are kept in a device directory or volume table of contents (VTOC).

Directories may be created within another directory. Directories have parent-child relationship as shown in the above Figure 1.

3.2.1.1 Directory Structure

The file systems of computers can be extensive. Some systems store thousands of files on hundreds of gigabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts; first, the file system is broken into in the IBM world or volumes in the PC and Macintosh arenas. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure. In this way, the user needs to be concerned with only the logical directory and file structure, and can ignore completely the problems of physically allocating space for files. For this reason partitions can be thought of as virtual disks.

Second, each partition contains information about files within it. This information is kept in a device directory or volume table of contents. The device directory (more commonly known simply as a "directory") records information such as name, location, size, and type for all files on that partition.

3.2.1.2 The Logical Structure of a Directory

3.2.1.2.1 Single-Level Directory

The simplest directory structure is the single-level tree. A single level tree system has only one directory. All files are contained in the same directory, which is easy to support and understand. Names in that directory refer to files or other non-directory objects. Such a system is practical only on systems with very limited numbers of files. A single-level directory has significant limitations, when the number of files increases or when there is more than one user. Since all files are stored in the same directory, the name given to each file should be unique. If there are two users and they give the same name to their file, then there is a problem.

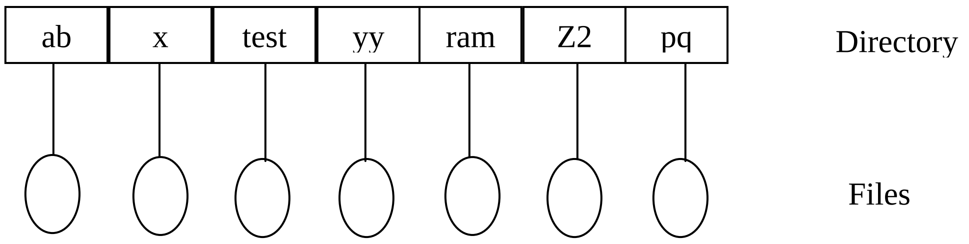


Figure 2 Single-level Directory

Even with a single user, as the number of files increase, it becomes difficult to remember the names of all the files, so as to create only files with unique names. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.

3.2.1.2.2 Two-Level Directory

The disadvantage of a single-level directory is confusion of file names. The standard solution is to create a separate directory for each user. In a two level system, only the root level directory may contain names of directories and all other directories refer only to non-directory objects. In the two-level directory structure, each user has his/her own user file directory (UFD). Each UFD has a similar structure, but lists only the files of a single user. When a user starts or a user logs in, the system's master file directory is searched. The master file directory is indexed by user name or account.

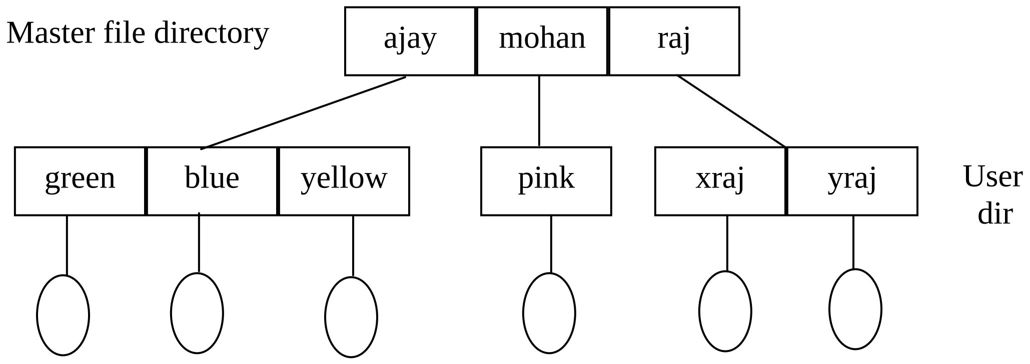


Figure 3 Two-Level Directory

When in a UFD a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as till the filenames within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new user file directory and adds an entry for it to the master file directory. The execution of this program might be restricted to system administrators.

The two-level directory structure solves the name-collision problem, but it still has problems. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent, but is a disadvantage when the users co-operate on some task and to access one user's account by other users is not allowed.

If access is to be permitted, one user must have the ability to name a file in another user's directory.

A two-level directory can be thought of as a tree, or at least an inverted tree. The root of the tree is the master file directory. Its direct descendants are the UFDs. The descendants of the user file directories are the files themselves.

Thus, a user name and a file name define a path name. Every file in the system has a path name. To name a file uniquely, user must know the path name of the file desired.

For example, if user A wishes to access her own test file named test, she can simply refer to test. To access the test file of user B (with directory-entry name userb), however, she might have to refer to /userb/test. Every system has its own syntax for naming files in directories other than the user's own.

There is additional syntax to specify the partition of a file. For instance, in MS-DOS a letter followed by a colon specifies a partition. Thus, file specification might be "C:\userb\bs.test".

Some systems go even further and separate the partition, directory name, and file name parts of the specification. For instance, in VMS, the file "login.com" might be specified as: "u:[sstdeck1]login.com;"

where "u" is the name of the partition, "sst" is the name of the directory, "deck" is the name of subdirectory, and "1", is the version number. Other systems simply treat the partition name as part of the directory name. The first name given is that of the partition, and the rest is the directory and file. For instance, "/u/pbg/test" might specify partition "u”, directory "pbg", and file "test". A special case of this situation occurs in regard to the system files. Those programs provided as a part of the system (loaders, assemblers, compilers, utility routines, libraries, and so on) are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and are executed. Many command interpreters act by simply treating the command as the name of a file to load and execute. As the directory system is defined presently, this file name would be searched for in the current user file directory .One solution would be to copy the system files into each user file directory. However, copying all the system files would be enormously wasteful of space. The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files.

Whenever a file name is given to be loaded, the operating system first searches the local user file directory. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the search path. This idea can be extended, such that the search path contains an unlimited list of directories to search when a command name is given. This method is used in UNlX and MS-DOS.

3.2.1.2.3 Tree-Structured Directories

A tree system allows growth of the tree beyond the second level. Any directory may contain names of additional directories as well as non-directory objects. This generalization allows users to create their own sub-directories and to organize their files accordingly. The MS-DOS system, for instance, is structured as a tree. In fact, a tree is the most common directory structure. The tree has a root directory. Every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.

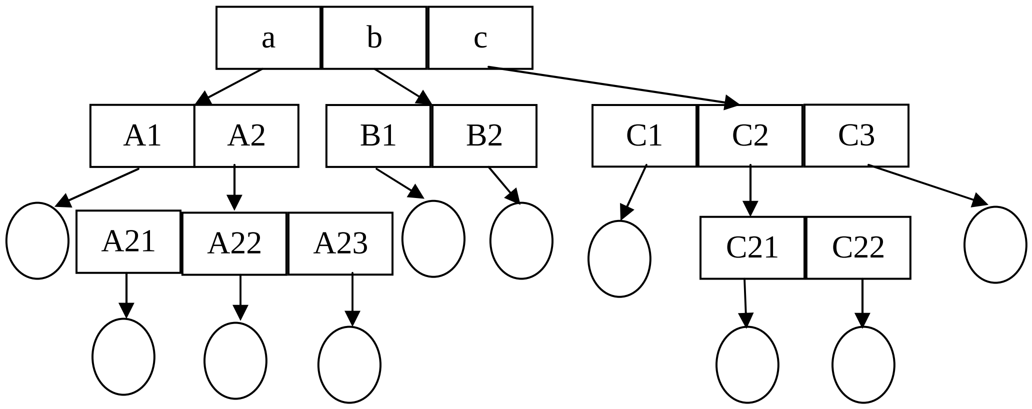


Figure 4 Tree-Structured Directories

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file but it is treated in a special way. All directories have the same internal format, one bit in each directory entry defines the entry as a file (0) or as a subdirectory (1) Special system calls are used to create and delete directories.

In normal use, each user has a current directory .The current directory should contain most of the files that are of current interest to the user. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must either specify a path name or change the current directory to be the directory holding that file. To change the current directory to a different directory, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

Thus, the user can change his current directory whenever he desires. From one change directory system call to the next, all open system calls search the current directory for the specified file.

The initial current directory of a user is designated when the user job starts or the user logs in. The operating system searches the accounting file (or ask) other predefined location to find an entry for this user (for accounting). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user, which specifies the user's initial current directory.

Path names can be of two types: absolute path names or relative path names.

(a) Absolute path: An absolute path name begins at the root and follows a path down to the desired file, giving the directory names on the path. An absolute path name is an unambiguous way of referring to a file. Thus identically named files created by different users differ in their absolute path names.

(b) Relative path: A relative path name defines a path from the current directory.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book or different forms of information for example, the directory programs may contain source programs; the directory bin may store all the binary files. An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. But if the directory to be deleted is not empty, containing files and subdirectories then one of two approaches can be taken. As in MS-DOS, if we want to delete a directory then first of all we have to empty it i.e. delete its contents and If there are any subdirectories, the procedure must be applied recursively to them, so that they can be deleted also. But this approach may be time consuming.

An alternative approach, such as that taken by the UNIX rm command, to provide the option that, when a request is made to delete a directory, and that directory's files and subdirectories are also to be deleted. Note that either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but more dangerous, because an entire director structure may be removed with one command. If that command was issued in error, a large number of files and directories would need to be restored from backup tapes.

With a tree-structured directory system, users can access, in addition their files, the files of other users. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or relative path name. Alternatively, user B could change her current directory be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could define her search path to be (1) her local directory, (2) the system file directory, and user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

A path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system automates the search for executable programs. It maintains a file called the "Desktop File", containing the name and location of all executable programs it has seen. Where a new hard disk or floppy disk is added to the system, or the network accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the pertinent information. This mechanism supports the double-click execution functionality. A double-click on a file causes its creator attribute to be read, and the "Desktop File" to be searched for a match.

3.2.1.2.4 Acyclic-Graph Directories

Sharing of file is another important issue in deciding the directory structure. If more than one user are working on some common project. So the files associated with that project should be placed in a common directory that can be shared among a number of users.

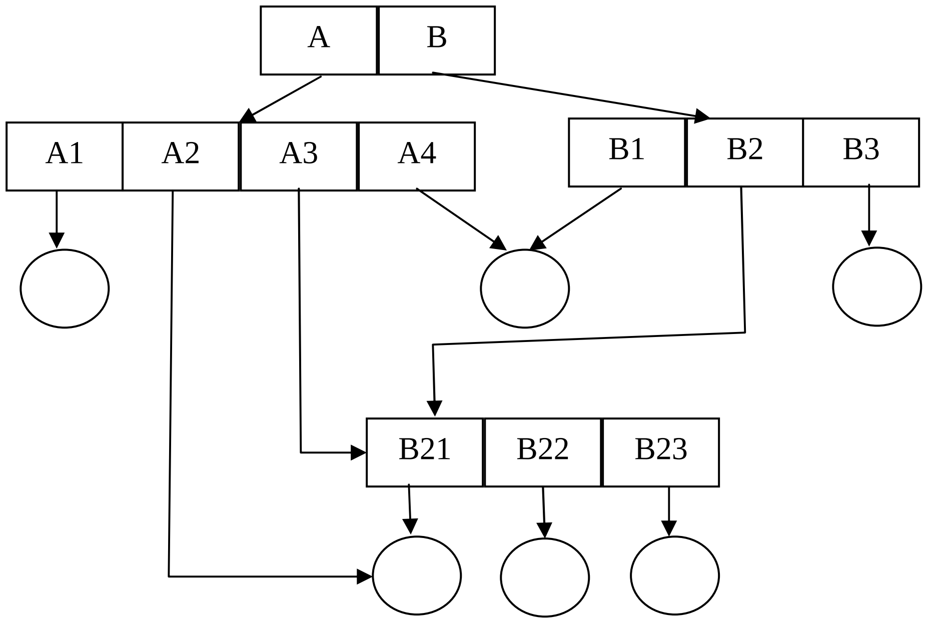


Figure 5 Acyclic-Graph Directories

The important characteristic of sharing is that if a user is making a change in a shared file that is to be reflected to other user also. In this way a shared file is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, there is only one actual file, so any changes made by the person would be immediately visible to the other.

This form of sharing is particularly important for shared subdirectories; a new file created by one person will automatically appear in all the shared subdirectories. File sharing is facilitated by acyclic graph structure. The tree structure doesn’t permit the sharing of files.
In a situation where several people are working as a team, all the files to be shared may be put together into one directory. The user file directories of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into several different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.
Shared files and subdirectories can be implemented in several ways. A common used in UNIX systems, is to create a new directory entry called a link. A link is a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name. When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We resolve the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry (or by their having a special type on systems that support types), and are effectively named indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.
The other approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency if the file is modified. An acyclic-graph directory structure is more flexible than is a simple tree structure, but is also more complex. Several problems must be considered carefully. Notice that a file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system this problem becomes significant, since we do not want to traverse shared structures more than once.
Another problem involves deletion. When can the space allocated to a shared file be de-allocated and reused? One possibility is to remove the file whenever anyone deletes it, but

this action may leave dangling pointers to the now non-existent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is de-allocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated link is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty. The trouble with this approach is the variable and potentially large size of the file-reference list.

However, we really do not need to keep the entire list -we need to keep only a count of the number of references. So a reference count is maintained with shared file, whenever a reference is made to it, it is incremented by one. On deleting a link, the reference count is decremented by one, when it becomes zero the file can be deleted. The UNIX operating system uses this approach for non-symbolic links, or hard links, keeping a reference count in the file information block or inode. By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories links. For example, in MS-DOS, the directory structure is a tree structure, rather than an acyclic graph, thereby avoiding the problems associated with file deletion in an acyclic-graph directory structure.

3.2.1.2.5 General Graph Directory

One serious problem with using an acyclic graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to existing tree structure preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse file in the graph and to determine when there are no more references to a file. We want to avoid file is traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding that file, we want to avoid searching that subdirectory again; the second search would be a waste of time.

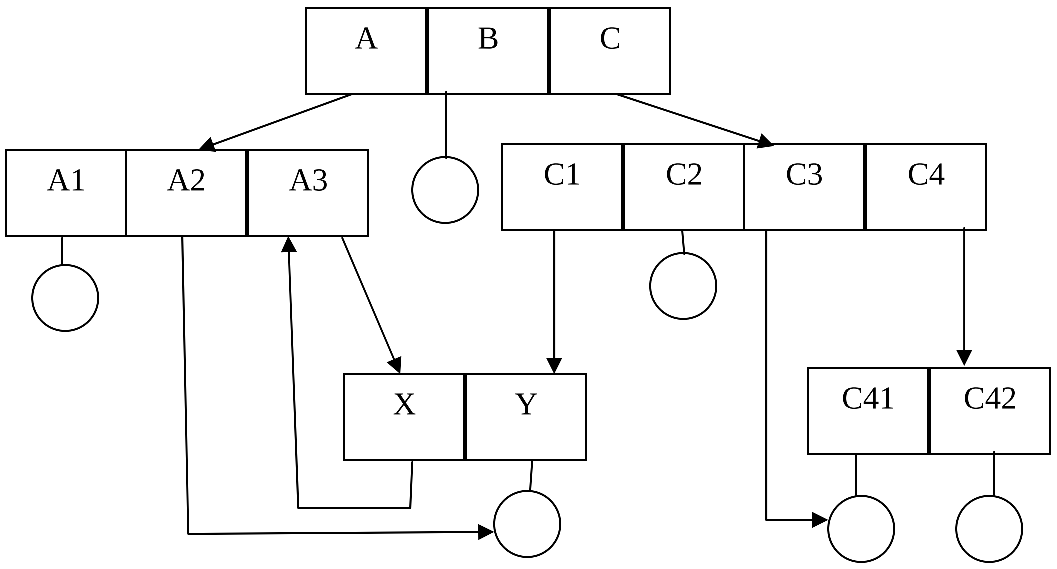


Figure 6 General Graph Directory

To improve the performance of the system we should avoid searching any component twice in the systems where cycles are permitted. If cycles are not identified by the algorithm then it can be trapped in an infinite loop. One solution is to arbitrarily limit the number of directories, which will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a value zero in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, it is also possible, when cycles exist, that the reference count may be nonzero, even when it is no longer possible to refer to a directory or file. This anomaly is due to the self-referencing (a cycle) in the directory structure. In this case, it is generally necessary to use a garbage collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Garbage collection for a disk based file system, however, is extremely time-consuming and is thus seldom attempted. Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles, as new links are added to the structure. There are algorithms to detect cycles in graphs. However, they are computationally expensive, especially when the graph is on disk storage. Generally, tree directory structures are more common than are acyclic-graph structures.

3.2.1.3 Directory Operations

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, then it becomes apparent that the directory itself can be organized in many ways. The different operations that are to be carried out on directories are:

(a) To insert entries.

(b) To delete entries.

(c) To search for a named entry.

(d) To list all the entries in the directory.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

➢ **Search for a directory:** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.

- **Create a directory:** New files need to be created and added to the directory.
- **Delete a directory:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a directory:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** It is useful to be able to access every directory and every file within a directory structure. For reliability it is a good idea to save the contents and structure of the entire file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.
- **Copying a directory:** A directory may be copied from one location to another.
- **Moving a directory:** A directory may be moved from one location to a new location with all its contents.

3.2.2 FILE PROTECTION & SECURITY

The security of the information is a major issue in file system. The files are to be protected from the physical damage as well as improper access. One way of ensuring the security is through backup. By maintaining the duplicate copy of the files, the reliability is improved. In many systems this is done automatically without human intervention. The backup of the files is done at regular interval automatically. So if a copy of the file is accidentally destroyed, we have its backup copy.

There are a number of factors causing the damage to the file system such as:

(a) Hardware problems.
(b) Power failure
(c) Head crashes
(d) Dirt

(e) Temperature

(f) Bugs in the software

These things can result into the loss of contents of files. Protection can be provided in many ways. For a small single-user system, we 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.

3.2.2.1 Types of Access

The need for protecting files is a direct result of the ability to access files. On systems that do not permit access to the files of other users, protection is not needed. Thus, one extreme would be to provide complete protection by prohibiting access. The other extreme is to provide free access with no protection. Both of these approaches are too extreme for general use. What is needed is the controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read** - Read information contained in the file.
- **Write** - Write new information into a file at any point or overwrite existing information in a file.
- **Execute** - Load the contents of a file into main memory and create a process to execute it.
- **Append** - Write new information at the end of the file.
- **Delete** - Delete the file and release its storage space for use in other files.
- **List** – Read the names contained in a directory.
- **Change access** - Change some user’s access rights for some controlled operation.

Other operations, such as renaming, copying, or editing the file, may also be controlled. For many systems, however, these higher-level functions (such as copying) may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In his case, a user with read access can also cause the file to be copied, printed, and so on.

Many different protection mechanisms have been proposed. Each scheme has its advantages and disadvantages and must be selected as appropriate for intended application. A small computer system that is used by only a few members of a research group may not need the same types of protection as will a large corporate computer that is used for research, finance, and personnel iterations.

3.2.2.2 Protection structures

An access privilege is a right to make a specific form of access to a file. An access descriptor describes access privileges for a file. The common accesses privileges read, write, and execute are generally represented by r, w, and x descriptors. A user holds access privileges to one or more files and a file is accessible to one or more users. Access control information for a file is a collection of access descriptors for access privileges held by various users. Access control information can be organized in various forms such as Access Control Matrix, access Control Lists etc. which are discussed in the following section:

3.2.2.2.1 Access Control Matrix

Access control matrix (ACM) consists of rows and columns as shown in the following figure. Each row describes the access privileges held by a user. Each column describes the access control information for a file. Thus ACM $(u_i, f_j) = a_{ij}$ implies that user u_i can access file f_j in accordance with access privileges a_{ij}.

Files→ Users ↓	f_1	f_2	f_3
u_1	{r}	(r, w}	{r, w, x}
u_2		{r}	{r, x}
u_3	{w}	{r, w, x}	{r}

Figure 7 Access Control Matrix

The important advantages of ACM are:

(a) Simplicity and efficiency of access.

(b) All information is stored in one structure.

But its main drawback is its size and sparseness. The size can be reduced by assigning access privileges to group of users rather than the individual users resulting in the reduction of number of rows. The solution of sparseness is the use of lists instead of matrix as discussed following.

3.2.2.2.2. Access control Lists and Groups

The most common approach to the protection problem is to make access dependent on the stems identity of the user. Various users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an access control list (ACL), specifying the user name and the types of access allowed for each user. Each element of the access control list is an access control pair (<user name>, <list of access privileges>).

When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length. It depends on the number of users and the number of access privileges defined in the system. Most file systems uses three kinds of access privileges: (a) Read - file can be read, (b) write – file can be modified and new data can be added, and (c) execute – permits the execution of the program. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

(a) Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.

(b) The directory entry that previously was of fixed size needs now to be of variable size, resulting in space management being more complicated.

To reduce the size of protection information, users can be classified in some convenient manner and an access control pair can be specified for each class of user rather than for individual users. Now an access control list has only as many pairs as the number of user classes. To condense the length of the access list, many systems recognize three classifications of users in connection with each file (e.g. in UNIX):

1. Owner - The user who created the file is the owner
2. Group - A set of users who are sharing the file and need similar access is a group or workgroup.
3. Universe - All other users in the system constitute the universe.

Note that, for this scheme to work properly, group membership must be controlled tightly. This control can be accomplished in a number of different ways. For example, in the UNIX system, groups can be created and modified by only the manager of the facility (or by any super-user). Thus, this control is achieved through human interaction. In the VMS system, with each file, an access list (also known as an access control list) may be associated, listing those users who can access the file. The owner of the file can create and modify this access lists are discussed above.

With this more limited protection classification, only three fields are needed to define protection. Each field is often a collection of bits, each of which either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each: rwx, where r controls read access, w controls write access, and x controls execution. A separate field is kept for the file owner, for the owner's group and for all other users. In this scheme, 9 bits per file are needed to record protection information.

3.2.2.2.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password.

There are several disadvantages to this scheme.

(a) First, if we associate a separate password with each file, then the number of passwords that a user needs to remember may become large, making the scheme impractical.

(b) If only one password is used for all the files, then, once it is discovered, all files are accessible.

Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBM VM/CMS operating system allows three passwords for a minidisk: one each for read, write, and multi write access. Second, commonly, only one password is associated with each file. Thus, protection is on an all-or-nothing basis. To provide protection on a more detailed level, we must use multiple passwords.

Limited file protection is also currently available on single user systems, such as MS-DOS and Macintosh operating system. These operating systems, when originally designed, essentially ignored dealing with the protection problem. However, since these systems are being placed on networks where file sharing and communication is necessary, protection mechanisms have to be retrofitted into the operating system. Note that it is almost always easier to design a feature into an new operating system than it is to add a feature to an existing one. Such updates are usually less effective and are not seamless.

We note that, in a multilevel directory structure, we need not only to protect individual files, but also to protect collections of files contained in a subdirectory, that is, we need to provide a mechanism for directory protection.

The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file may be significant in itself. Thus, listing the contents of a directory must be a protected operation. Therefore, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file, depending on the path name used.

3.3 Keywords

Directory: A directory may be defined as an object that contains the names of the file system objects.

Access Control Matrix: ACM is a matrix in which each row describes the access privileges held by a user and each column, access control information for a file.

Access Control List: It is a structure to implement identity-dependent access to each file and directory where each element of the ACL is an access control pair (<user name>, <list of access privileges>).

3.4 SUMMARY

A file system helps the user in organizing the files through the use of directories that contains information about a group of files. A number of directory structures are used such as Single-level Directory Two-level Directory, Tree-structured Directories, Acyclic-Graph Directories, and General Graph Directory. Each approach has its merits and demerits. A number of operations are carried out on directories such as insertion, deletion, search, rename, traversal etc. So file system should facilitate these operations. Another important issue in file system is the protection of the information from physical damage and unauthorized access. To provide the access privileges to the files to different users two common mechanisms Access Control Matrix and Access Control Lists were discussed. ACM are characterized by their simplicity and efficiency but suffers from large size and sparseness. The problem of size is tackled by using the groups. If there are a number of blank entries in the ACM, then ACL can be a preferred solution.

3.5 SUGGESTED READINGS / REFERENCE MATERIAL

6. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
7. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
8. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
9. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
10. Operating Systems, Har ris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

3.6 SELF-ASSESSMENT QUESTIONS (SAQ)

1. Define field, record, file, file sharing, and file protection.
2. What are the limitations of acyclic directory structure?
3. Which file operations are applicable to directories? Which are not?
4. How is a directory different from a file?
5. What are the different logical structures of the directory? Discuss their merits and demerits?
6. Discuss the advantages and disadvantages of Access Control Lists (ACL) and Access Control Matrix (ACM).
7. Discuss the advantages and disadvantages of two-level directory structure over single-level directory structure.

Lesson number: 4	***Writer: Dr. Rakesh Kumar***
CPU Scheduling	***Vetter: Prof. Dharminder Kr.***

3.0 OBJECTIVE

The objective of this lesson is to make the students familiar with the various issues of CPU scheduling. After studying this lesson, they will be familiar with:

1. Process states & transitions.
2. Different types of scheduler
3. Scheduling criteria
4. Scheduling algorithms

3.1 INTRODUCTION

In nearly every computer, the resource that is most often requested is the CPU or processor. Many computers have only one processor, so this processor must be shared via time-multiplexing among all the programs that need to execute on the computer. Here we need to make an important distinction between a program & an executing program.

"One of the most fundamental concepts of modern operating systems is the distinction between a program & the activity of executing a program. The former is merely a static set of directions; the latter is a dynamic activity whose properties change as time progresses. This activity is knows as a process. A process encompasses the current status of the activity, called the process state. This state includes the current position in the program being executed (the value of the program counter) as well as the values in the other CPU registers & the associated memory cells. Roughly speaking, the process state is a snapshot of the machine at that time. At different times during the execution of a program (at different times in a process) different snapshots (different process states) will be observed."

The operating system is responsible for managing all the processes that are running on a computer & allocated each process a certain amount of time to use the processor. In addition, the operating system also allocates various other resources that processes will need such as computer memory or disks. To keep track of the state of all the processes, the operating system maintains a table known as the process table. Inside this table, every process is listed along with the resources the processes are using & the current state of the process. Processes can be in one of three states: running, ready, or waiting (blocked). The running state means that the process has all the resources it need for execution & it has been given permission by the operating system to use the processor. Only one process can be in the running state at any given time. The remaining processes are either in a waiting state (i.e., waiting for some external event to occur such as user input or a disk access) or a ready state (i.e., waiting for permission to use the processor). In a real operating system, the waiting & ready states are implemented as queues, which hold the processes in these states.

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned & to which processes is called processor scheduling or CPU scheduling.

When more than one process is runable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, & algorithm it uses is called the scheduling algorithm. In operating system literature, the term “scheduling” refers to a set of policies & mechanisms built into the operating system that govern the order in which the work to be done by a computer system is completed. A scheduler is an OS module that selects the next job to be admitted into the system & the next process to run. The primary objective of scheduling is to optimize system performance in accordance with the criteria deemed most important by the system designers.

3.2 PRESENTATION OF CONTENTS

3.2.1 Definition of Process

3.2.2 Process States & Transitions

3.2.3 Types of schedulers

- 3.2.3.1 The long-term scheduler
- 3.2.3.2 The medium-term scheduler
- 3.2.3.3 The short-term scheduler

3.2.4 Scheduling & Performance Criteria

- 3.2.4.1 User-oriented Scheduling Criteria
- 3.2.4.2 System-oriented Scheduling Criteria

3.2.5 Scheduler Design

3.2.6 Scheduling Algorithms

- 3.2.6.1 First-Come, First-Served (FCFS) Scheduling
- 3.2.6.2 Shortest Job First (SJF)
- 3.2.6.3 Shortest Remaining Time Next (SRTN) Scheduling
- 3.2.6.4 Round Robin
- 3.2.6.5 Priority-Based Preemptive Scheduling (Event-Driven, ED)
- 3.2.6.6 Multiple-Level Queues (MLQ) Scheduling
- 3.2.6.7 Multiple-Level Queues with Feedback Scheduling

3.2.1 Definition of Process

The notion of process is central to the understanding of operating systems. There are quite a few definitions presented in the literature, but no "perfect" definition has yet appeared.

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

- A program in Execution.
- An asynchronous activity.
- The 'animated sprit' of a procedure in execution.
- The entity to which processors are assigned.
- The 'dispatchable' unit.

and many more definitions have been given. As we can see from above that there is no universally agreed upon definition, but the definition "Program in Execution" seem to be most frequently used. Now that we agreed upon the definition of process, the question is what is the relation between process & program. Process is not the same as program. In the following discussion we point out some of the difference between process & program. As we have mentioned earlier 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 consider being a 'passive' entity. As we all know that a program is an algorithm expressed in some suitable notation, (e.g., programming language). Being a passive, a program is only a part of process. Process, on the other hand, includes:

- Current value of Program Counter (PC)
- Contents of the processors registers
- Value of the variables
- The process stack (SP) which typically contains temporary data such as subroutine parameter, return address, & temporary variables.
- A data section that contains global variables.

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, & variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back & forth among processes. The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

- Code for the program.
- Program's static data.
- Program's dynamic data.
- Program's procedure call stack.
- Contents of general purpose register.
- Contents of program counter (PC)
- Contents of program status word (PSW).
- Operating Systems resource in use.

A process goes through a series of discrete process states.

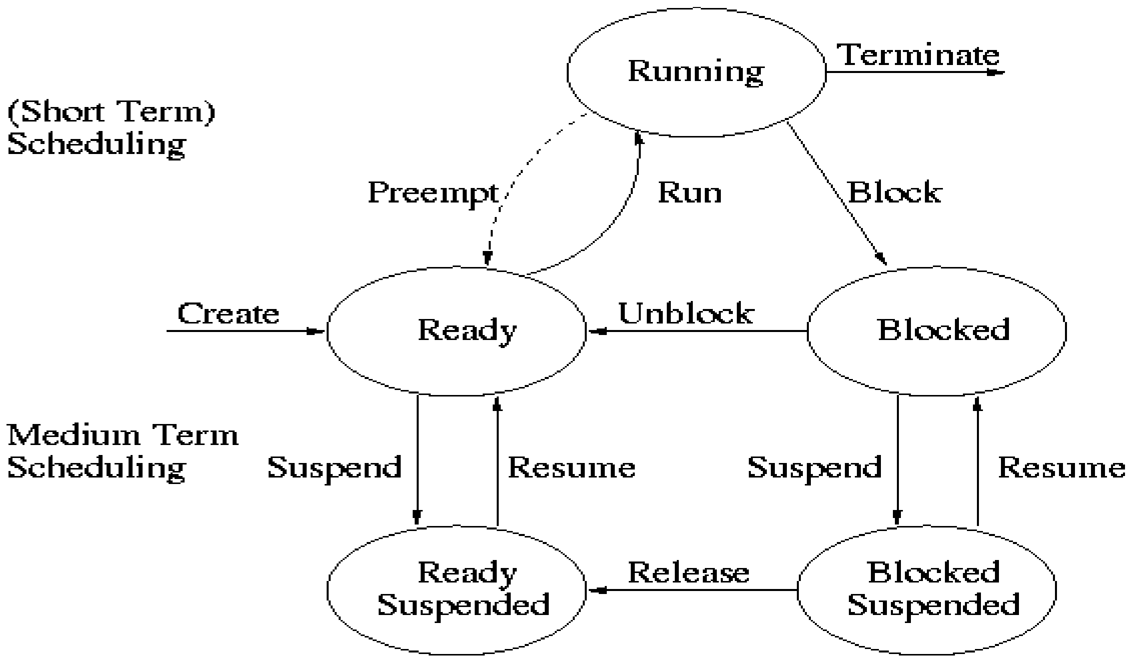
- **New State:** The process being created.
- **Running State:** A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
- **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
- **Ready State:** A process is said to be ready if it use a CPU if one were available. A ready state process is runable but temporarily stopped running to let another process run.
- **Terminated state:** The process has finished execution.

3.2.2 Process States & Transitions

The diagram below contains much information.

Consider a running process P that issues an I/O request

- The process blocks
- At some later point, a disk interrupt occurs & the driver detects that P's request is satisfied.
- P is unblocked, i.e. is moved from blocked to ready
- At some later time the operating system looks for a ready job to run & picks P & P moved to running state.
- A suspended process may be removed from the main memory & placed in the backup memory. Subsequently they may be released & moved to the ready state by the medium term scheduler.



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

3.2.3 TYPES OF SCHEDULERS

The schedulers may be categorized as long term scheduler, medium term scheduler, & short term scheduler as shown in Figure 1 & Figure 2. Figure 1 shows the possible traversal paths of jobs & programs through the components & queues, depicted by rectangles, of a computer system. The primary places of action of the three types of schedulers are marked with down-arrows. As shown in Figure 2, a submitted batch job joins the batch queue while waiting to be processed by the long-term scheduler. Once scheduled for execution, processes spawned by the batch job enter the ready queue to await processor allocation by the short-term scheduler. After becoming suspended, the running process may be removed from memory & swapped out to secondary storage. Such processes are subsequently admitted to main memory by the medium-term scheduler in order to be considered for execution by the short-term scheduler.

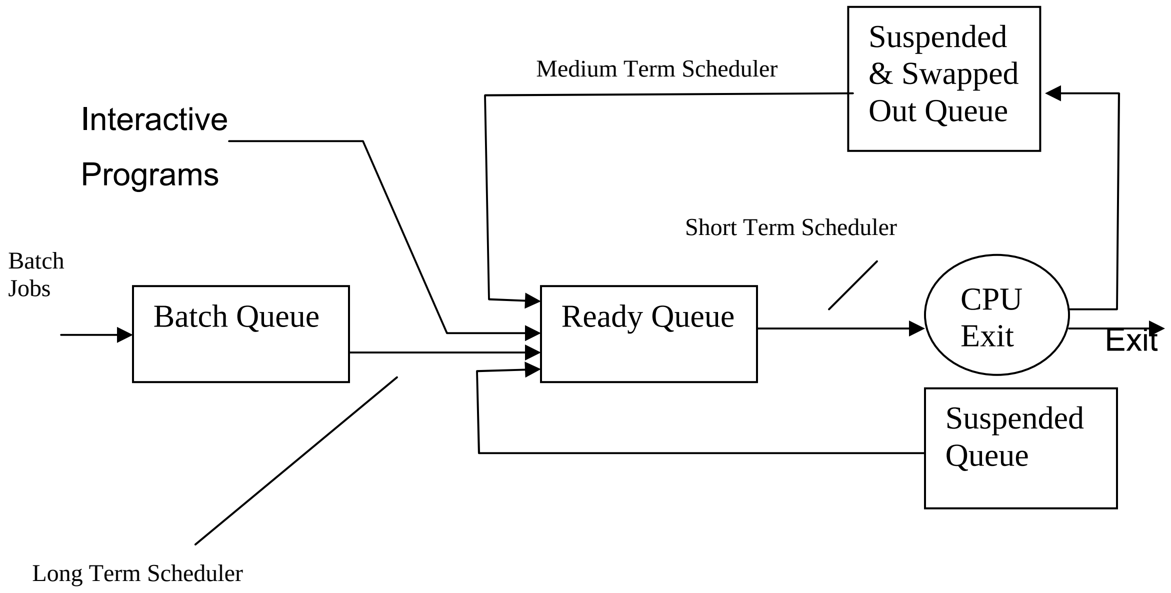


Figure 1- Process Schedulers

3.2.3.1 The long-term scheduler

The long-term scheduler decides when to start jobs, i.e., do not necessarily start them when submitted. CTSS (an early time sharing system at MIT) did this to insure decent interactive response time. The long-term scheduler, when present, works with the batch queue & selects the next batch job to be executed. Batch is usually reserved for resource-intensive (processor time, memory, special I/O devices), low-priority programs that may be used as fillers to keep the system resources busy during periods of low activity of interactive jobs. As pointed out earlier, batch jobs contain all necessary data & commands for their execution. Batch jobs usually also contain programmer-assigned estimates of their resource needs, such as memory size, expected execution time, & device requirements. Knowledge about the anticipated job behavior facilitates the work of the long-term scheduler.

The primary objective of the long-term scheduler is to provide a balanced mix of jobs, such as processor-bound & I/O-bound, to the short-term scheduler. In a way, the long-term scheduler acts as a first-level throttle in keeping resource utilization at the desired level. For example, when the processor utilization is low, the scheduler may admit more jobs to increase the number of processes in a ready queue, & with it the probability of having some useful work awaiting processor allocation. Conversely, when the utilization factor becomes high as reflected in the response time, the long-term scheduler may opt to reduce the rate of batch-job admission accordingly. In addition, the long-term scheduler is usually invoked whenever a completed job departs the system. The frequency of invocation of the long-term scheduler is thus both system-and workload-dependent; but it is generally much lower than for the other two types of schedulers. As a result of the relatively infrequent execution & the availability of an estimate of its workload's characteristics, the long-term scheduler may incorporate rather complex & computationally intensive algorithms for admitting jobs into the system. In terms of the process state-transition diagram, the long-term scheduler is basically in charge of the dormant-to-ready transitions. Ready processes are placed in the ready queue (ready list, in our earlier discussion) for consideration by the short-term scheduler.

3.2.3.2 The medium-term scheduler

The medium term scheduler suspend (swap out) some process if memory is over-committed. The criteria for choosing a victim may be (a) How long since previously suspended? (b) How much CPU time used recently? (c) How much memory does it use? (d) External priority (pay more, get swapped out less) etc.

A running process may become suspended by making an I/O request or by issuing a system call. Given that suspended processes cannot make any progress towards completion until the related suspending condition is removed, it is sometimes beneficial to remove them from main memory to make room for other processes. In practice, the main-memory capacity may impose a limit on the number of active processes in the system. When a number of those processes become suspended, the remaining supply of ready processes in systems where all suspended processes remain resident in memory may become reduced to a level that impairs functioning of the short-term scheduler by leaving it few or no options for selection. In systems with no support for virtual memory, moving suspended processes to secondary storage may alleviate this problem. Saving the image of a suspended process in secondary storage is called swapping, & the process is said to be swapped out or rolled out.

The medium-term scheduler is in charge of handling the swapped-out processes. It has little to do while a process remains suspended. However, once the suspending condition is removed, the medium-term scheduler attempts to allocate the required amount of main memory, & swap the process in & make it ready. To work properly, the medium-term scheduler must be provided with information about the memory requirements of swapped-out processes.

In terms of the state-transition diagram, the medium-term scheduler controls suspended-to-ready transitions of swapped processes. This scheduler may be invoked when memory space is vacated by a departing process or when the supply of ready processes falls below a specified limit.

Medium-term scheduling is really part of the swapping function of an operating system. The success of the medium-term scheduler is based on the degree of multiprogramming that it can maintain, by keeping as many processes “runnable” as possible. More processes can remain executable if we reduce the resident set size of all processes. The medium-term scheduler makes decisions as to which pages of which processes need stay resident, & which pages must be swapped out to make room for other processes. The sharing of some pages of memory, either explicitly or through the use of shared or dynamic link libraries complicates the task of the medium-term scheduler, which now must maintain reference counts on each page. The responsibilities of the medium-term scheduler may be further complicated in some operating systems, in which some processes may request (demand?) that their pages remain locked in physical memory:

3.2.3.3 The short-term scheduler

The long-term scheduler runs relatively infrequently, when a decision must be made as to the admission of new processes: maybe on average every ten seconds. The medium-term scheduler runs more frequently, deciding which process’s pages to swap to & from the swapping device: typically once a second. The short-term scheduler, often termed the dispatcher, executes most frequently (every few hundredths of a second) making fine-grained decisions as to which process to move to Running next. The short-term scheduler is invoked whenever an event occurs which provides the opportunity, or requires, the interruption of the current process & the new (or continued) execution of another process. Such opportunities include:

- Clock interrupts, provide the opportunity to reschedule every few milliseconds,
- Expected I/O interrupts, when previous I/O requests are finally satisfied,
- Operating system calls, when the running process asks the operating system to perform an activity on its behalf, and

- Unexpected, asynchronous, events, such as unexpected input, user-interrupt, or a fault condition in the running program.

The short-term scheduler allocates the processor among the pool of ready processes resident in memory. Its main objective is to maximize system performance in accordance with the chosen set of criteria. Since it is in charge of ready-to-running state transitions, the short-term scheduler must be invoked for each process switch to select the next process to be run. In practice, the short-term scheduler is invoked whenever an event (internal or external) causes the global state of the system to change. Given that any such change could result in making the running process suspended or in making one or more suspended processes ready, the short-term scheduler should be run to determine whether such significant changes have indeed occurred and, if so, to select the next process to be run. Some of the events occurred and, if so, to select the next process to be run.

Most of the process-management OS services discussed in this lesson requires invocation of the short-term scheduler as part of their processing. For example, creating a process or resuming a suspended one adds another entry to the ready list (queue), & the scheduler is invoked to determine whether the new entry should also become the running process. Suspending a running process, changing priority of the running process, & exiting or aborting a process are also events that may necessitate selection of a new running process, changing priority of the running process, & exiting or aborting a process are also events that may necessitate selection of a new running process. Some operating systems include an OS call that allows system programmers to cause invocation of the short-term scheduler explicitly, such as the DECLARE_SIGNIFICANT_EVENT call in the RSX-11M operating system. Among other things, this service is useful for invoking the scheduler from user-written event-processing routines, such as device (I/O) drivers.

As indicated in Figure 2, interactive programs often enter the ready queue directly after being submitted to the OS, which then creates the corresponding process. Unlike-batch jobs, the influx of interactive programs are not throttled, & they may conceivably saturate the system. The necessary control is usually provided indirectly by deterioration response time, which tempts the users to give up & try again later, or at least to reduce the rate of incoming requests.

Figure 2 illustrates the roles & the interplay among the various types of schedulers in an operating system. It depicts the most general case of all three types being present. For example, a larger operating system might support both batch & interactive programs & rely on swapping to maintain a well-behaved mix of active processes. Smaller or special-purpose operating systems may have only one or two types of schedulers available. Along-term scheduler is normally not found in systems without support for batch, & the medium-term scheduler is needed only when swapping is used by the underlying operating system. When more than one type of scheduler exists in an operating system, proper support for communication & interaction is very important for attaining satisfactory & balanced performance. For example, the long-term & the medium-term schedulers prepare workload for the short-term scheduler. If they do not provide a balanced mixed of compute-bound & I/O-bound processes, the short-term scheduler is not likely to perform well no matter how sophisticated it may be on its own merit.

3.2.4 SCHEDULING & PERFORMANCE CRITERIA

The objectives of a good scheduling policy include

- Fairness.
- Efficiency.
- Low response time (important for interactive jobs).
- Low turnaround time (important for batch jobs).
- High throughput
- Repeatability.
- Fair across projects.
- Degrade gracefully under load.

The success of the short-term scheduler can be characterized by its success against user-oriented criteria under which a single user (selfishly) evaluates their perceived response, or system-oriented criteria where the focus is on efficient global use of resources such as the processor & memory. A common measure of the system-oriented criteria is throughput, the rate at which tasks are completed. On a single-user, interactive operating system, & the user-oriented criteria take precedence: it is unlikely that an individual will exhaust resource consumption, but responsiveness remains all important. On a multi-user, multi-tasking system, the global system-oriented criteria are more important as they attempt to provide fair scheduling for all, subject to priorities & available resources.

3.2.4.1 User-oriented Scheduling Criteria

Response time

In an interactive system this measures the time between submissions of a new process request & the commencement of its execution. Alternatively, it can measure the time between a user issuing a request to interactive input (such as a prompt) & the time to echo the user's input or accept the carriage return.

Turnaround time

This is the time between submission of a new process & its completion. Depending on the mixture of current tasks, two submissions of identical processes will likely have different turnaround times. Turnaround time is the sum of execution & waiting times.

Deadlines

In a genuine real-time operating system, hard deadlines may be requested by processes. These either demands that the process is completed with a guaranteed upper-bound on its turnaround time, or provide a guarantee that the process will receive the processor in a guaranteed maximum time in the event of an interrupt. A real-time long-term scheduler should only accept a new process if it can guarantee required deadlines. In combination, the short-term scheduler must also meet these deadlines.

Predictability

With lower importance, users expect similar tasks to take similar times. Wild variations in response & turnaround times are distracting.

3.2.4.2 System-oriented Scheduling Criteria

Throughput

The short-term scheduler attempts to maximize the number of completed jobs per unit time. While this is constrained by the mixture of jobs, & their execution profiles, the policy affects utilization & thus completion.

Processor utilization

The percentage of time that the processor may be fed with work from Ready. In a single-user, interactive system, processor utilization is very unlikely to exceed a few percent.

Fairness

Subject to priorities, all processes should be treated fairly, & none should suffer processor starvation. This simply implies, in most cases, that all processes are moved to the ends of their respective state queues, & may not “jump the queue”.

Priorities

Conversely, when processes are assigned priorities, the scheduling policy should favor higher priorities.

3.2.5 SCHEDULER DESIGN

Design process of a typical scheduler consists of selecting one or more primary performance criteria & ranking them in relative order of importance. The next step is to design a scheduling strategy that maximizes performance for the specified set of criteria while obeying the design constraints. One should intentionally avoid the word "optimization" because most scheduling algorithms actually implemented do not schedule optimally. They are based on heuristic techniques that yield good or near-optimal performance but rarely achieve absolutely optimal performance. The primary reason for this situation lies in the overhead that would be incurred by computing the optimal strategy at run-time, & by collecting the performance statistics necessary to perform the optimization. Of course, the optimization algorithms remain important, at least as a yardstick in evaluating the heuristics. Schedulers typically attempt to maximize the average performance of a system, relative to a given criterion. However, due consideration must be given to controlling the variance & limiting the worst-case behavior. For example, a user experiencing 10-second response time to simple queries has little consolation in knowing that the system's average response time is under 2 seconds.

One of the problems in selecting a set of performance criteria is that they often conflict with each other. For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time deteriorates. As is the case with most engineering problems, the design of a scheduler usually requires careful balance of all the different requirements & constraints. With the knowledge of the primary intended use of a given system, operating-system designers tend to maximize the criteria most important in a given environment. For example, throughput & component utilization are the primary design objectives in a batch system. Multi-user systems are dominated by concerns regarding the terminal response time, & real-time operating systems are designed for the ability to handle burst of external events responsively.

3.2.7 SCHEDULING ALGORITHMS

The scheduling mechanisms described in this section may, at least in theory, be used by any of the three types of schedulers. As pointed out earlier, some algorithms are better suited to the needs of a particular type of scheduler. Depending on whether a particular scheduling discipline is primarily used by the long-term or by the short-term scheduler, we illustrate its working by using the term job or process for a unit of work, respectively.

The scheduling policies may be categorized as preemptive & non-preemptive. So it is important to distinguish preemptive from non-preemptive scheduling algorithms. Preemption means the operating system moves a process from running to ready without the process requesting it. Without preemption, the system implements “run to completion". Preemption needs a clock interrupt (or equivalent). Preemption is needed to guarantee fairness & it is found in all modern general-purpose operating systems.

Non-pre-emptive: In non-preemptive scheduling, once a process is executing, it will continue to execute until

- It terminates, or

- It makes an I/O request which would block the process, or
- It makes an operating system call.

Pre-emptive: In the preemptive scheduling, the same three conditions as above apply, & in addition the process may be pre-empted by the operating system when

- A new process arrives (perhaps at a higher priority), or
- An interrupt or signal occurs, or
- A (frequent) clock interrupt occurs.

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. Following are some scheduling algorithms we will study: FCFS Scheduling, Round Robin Scheduling, SJF Scheduling, SRTN Scheduling, Priority Scheduling, Multilevel Queue Scheduling, & Multilevel Feedback Queue Scheduling.

3.2.6.1 First-Come, First-Served (FCFS) Scheduling

The simplest selection function is the First-Come-First-Served (FCFS) scheduling policy. In it

1. The operating system kernel maintains all Ready processes in a single queue,
2. The process at the head of the queue is always selected to execute next,
3. The Running process runs to completion, unless it requests blocking I/O,
4. If the Running process blocks, it is placed at the end of the Ready queue.

Clearly, once a process commences execution, it will run as fast as possible (having 100% of the CPU, & being non-pre-emptive), but there are some obvious problems. By failing to take into consideration the state of the system & the resource requirements of the individual scheduling entities, FCFS scheduling may result in poor performance. As a consequence of no preemption, component utilization & the system throughput rate may be quite low.

Processes of short duration suffer when “stuck” behind very long-running processes. Since there is no discrimination on the basis of the required service, short jobs may suffer considerable turnaround delays & waiting times when one or more long jobs are in the system. For example, consider a system with two jobs, J1 & J2, with total execution times of 20 & 2 time units, respectively. If they arrive shortly one after the other in the order J1-J2, the turnaround times are 20 & 22 time units, respectively (J2 must wait for J1 to complete), thus yielding an average of 21 time units. The corresponding waiting times are 0 & 20 unit, yielding an average of 10 time units. However, when the same two jobs arrive in the opposite order, J2-J1, the average turnaround time drops to 11, & the average waiting time is only 1 time unit.

Compute-bound processes are favored over I/O-bound processes.

We can measure the effect of FCFS by examining:

- The average turnaround time of each task (the sum of its waiting & running times), or
- The normalized turnaround time (the ratio of running to waiting times).

3.2.6.2 Shortest Job First (SJF)

In this scheduling policy, the jobs are sorted on the basis of total execution time needed & then it run the shortest job first. It is a non-preemptive scheduling policy. Now First consider a static situation where all jobs are available in the beginning, & we know how long each one takes to run, & we implement “run-to-completion'' (i.e., we don't even switch to another process on I/O). In this situation, SJF has the shortest average waiting time. Assume you have a schedule with a long job right before a short job. Now if we swap the two jobs, this decreases the wait for the short by the length of the long job & increases the wait of the long job by the length of the short job. & this in turn decreases the total waiting time for these two. Hence decreases the total waiting for all jobs & hence decreases the average waiting time as well. So in this policy whenever a long job is right before a short job, we swap them & decrease the average waiting time. Thus the lowest average waiting time occurs when there are no short jobs rights before long jobs. This is an example of priority scheduling. This scheduling policy can starve processes that require a long burst.

3.2.6.3 Shortest Remaining Time Next (SRTN) Scheduling

Shortest remaining time next is a scheduling discipline in which the next scheduling entity, a job or a process, is selected on the basis of the shortest remaining execution time. SRTN scheduling may be implemented in either the non-preemptive or the preemptive variety. The non-preemptive version of SRTN is called shortest job first (SJF). In either case, whenever the SRTN scheduler is invoked, it searches the corresponding queue (batch or ready) to find the job or the process with the shortest remaining execution time. The difference between the two cases lies in the conditions that lead to invocation of the scheduler and, consequently, the frequency of its execution. Without preemption, the SRTN scheduler is invoked whenever a job is completed or the running process surrenders control to the OS. In the preemptive version, whenever an event occurs that makes a new process ready, the scheduler is invoked to compare the remaining processor execution time of the running process with the time needed to complete the next processor burst of the newcomer. Depending on the outcome, the running process may continue, or it may be preempted & replaced by the shortest-remaining-time process. If preempted, the running process joins the ready queue.

SRTN is a provably optimal scheduling discipline in terms of minimizing the average waiting time of a given workload. SRTN scheduling is done in a consistent & predictable manner, with a bias towards short jobs. With the addition of preemption, an SRTN scheduler can accommodate short jobs that arrive after commencement of a long job. Preferred treatment of short jobs in SRTN tends to result in increased waiting times of long jobs in comparison with FCFS scheduling, but this is usually acceptable.

The SRTN discipline schedules optimally assuming that the exact future execution times of jobs or processes are known at the time of scheduling. In the case of short-term scheduling & preemption's, even more detailed knowledge of the duration of each individual processor burst is required. Dependence on future knowledge tends to limit the effectiveness of SRTN implementations in practice, because future process behavior is unknown in general & difficult to estimate reliably, except for some very specialized deterministic cases.

Predictions of process execution requirements are usually based on observed past behavior, perhaps coupled with some other knowledge of the nature of the process & its long-term statistical properties, if available. A relatively simple predictor, called the exponential smoothing predictor, has the following form:

$$P_n = \alpha 0_n\text{-}1 + (1 - \alpha)P\text{-}1$$

where 0_n is the observed length of the (n-1)th execution interval, P_n-1 is the predictor for the same interval, & α is a number between 0 & 1. The parameter α controls the relative weight assigned to the past observations & predictions. For the extreme case of $\alpha = 1$, the past predictor is ignored, & the new prediction equals the last observation. For $\alpha = 0$, the last observation is ignored. In general, expansion of the recursive relationship yields

$$P_n = \alpha \sum_{I=0}^{n-1} (1 - \alpha)^i 0_{n-i-1}$$

Thus the predictor includes the entire process history, with its more recent history weighted more.

Many operating systems measure & record elapsed execution time of a process in its PCB. This information is used for scheduling & accounting purposes. Implementation of SRTN scheduling obviously requires rather precise measurement & imposes the overhead of predictor calculation at run time. Moreover, some additional feedback mechanism is usually necessary for corrections when the predictor is grossly incorrect.

SRTN scheduling has important theoretical implications, & it can serve as a yardstick for assessing performance of other, realizable scheduling disciplines in terms of their deviation from the optimum. Its practical application depends on the accuracy of prediction of the job & process behavior, with increased accuracy calling for more sophisticated methods & thus resulting in greater overhead. The preemptive variety of SRTN incurs the additional overhead of frequent process switching & scheduler invocation to examine each & every process transition into the ready state. This work is wasted when the new ready process has a longer remaining execution time than the running process.

3.2.6.4 Round Robin

In interactive environments, such as time-sharing systems, the primary requirement is to provide reasonably good response time and, in general, to share system resources equitably among all users. Obviously, only preemptive disciplines may be considered in such environments, & one of the most popular is time slicing, also known as round robin (RR).

It is a preemptive scheduling policy. This scheduling policy gives each process a slice of time (i.e., one quantum) before being preempted. As each process becomes ready, it joins the ready queue. A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is preempted, & the oldest process in the ready queue is selected to run next. The time interval between each interrupt may vary.

It is one of the most common & most important scheduler. This is not the simplest scheduler, but it is the simplest preemptive scheduler. It works as follows:

- ➢ The processes that are ready to run (i.e. not blocked) are kept in a FIFO queue, called the "Ready" queue.
- ➢ There is a fixed time quantum (50 msec is a typical number) which is the maximum length that any process runs at a time.
- ➢ The currently active process P runs until one of two things happens:
  - P blocks (e.g. waiting for input). In that case, P is taken off the ready queue; it is in the "blocked" state.
  - P exhausts its time quantum. In this case, P is pre-empted, even though it is still able to run. It is put at the end of the ready queue.

    In either case, the process at the head of the ready queue is now made the active process.
- ➢ When a process unblocks (e.g. the input it's waiting for is complete) it is put at the end of the ready queue.

Suppose the time quantum is 50 msec, process P is executing, & it blocks after 20 msec. When it unblocks, & gets through the ready queue, it gets the standard 50 msec again; it doesn't somehow "save" the 30 msec that it missed last time.

It is an important preemptive scheduling policy. It is essentially the preemptive version of FCFS. The key parameter here is the quantum size q. When a process is put into the running state a timer is set to q. If the timer goes off & the process is still running, the OS preempts the process. This process is moved to the ready state where it is placed at the rear of the ready queue. The process at the front of the ready list is removed from the ready list & run (i.e., moves to state running). When a process is created, it is placed at the rear of the ready list. As q gets large, RR approaches FCFS. As q gets small, RR approaches PS (Processor Sharing).

What value of q should we choose? Actually it is a tradeoff (1) Small q makes system more responsive, (2) Large q makes system more efficient since less process switching.

Round robin scheduling achieves equitable sharing of system resources. Short processes may be executed within a single time quantum & thus exhibit good response times. Long processes may require several quanta & thus be forced to cycle through the ready queue a few times before completion. With RR scheduling, response time of long processes is directly proportional to their resource requirements. For long processes that consist of a number of interactive sequences with the user, primarily the response time between the two consecutive interactions matters. If the computational requirements between two such sequences may be completed within a single time slice, the user should experience good response time. RR tends to subject long processes without interactive sequences to relatively long turnaround & waiting times. Such processes, however, may best be run in the batch mode, & it might even be desirable to discourage users from submitting them to the interactive scheduler.

Implementation of round robin scheduling requires support of an interval timer-preferably a dedicated one, as opposed to sharing the system time base. The timer is usually set to interrupt the operating system whenever a time slice expires & thus force the scheduler to be invoked. The scheduler itself simply stores the context of the running process, moves it to the end of the ready queue, & dispatches the process at the head of the ready queue. The scheduler is also invoked to dispatch a new process whenever the running process surrenders control to the operating system before expiration of its time quantum, say, by requesting I/O. The interval timer is usually reset at that point, in order to provide the full time slot to the new running process. The frequent setting & resetting of a dedicated interval timer makes hardware support desirable in systems that use time slicing.

Round robin scheduling is often regarded as a "fair" scheduling discipline. It is also one of the best-known scheduling disciplines for achieving good & relatively evenly distributed terminal response time. The performance of round robin scheduling is very sensitive to the choice of the time slice. For this reason, duration of the time slice is often made user-tunable by means of the system generation process.

The relationship between the time slice & performance is markedly nonlinear. Reduction of the time slice should not be carried too far in anticipation of better response time. Too short a time slice may result in significant overhead due to the frequent timer interrupts & process switches. On the other hand, too long a time slice reduces the preemption overhead but increases response time.

Too short a time slice results in excessive overhead, & too long a time slice degenerates from round-robin to FCFS scheduling, as processes surrender control to the OS rather than being preempted by the interval timer. The "optimal" value of the time slice lies somewhere in between, but it is both system-dependent & workload-dependent. For example, the best value of time slice for our example may not turn out to be so good when other processes with different behavior are introduced in the system, that is, when characteristics of the workload change. This, unfortunately, is commonly the case with time-sharing systems where different types of programs may be submitted at different times.

In summary, round robin is primarily used in time-sharing & multi-user systems where terminal response time is important. Round robin scheduling generally discriminates against long non-interactive jobs & depends on the judicious choice of time slice for adequate performance. Duration of a time slice is a tunable system parameter that may be changed during system generation.

Variants of Round Robin

State dependent RR

It is same as RR but q is varied dynamically depending on the state of the system. It favors processes holding important resources. For example, non-swappable memory.

External priorities

In it a user can pay more & get bigger q. That is one process can be given a higher priority than another. But this is not an absolute priority, i.e., the lower priority (i.e., less important) process does get to run, but not as much as the high priority process.

3.2.6.5 Priority-Based Preemptive Scheduling (Event-Driven, ED)

In it each job is assigned a priority (externally, perhaps by charging more for higher priority) & the highest priority ready job is run. In this policy, If many processes have the highest priority, it uses RR among them. In principle, each process in the system is assigned a priority level, & the scheduler always chooses the highest-priority ready process. Priorities may be static or dynamic. In either case, the user or the system assigns their initial values at the process-creating time. The level of priority may be determined as an aggregate figure on the basis of an initial value, characteristic, resource requirements, & run-time behavior of the process. In this sense, many scheduling disciplines may be regarded as being priority-driven, where the priority of a process represents its likelihood of being scheduled next. Priority-based scheduling may be preemptive or non-preemptive.

A common problem with priority-based scheduling is the possibility that low-priority processes may be effectively locked out by the higher priority ones. In general, completion of a process within finite time of its creation cannot be guaranteed with this scheduling policy. In systems where such uncertainty cannot be tolerated, the usually remedy is provided by the aging priority, in which the priority of each process is gradually increased after the process spends a certain amount of time in the system. Eventually, the older processes attain high priority & are ensured of completion in finite time.

By means of assigning priorities to processes, system programmers can influence the order in which an ED scheduler services coincident external events. However, the high-priority ones may starve low-priority processes. Since it gives little consideration to resource requirements of processes, event-driven scheduling cannot be expected to excel in general-purpose systems, such as university computing centers, where a large number of user processes are run at the same (default) level of priority.

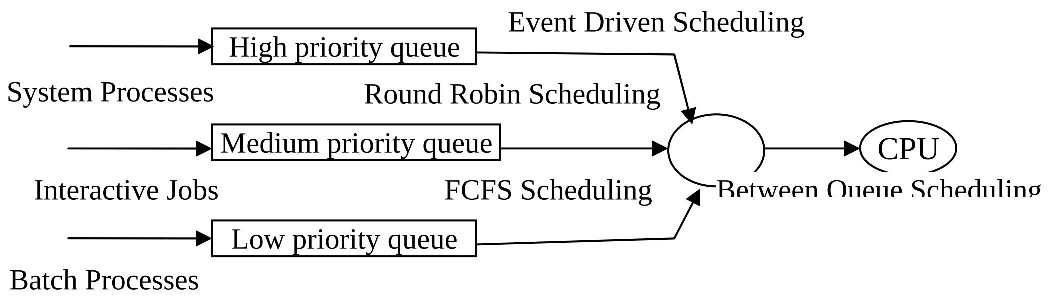
Another variant of priority-based scheduling is used in the so-called hard real-time systems, where each process must be guaranteed execution before expiration of its deadline. In such systems, time-critical processes are assumed to be assigned execution deadlines. The system workload consists of a combination of periodic processes, executed cyclically with a known period, & of periodic processes, executed cyclically with a known period, & of a periodic processes whose arrival times are generally not predictable. An optimal scheduling discipline in such environments is the earliest-deadline scheduler, which schedules for execution the ready process with the earliest deadline. Another form of scheduler, called the least laxity scheduler or the least slack scheduler, has also been shown to be optimal in single-processor systems. This scheduler selects the ready process with the least difference between its deadline & computation time. Interestingly, neither of these schedulers is optimal in multiprocessor environments.

Priority aging

It is a solution to the problem of starvation. As a job is waiting, raise its priority so eventually it will have the maximum priority. This prevents starvation. It is preemptive policy. If there are many processes with the maximum priority, it uses FCFS among those with max priority (risks starvation if a job doesn't terminate) or can use RR.

3.2.6.6 Multiple-Level Queues (MLQ) Scheduling

The scheduling policies discussed so far are more or less suited to particular applications, with potentially poor performance when applied inappropriately. What should one use in a mixed system, with some time-critical events, a multitude of interactive users, & some very long non-interactive jobs? One approach is to combine several scheduling disciplines. A mix of scheduling disciplines may best service a mixed environment, each charged with what it does best. For example, operating-system processes & device interrupts may be subjected to event-driven scheduling, interactive programs to round robin scheduling, & batch jobs to FCFS or STRN.



Multilevel Queue Scheduling

One way to implement complex scheduling is to classify the workload according to its characteristics, & to maintain separate process queues serviced by different schedulers. This approach is often called multiple-level queues (MLQ) scheduling. A division of the workload might be into system processes, interactive programs, & batch jobs. This would result in three ready queues, as depicted in above Figure. A process may be assigned to a specific queue on the basis of its attributes, which may be user-or system-supplied. Each queue may then be serviced by the scheduling discipline best suited to the type of workload that it contains. Given a single server, some discipline must also be devised for scheduling between queues. Typical approaches are to use absolute priority or time slicing with some bias reflecting relative priority of the processes within specific queues. In the absolute priority case, the processes from the highest-priority queue (e.g. system processes) are serviced until that queue becomes empty. The scheduling discipline may be event-driven, although FCFS should not be ruled out given its low overhead & the similar characteristics of processes in that queue. When the highest-priority queue becomes empty, the next queue may be serviced using its own scheduling discipline (e.g., RR for interactive processes). Finally, when both higher-priority queues become empty, a batch-spawned process may be selected. A lower-priority process may, of course, be preempted by a higher-priority arrival in one of the upper-level queues. This discipline maintains responsiveness to external events & interrupts at the expense of frequent preemption's. An alternative approach is to assign a certain percentage of the processor time to each queue, commensurate with its priority.

Multiple queues scheduling is a very general discipline that combines the advantages of the "pure" mechanisms discussed earlier. MLQ scheduling may also impose the combined overhead of its constituent scheduling disciplines. However, assigning classes of processes that a particular discipline handles poorly by itself to a more appropriate queue may offset the worst-case behavior of each individual discipline. Potential advantages of MLQ were recognized early on by the O/S designers who have employed it in the so-called fore-ground/background (F/B) system. An F/B system, in its usual form, uses a two-level queue-scheduling discipline. The workload of the system is divided into two queues-a high-priority queue of interactive & time-critical processes & other processes that do not service external events. The foreground queue is serviced in the event-driven manner, & it can preempt processes executing in the background.

3.2.6.7 Multiple-Level Queues with Feedback Scheduling

Multiple queues in a system may be used to increase the effectiveness & adaptive ness of scheduling in the form of multiple-level queues with feedback. Rather than having fixed classes of processes allocated to specific queues, the idea is to make traversal of a process through the system dependent on its run-time behavior. For example, each process may start at the top-level queue. If the process is completed within a given time slice, it departs the system after having received the royal treatment. Processes that need more than one time slice may be reassigned by the operating system to a lower-priority queue, which gets a lower percentage of the processor time. If the process is still now finished after having run a few times in that queue, it may be moved to yet another, lower-level queue. The idea is to give preferential treatment to short processes & have the resource-consuming ones slowly "sink" into lower-level queues, to be used as fillers to keep the processor utilization high. This philosophy is supported by program-behavior research findings suggesting that completion rate has a tendency to decrease with attained service. In other words, the more service a process receives, the less likely it is to complete if given a little more service. Thus the feedback in MLQ mechanisms tends to rank the processes dynamically according to the observed amount of attained service, with a preference for those that have received less.

On the other hand, if a process surrenders control to the OS before its time slice expires, being moved up in the hierarchy of queues may reward it. As before, different queues may be serviced using different scheduling discipline. In contrast to the ordinary multiple-level queues, the introduction of feedback makes scheduling adaptive & responsive to the actual, measured run-time behavior of processes, as opposed to the fixed classification that may be defeated by incorrect guessing or abuse of authority. A multiple-level queue with feedback is the most general scheduling discipline that may incorporate any or all of the simple scheduling strategies discussed earlier. Its overhead may also combine the elements of each constituent scheduler, in addition to the overhead imposed by the global queue manipulation & the process-behavior monitoring necessary to implement this scheduling discipline.

3.3 SUMMARY

An important, although rarely explicit, function of process management is processor allocation. Three different schedulers may coexist & interact in a complex operating system: long-term scheduler, medium-term scheduler, & short-term scheduler. Of the presented scheduling disciplines, FCFS scheduling is the easiest to implement but is a poor performer. SRTN scheduling is optimal but unrealizable. RR scheduling is most popular in time-sharing environments, & event-driven & earliest-deadline scheduling are dominant in real-time & other systems with time-critical requirements. Multiple-level queue scheduling, & its adaptive variant with feedback, is the most general scheduling discipline suitable for complex environments that serve a mixture of processes with different characteristics.

3.4 Keywords

Long-term scheduling: the decisions to introduce new processes for execution, or re-execution.

Medium-term scheduling: the decision to add to (grow) the processes that are fully or partially in memory.

Short-term scheduling: the decisions as to which (Ready) process to execute next.

Non-preemptive scheduling: In non-preemptive scheduling, process will continue to execute until it terminates, or makes an I/O request which would block the process, or makes an operating system call.

In preemptive scheduling, the process may be pre-empted by the operating system when a new process arrives (perhaps at a higher priority), or an interrupt or signal occurs, or a (frequent) clock interrupt occurs.

3.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. Discuss various process scheduling policies with their cons & pros.
2. Define process. What is the difference between a process & a program? Explain.
3. What are the different states of a process? Explain using a process state transition diagram.
4. Which type of scheduling is used in real life operating systems? Why?
5. Which action should the short-term scheduler take when it is invoked but no process is in the ready state? Is this situation possible?
6. How can we compare performance of various scheduling policies before actually implementing them in an operating system?
7. SJF is a sort of priority scheduling. Comment.
8. What do you understand by starvation? How does SJF cause starvation? What is the solution of this problem?

9. What qualities are to be there in a scheduling policy? Explain.
10. Differentiate between user-oriented scheduling criteria & system-oriented scheduling criteria.

3.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 5	***Writer: Dr. Rakesh Kumar***
Memory Management - I	***Vetter: Prof. Dharminder Kr.***

5.0 OBJECTIVE

The lesson presents the principles of managing the main memory, one of the most precious resources in a multiprogramming system. In our sample hierarchy of OS layers, memory management belongs to layer 3. Memory management is primarily concerned with allocation of physical memory of finite capacity to requesting processes. No process may be activated before a certain amount of memory can be allocated to it. The objective of this lesson is to make the students acquainted with the concepts of contiguous memory management.

5.1 INTRODUCTION

Memory is large array of words or bytes, each having its unique address. CPU fetches instructions from memory according to value of program counter. The instructions undergo instruction execution cycle. To increase both CPU utilization & speed of its response to users, computers must keep several processes in memory. Specifically, the memory management modules are concerned with following four functions:

1. Keeping track of whether each location is allocated or unallocated, to which process & how much.
2. Deciding to whom should the memory is allocated, how much, when & where. If memory is to be shared by more than one process concurrently, it must be determined which process' request should be satisfied.
3. Once it is decided to allocate memory, the specific locations must be selected & allocated. Memory status information is updated.
4. Handling the deallocation/reclamation of memory. After the process holding memory is finished, memory locations held by it are declared free by changing the status information.

There are varieties of memory management systems. They are:

1. Contiguous, real memory management system such as:
   - Single, contiguous memory management system
   - Fixed partitioned memory management system
   - Variable Partitioned memory management system
2. Non-Contiguous, real memory management system
   - Paged memory management system
   - Segmented memory management system
   - Combined memory management system
3. Non-Contiguous, virtual memory management system
   - Virtual memory management system

These systems can be divided into two major parts (i) Contiguous & (ii) Non-Contiguous

Contiguous Memory Management: In this approach, each program occupies a single contiguous block of storage locations.

Non-Contiguous Memory Management: In these, a program is divided into several blocks or segments that may be placed throughout main storage in pieces or chunks not necessarily adjacent to one another. It is the function of OS to manage these different chunks in such a way that they appear to be contiguous to the user.

Various issues to be considered in various memory management schemes are relocation, address translation, protection, sharing, & evaluation.

Relocation & address translation: The process of associating program instructions & data to physical memory addresses is called address binding or relocation. So binding is mapping from one address to another. It is of two types:

- Static Binding: It is taking place before execution; it may be (i) Compile time: where the compiler or assembler translates symbolic addresses to absolute addresses & (ii) Load time where the compiler translates symbolic addresses to relative addresses. The loader translates these to absolute addresses.
- Dynamic Binding: In it new locations are determined during execution. The program retains its relative addresses. The absolute addresses are generated by hardware.

Memory Protection & Sharing: Protection is used to avoid interference between programs existing in memory. Sharing is the opposite of protection.

Evaluation: Evaluation of these schemes is done on various parameters such as:

- **Wasted memory:** It is the amount of physical memory, which remains unused & thus wasted.
- **Access time** is the time to access the physical memory by the OS.
- **Time complexity** is related to overheads of the allocation or deallocation methods.

5.2 PRESENTATION OF CONTENTS

5.2.1 SINGLE CONTIGUOUS MEMORY MANAGEMENT

5.2.2 FIXED PARTITIONED MEMORY MANGEMENT SYSTEM

5.2.2.1 Principles of Operation

5.2.2.2 Fragmentation

5.2.2.3 Swapping

5.2.2.4 Relocation

5.2.2.4.1 Static Relocation

5.2.2.4.2 Dynamic Relocation

5.2.2.5 Protection

5.2.2.6 Sharing

5.2.2.7 Evaluation

5.2.3 VARIABLE PARTITIONED MEMORY ALLOCATION

5.2.3.1 Principles of Operation

52.3.2 Compaction

5.2.3.3 Protection

5.2.3.4 Sharing

5.2.3.5 Evaluation

5.2.4 SEGMENTATION

5.2.4.1 Principles of Operation

5.2.4.2 Protection

5.2.4.3 Sharing

5.2.1 SINGLE CONTIGUOUS MEMORY MANAGEMENT

In this scheme, the physical memory is divided into two contiguous areas. One of them is permanently allocated to the resident portion of the OS. Mostly, the OS resides in low memory (0 to P as shown in Figure 1). The remaining memory is allocated to transient or user processes, which are loaded & executed one at a time, in response to user commands. This process is run to completion & then the next process is brought in memory.

In this scheme, the starting physical address of the program is known at the time of compilation. The machine contains absolute addresses. They do not need to be changed or translated at the time of execution. So there is no issue of relocation or address translation.

00

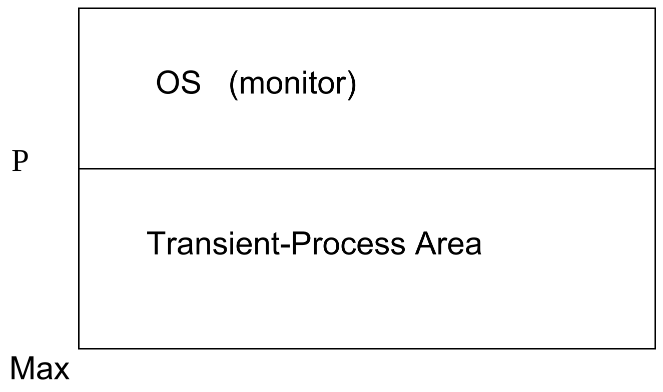


Figure 1. Single contiguous memory management

In this scheme as there is at most one process is in memory at any given time so there is a rare issue of interference between programs. However, it is desirable to protect the OS code from being tampered by the executing transient process.

A common way used in embedded systems to protect the OS code from user programs is to place the OS in read-only memory. This method is rarely used because of its inflexibility & inability to patch & update the OS code. In systems where the OS is in read-write memory, protection from user processes usually requires some sort of hardware assistance such as the fence registers & protection bits.

Fence registers are used to draw a boundary between the OS & the transient-process area. Assuming that the resident portion of the OS is in low memory, the fence register is set to the highest address occupied by OS code. Each memory address generated by a user process is compared against the fence. Any attempt to read or write the space below the fence may thus be detected & denied before completion of the related memory reference. Such violations usually trap to the OS, which in turn may abort the offending program. To serve the purpose of protection, modification of the fence register must be a privileged operation not executable by user processes. Consequently, this method requires the hardware ability to distinguish between execution of the OS & of user processes, such as the one provided by user & supervisor modes of operation.

Another approach to memory protection is to record the access rights in the memory itself. One possibility is to associate a protection bit with each word in memory. The memory may then easily be divided into two zones of arbitrary size by setting all protection bits in one area, & resetting them in the other area. For example, initially all protection bits may be reset. During system startup, protection bits may be set in all locations where the OS is loaded. User programs may then be loaded & executed in the remaining memory locations. Prohibiting user processes from accessing any memory location whose protection bit is set may enforce OS protection. At the same time, the OS & system utilities, such as the loader, may be allowed unrestricted access to memory necessary for their activities. This approach requires a hardware-supported distinction between at least two distinct levels of privilege in the execution of machine instructions.

Sharing of code & data in memory does not make much sense in single-process environments, & single-process OS hardly ever support it. Users' programs may of course, pass data to teach other in private arrangements, say, by means of memory locations known to be safe from being overwritten between executions of participating processes. Such schemes are obviously unreliable, & their use should be avoided whenever possible.

Single-process OS are relatively simple to design & to comprehend. They are often used in systems with little hardware support. But the lack of support for multiprogramming reduces utilization of both processor & memory. Processor cycles are wasted because there is no pending work that may be executed while the running process is waiting for completion of its I/O operations. Memory is underutilized because its portion not devoted to the OS & the single active user is wasted. On the average, wasted memory in a specific system is related to the difference between the size of the transient-process area & the average process size weighted by the respective process-execution (and residence) times. This method has fast access time & very little time-complexity. Its usage is limited due to lack of multi-user facility.

One additional problem is sometimes encountered in systems with simplistic static forms of memory management. To be useable across a wide range of configurations with different capacities of installed memory, system programs in such environments tend to be designed to use the least amount of memory possible. Besides sacrificing speed & functionality, such programs usually take little advantage of additional memory when it is available.

5.2.2 FIXED PARTITIONED MEMORY MANGEMENT SYSTEM

In this scheme, memory is divided into number of contiguous regions called partitions, could be of different sizes. But once decided, they could not be changed. Partitions are fixed at the time of system generation. System generation is a process of setting the OS to specific requirements. Various processes of the OS are allotted different partitions. There are two forms of memory partitioning (i) Fixed Partitioning & (ii) Variable Partitioning.

In fixed partitioning the main memory is divided into fixed number of partitions during system startup. The number & sizes of individual partitions are decided by the factors like capacity of the available physical memory, desired degree of multiprogramming, & the typical sizes of processes most frequently run on a given installation. Since, in principle, at most one process may execute out of a given partition at any time, the number of partitions represents an upper limit on the number of active processes in a system i.e. degree of multiprogramming. Given the impact of memory partitioning on overall performance, some systems allow for manual redefinition of partition sizes.

Programs are queued to run in the smallest available partition. An executable prepared to run in one partition may not be able to run in another without being relinked. This technique is called absolute loading.

5.2.2.1 Principles of Operation

An example of partitioned memory is depicted in Figure 2. Out of the six partitions, one is assumed to be occupied by the resident portion of the OS, & three others by user processes P_i, P_j, & P_k, as indicated. The remaining two partitions, shaded in Figure 2, are free & available for allocation.

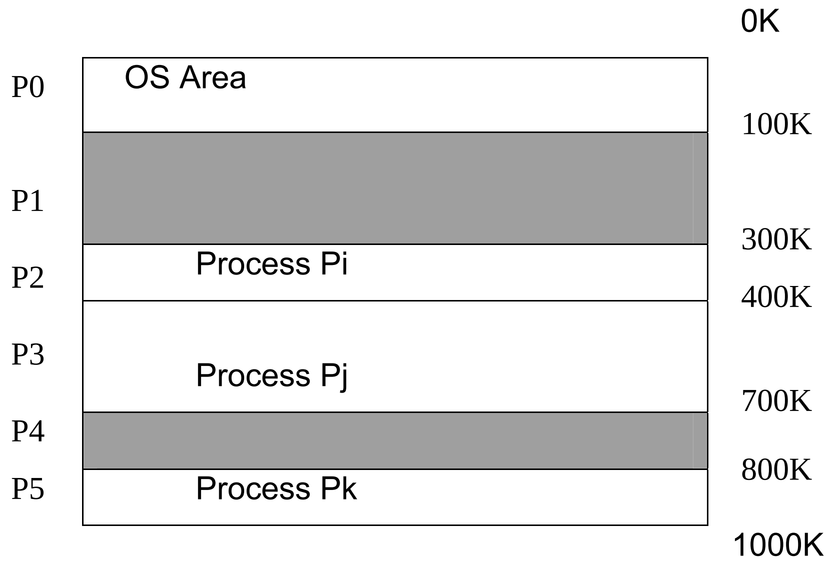


Figure 2 – Fixed Partitions

On declaring fixed partitions, the OS creates a Partition Description Table (PDT) to keep track of status of each partition for allocation purposes. A sample PDT format is given in Figure 3 according to the partitions given in Figure 2.

Partition Number	Partition Base	Partition size	Partition Status
0	0K	100K	Allocated
1	100K	200K	Free
2	300K	100K	Allocated
3	400K	300K	Allocated
4	700K	100K	Free
5	800K	200K	Allocated

Figure 3 – Partition description table

Each partition is described by its base address, size, & status. When fixed partitioning is used, only the status field of each entry varies i.e. free or allocated, in the course of system operation. Initially, all the entries are marked “FREE”. As & when process is loaded into partitions, the status entry for that partition is changed to “ALLOCATED”.

Initially, all memory is available for user processes & is called hole. On arrival of a process, a hole large enough for that process is allocated to it. The OS then reads the program image from disk to the space reserved. After becoming resident in memory, the newly loaded process makes a transition to the ready state & thus becomes eligible for execution.

When a nonresident process is to be activated, the OS searches a free memory partition of sufficient size in the PDT. If the search is successful, the status field of the selected entry is marked ALLOCATED, & the process image is loaded into the corresponding partition. Since the assumed format of the PDT does not provide any indication as to which process is occupying a given partition, the identity of the assigned partition may be recorded in the PCB. When the process departs, using this information the status of related partition is made FREE. To implement these ideas, two questions are to be answered, (i) how to select a specific partition for a given process, (ii) What to do when no suitable partition is available for allocation. The strategies of partition allocation are:

First-fit: This strategy allocates the first available space that is big enough to accommodate process. Search may start at beginning of set of holes or where previous first-fit ended. Searching stops as soon as it finds a free hole that is large enough.

Best-fit: This strategy allocates the smallest hole that is big enough to accommodate process. Entire list ordered by size is searched & matching smallest left over hole is chosen.

Worst fit: This strategy allocates the largest hole. Entire list is searched. It chooses largest left over hole.

These strategies may be compared on the basis of execution speed & memory utilization must be made. These algorithms have to search the PDT to identify a free partition of adequate size. However, while the first fit terminates upon finding the first such partition, the best fit must process all PDT entries to identify the tightest fit. So first fit tend to execute faster but best fit may achieve higher utilization of memory by creating the smallest possible gap resulting from the difference in size between the process & its allocated partition. Both first-fit & best fit are better than worst-fit in terms of time & storage utilization, but first-fit is faster.

In case of a relatively small number of fixed partitions in a system, the execution time differences between the these approaches may not be large enough to outweigh the lower degree of memory utilization attributable to the first fit. When the number of partitions is large neither first fit nor best fit is clearly superior.

Request for partitions may be due to (1) creations of new processes or (2) reactivations of swapped-out processes. The memory manager attempts to satisfy these requests from the pool of free partitions. Common obstacles faced by it are:

1. No free partition is large enough to accommodate the incoming process.
2. All partitions are allocated.
3. Some partitions are free, but none of them is large enough to accommodate the incoming process.

If the process to be created is too large to fit into any of the system partitions, the OS produces an error message. This is basically a configuration error that may be remedied by redefining the partitions accordingly. Another option is to reduce a program's memory requirements by recording & possibly using some sort of overlays.

The case when all partitions are allocated may be handled by deferring loading of the incoming process until a suitable partition can be allocated to it. An alternative is to force a memory-resident process to vacate a sufficiently large partition. Eviction to free the necessary space incurs the additional overhead of selecting a suitable victim & rolling it out to disk. This technique is called swapping. Both deferring & swapping are also applicable to handling the third case, where free but unsuitable partitions are available. If the deferring option is chosen, memory utilization may be kept high if the OS continues to allocate free partitions to other waiting processes with smaller memory requirements. However, doing so may violate the ordering of process activation’s intended by the scheduling algorithm and, in turn, affect performance of the system.

The described memory-allocation situations illustrate the close relationship & interaction between memory management & scheduling functions of the OS. Although the division of labor in actual systems may vary, the memory manager is generally charged with implementing memory allocation & replacement policies. Processor scheduling, on the other hand, determines which process gets the processor, when, & for how long. The short-term scheduler considers only the set of ready processes, that is, those that have all the needed resources except for the processor. Ready processes are, by definition, resident in memory. By influencing the membership of the set of resident processes, a memory manager may affect the scheduler's ability to perform. On the other hand, the effectiveness of the short-term scheduler influences the memory manager by affecting the average memory-residence times of processes.

In systems with fixed partitioning of memory, the number of partitions effectively sets an upper limit on the degree of multiprogramming. Within the confines of this limit, processor utilization may be improved by increasing the ratio of ready to resident processes. This may be accomplished by removing suspended processes from memory when otherwise ready ones are available for loading in the related partitions. A removed process is usually kept in secondary storage until all resources needed for its execution, except for memory & the processor may be allocated to it. At that point, the process in question becomes eligible for loading into the main memory. The medium-term scheduler & the memory manager cooperate in further processing of such processes.

The OS holds the processes waiting to be loaded in the memory in a queue. The two methods of maintaining this queue are (i) Multiple Queues & (ii) Single Queues.

Multiple Queues: In this method there are as many queues as the number of partitions. Separate queue for each partition is maintained in which processes are added as they arrive. When a process wants to occupy memory, it is added to a proper queue depending upon size of processes. Benefit of this method is that a small process is not loaded in large partition so as to avoid memory wastage. This leads to longer queue for small partitions.

Single Queue: In this method, there is only one queue for all ready processes. The order of processes in the queue depends on the scheduling algorithm. In this case, first fit allocation strategy is more efficient & fast.

5.2.2.2 Fragmentation

Some amount of memory is wasted both in single & multiple partition allocation techniques. Fragmentation refers to the unused memory that the memory management system cannot allocate. It is of two types: External & Internal. **External Fragmentation** is waste of memory between partitions caused by scattered non-contiguous free space. It occurs when total available memory space is enough to satisfy the request for a process to be allocated, but it is not continuous. Selection of first fit & best fit can affect the amount of fragmentation. It is severe in variable size partitioning schemes. Compaction is a technique that is used to overcome this.

Internal fragmentation is waste of memory within a partition caused by difference between size of partition & the process allocated. It refers to the amount of memory, which is not being used & is allocated along with a process request i.e. available memory internal to partition. It is severe in fixed partitioning schemes.

5.2.2.3 Swapping

Removing suspended or preempted processes from memory & their subsequent bringing back is called swapping. The basic idea of swapping is to treat main memory as a 'pre-emptable' resource. Lifting the program from the memory & placing it on the disk is called 'Swapping out'. To bring the program again from the disk into the main memory is called 'Swapping in'. Normally, a blocked process is swapped out so as to create available space for a ready process. This results in improving CPU utilization. Swapping has traditionally been used to implement multiprogramming in systems with restrictive memory capacity. Swapping may also be helpful for improving processor utilization in partitioned memory environments by increasing the ratio of ready to resident processes. Swapping is usually employed in memory-management systems with contiguous allocation, such as fixed & variable partitioned memory & segmentation. Somewhat modified forms of swapping may also be present in virtual memory systems based on segmentation or on paging. Swapping brings flexibility even to systems with fixed partitions.

When the scheduler decides to admit a new process for which no suitable free partition can be found, the swapper may be invoked to vacate such a partition. The swapper is an OS process whose major responsibilities include:

- Selection of processes to swap out: Its criteria is suspended/blocked state, low priority, time spent in memory.
- Selection of processes to swap in: Its criteria are time spent on swapping device & priority.
- Allocation & management of swap space on a swapping device. Swap space can be system wide or dedicated.

Thus the swapper performs most of the functions of the medium-term scheduler. The swapper usually selects a victim among the suspended processes that occupy partitions large enough to satisfy the needs of the incoming process.
Although the mechanics of swapping out following the choice of a victim process is fairly simple in principle, implementation of swapping requires some specific provisions & considerations in OS that support it. These generally include the file system, specific OS services, & relocation.

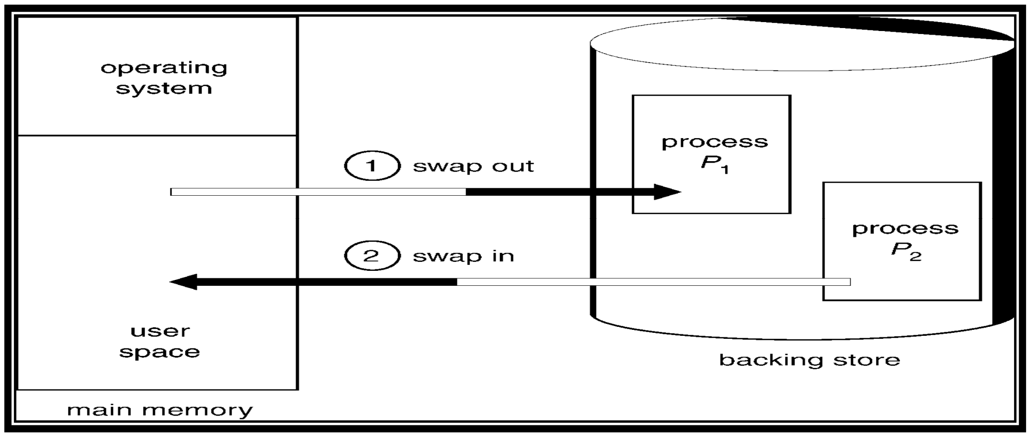


Figure 4 showing process of Swapping

A process is typically prepared for execution & submitted to the OS in the form of a file that contains a program in executable form & the related data. This file may also contain process attributes, such as priority & memory requirements. Such a file is sometimes called a process image. Since a process usually modifies its stack & data when executing, a partially executed process generally has a run-time image different from its initial static process image recorded on disk. Therefore, the dynamic run-time state of the process to be swapped out must be recorded for its proper subsequent resumption. In general, the modifiable portion of a process's state consists of the contents of its data & stack locations, as well as of the processor registers. Code is also subject to run-time modifications in systems that permit the code to modify itself. Therefore, the contents of a sizable portion or of the entire address space of a victim process must be copied to disk during the swapping-out operation. Since the static process image is used for initial activation, the (modified) run-time image should not overwrite the static process image on disk. Consequently, a separate swap file must be available for storing the dynamic image of a rolled-out process. There are two basic options regarding placement of a swap file:

- System-wide swap file
- Dedicated, per-process, swap files

In either case, swapping space for each swappable process is usually reserved & allocated statically, at process creation time, to avoid the overhead of this potentially lengthy operation at swap time.

In the system-wide swap file approach, a single large file is created, usually in the course of system initialization, to handle swapping requirements of all processes. The swap file is commonly placed on a fast secondary-storage device so as to reduce the latency of swapping. The location of each swapped out process image is noted within that file. An important trade-off in implementing a system-wide swap file is the choice of its size. If a smaller area is reserved for this file, the OS may not be able to swap out processes beyond a certain limit, thus affecting the performance.

An alternative is to have a dedicated swap file for each swappable process in the system. These swap files may be created either dynamically at process creation time or statically at program preparation time. This method is very flexible, but can be very inefficient due to the increased number of files & directories. In either case, the advantages of maintenance of separate swap files include elimination of the system swap-file dimensioning problem & of that file's overflow errors at run-time, & non-imposition of restrictions on the number of active processes. The disadvantages include more disk space expended on swapping, slower access, & more complicated addressing of swapping files scattered on the secondary storage.

Regardless of the type of swapping file used, the need to access secondary storage makes swapping a lengthy operation relative to processor instruction execution. This overhead must be taken into consideration in the decision of whether to swap a process in order to make room for another one.

Delays of this magnitude may be unacceptable for interrupt-service routines or other time-critical processes. For example, swapping out of a momentarily inactive terminal driver in a time-sharing system is certainly a questionable "optimization." OS that support swapping usually cope with this problem by providing some means for system programmers to declare a given process as being swappable or not. In effect, after the initial loading, an unswappable process remains fixed in memory even when it is temporarily suspended. Although this service is useful, a programmer may abuse it by declaring an excessive number of processes as fixed, thereby reducing the benefits of swapping. For this reason, the authority to designate a process as being un-swappable is usually restricted to a given class of privileged processes & users. All other processes, by default, may be treated as swappable.

An important issue in systems that support swapping is whether process-to-partition binding is static or dynamic, i.e., whether a swapped-out process can subsequently be loaded only into the specific partition from which it was removed or into any partition of adequate size. In general, static binding of processes to partitions may be done in any system with static partitioning of memory, irrespective of whether swapping is supported or not. Static process-to-partition binding eliminates the run-time overhead of partition allocation at the expense of lower utilization of memory due to potentially unbalanced use of partitions. On the other hand, systems in which processes are not permanently bound to specific partitions are much more flexible & have a greater potential for efficient use of memory. The price paid for dynamic binding of processes to partitions is the overhead incurred by partition allocation whenever a new process or a swapped process is to be loaded into main memory. Moreover, dynamic allocation of partitions usually requires some sort of hardware support for dynamic relocation.

5.2.2.4 Relocation

The term program relocatability refers to the ability to load & execute a given program into an arbitrary place in memory. Since different load addresses may be assigned during different executions of a single relocatable program, a distinction is often made between virtual addresses (or logical address) & the physical addresses where the program & its data are stored in memory during a given execution. In reality, the program may be loaded at different memory locations, which are called physical addresses. The problem of relocation & address translation is to find a way to map virtual addresses onto physical addresses. Depending on when & how the mapping from the virtual address space to the physical address space takes place in a given relocation scheme, there are two basic types of relocation: (i) Static relocation & (ii) Dynamic relocation.

5.2.2.4.1 Static Relocation

Static relocation is performed before or during the loading of the program into memory, by a relocating linker/ loader. Constants, physical I/O port addresses, & offsets relative to the program counter are examples of values that are not location-sensitive & that do not need to be adjusted for relocation. Other forms of addresses of operands may depend on the location of a program in memory so must be adjusted accordingly when the program is being loaded or moved to a different area of memory.

A language translator typically prepares the object module by assuming the virtual address 0 to be the starting address of the program, thus making virtual addresses relative to the program loading address. Relocation information, including virtual addresses that need adjustment following determination of the physical load address, is provided for subsequent processing by the linker & loader. Either when the linker combines object modules or when the process image is being loaded, all program locations that need relocation are adjusted in accordance with the actual starting physical address allocated to the program. Once the program is in memory, values that need relocation are indistinguishable from those that do not.

Since relocation information in memory is usually lost following the loading, a partially executed statically relocatable program cannot be simply copied from one area of memory into another & be expected to continue to execute properly. In systems with static relocation a swapped-out process must either be swapped back into the same partition from which it was evicted, or software relocation must be repeated whenever the process is to be loaded into a different partition.

Given the considerable space & time complexity of software relocation, systems with static relocation are practically restricted to supporting only static binding of processes to partitions. This method is slow process because it involves software translation. It is used only once before the initial loading of the program.

5.2.2.4.2 Dynamic Relocation

In it, mapping from the virtual address space to the physical address space is performed at run-time. Process images in systems with dynamic relocation are also prepared assuming the starting location to be a virtual address 0, & they are loaded in memory without any relocation adjustments. When the related process is being executed, all of its memory references are relocated during instruction execution before physical memory is actually accesses. This process is often implemented by means of specialized base registers. After allocating a suitable partition & loading a process image in memory, the OS sets a base register to the starting physical load address. This value is normally obtained from the relevant entry of the PDT. Each memory reference generated by the executing process is mapped into the corresponding physical address by having the contents of the base register added to it.

Dynamic relocation is illustrated in Figure 5. A sample process image prepared with an assumed starting address of virtual address 0 is shown unchanged before & after being loaded in memory. In this particular example, it is assumed that address 1000 is allocated as the starting address for loading the process image. This base address is normally available from the corresponding entry of the PDT, which is reachable by means of the link to the allocated partition in the PCB. Whenever the process in question is scheduled to run, the base register is loaded with this value in the course of process switching.

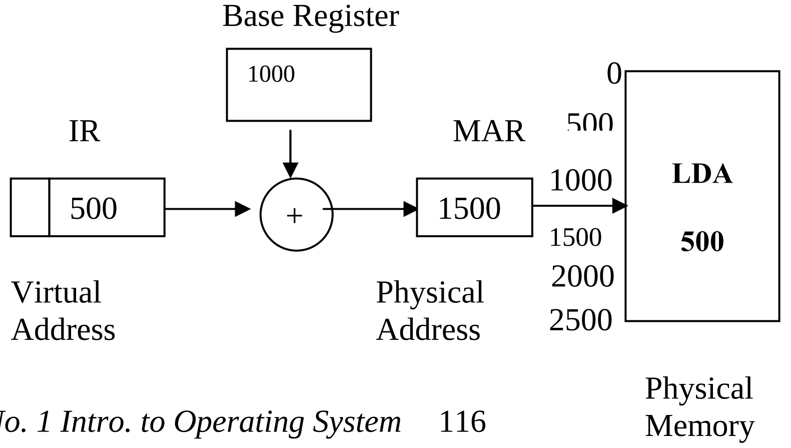


Figure 5 – Dynamic relocation

Relocation of memory references at run-time is illustrated by means of the instruction LDA 500, which is supposed to load the contents of the virtual address 500 (relative to program beginning) into the accumulator. As indicated, the target item actually resides at the physical address 1500 in memory. This address is produced by hardware by adding the contents of the base register to the virtual address given by the processor at run-time.

As suggested by Figure 4, relocation is performed by hardware & is invisible to programmers. In effect, all addresses in the process image are prepared by counting on the implicit based addressing to complete the relocation process at run-time. This approach makes a clear distinction between the virtual & the physical address space.

This is the most commonly used scheme amongst the schemes using fixed partitions due to its enhanced speed & flexibility. Its advantage is that it supports swapping easily. Only the base register value needs to be changed before dispatching.

5.2.2.5 Protection

Not only must the OS be protected from unauthorized tampering by user processes, but each user process must also be prevented from accessing the areas of memory allocated to other processes. Otherwise, a single erroneous or malevolent process may easily corrupt any or all other resident processes. There are two approaches for preventing such interference & achieving protection. These approaches involve the use of Limit Register & Protection Bits.

Implementation of memory protection in a given system tends to be greatly influenced by the available hardware support. In systems that use base registers for relocation, a common practice is to use limit registers for protection. The primary function of a limit register is to detect attempts to access address space beyond the boundary assigned to the executing program by the OS. The limit register is usually set to the highest virtual address in a program. As illustrated by Figure 6, each intended memory reference of an executing program is checked against the contents of the limit register before being forwarded to memory. In this way, any attempt to access a memory location outside of the specified area is detected & aborted by the protection hardware before being allowed to reach the memory. This violation usually traps to the OS, which may then take a remedial action, such as to terminate the offending process. The base & limit values for each process are normally kept in its PBC. Upon each process switch, the hardware base & limit registers are loaded with the values required for the new running process. Another approach to protection is to record the access rights in the memory itself. The bit-per-word approach described earlier, is not suitable for multiprogramming systems because it can separate only two distinct address spaces. Adding more bits to designate the identity of each word's owner may solve this problem, but this approach is rather costly. A more economical version of this idea has been implemented by associating a few individual words. For example, some models of the IBM 360 series use four such bits, called keys, per each 2 KB block of memory. When a process is loaded in memory, its identity is recorded in the protection bits of the occupied blocks. The validity of memory references is established at run-time by comparison of the running process's identity to the contents of protection bits of the memory block being accessed. If no match is found, the access is illegal & hardware traps to the OS for processing of the protection-violation exception. The OS is usually assigned a unique "master" key, say 0, that gives it unrestricted access to all blocks of memory. Note that this protection mechanism imposes certain restrictions on operating-system designers. For example, with 4-bit keys the maximum number of static partitions & of resident processes is 16. Likewise, associating protection bits with fixed-sized blocks forces partition sizes to be an integral number of such blocks.

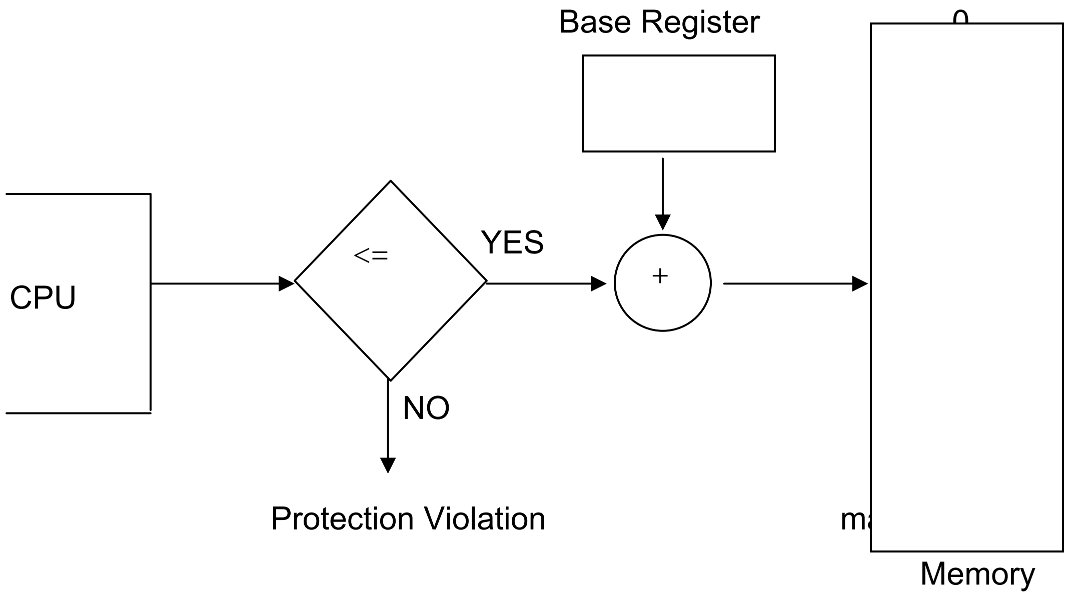


Figure 6 – Base-limit register

5.2.2.6 Sharing

Sharing of code & data poses a serious problem in fixed partitions because it might compromise on protection. There are three basic approaches to sharing in systems with fixed partitioning of memory:

- Entrust shared objects to OS.
- Maintain multiple copies, one per participating partition, of shared objects.
- Use shared memory partitions.

The easiest way to implement sharing without significantly compromising protection is to entrust shared objects to the OS. It means that any code or data goes through the OS for any request because the OS has the controlling access to shared resources. No additional provisions may be needed to support sharing. This scheme is possible but very tedious. Unfortunately, this simple approach increases the burden on the OS. Therefore, it is not followed in practice.

Unless objects are entrusted to the OS, sharing is quite difficult in systems with fixed partitioning of memory. The primary reason is their reliance on rather straightforward protection mechanisms based mostly on the strict isolation of distinct address spaces. Since memory partitions are fixed, disjoint, & usually difficult to access by processes not belonging to the OS, static partitioning of memory is not very conducive to sharing.

Another approach is to keep copies of the sharable code/ data in all partitions where required. It is wasteful & leads to inconsistencies. Since there is no commonly accessible original, each process runs using its copy of the shared object. Consequently, updates are made only to copies of the shared object. For consistency, updates made to any must be propagated to all other copies, by copying the shared data from the address space of the running process to all participating partitions upon every process switch. Swapping, when supported, introduces the additional complexity of potentially having one or more participating address spaces absent from main memory. This approach of sharing does not make much sense in view of the fact that no saving of memory may be expected.

Another traditional simple approach to sharing is to place the data in a dedicated "common" partition. However, any attempt by a participating process to access memory outside of its own partition is normally regarded as a protection violation. In systems with protection keys, changing the keys of all shared blocks upon every process switch in order to grant access rights to the currently running process may circumvent this obstacle. Keeping track of which blocks are shared & by whom, as well as the potentially frequent need to modify keys, results in notable OS overhead necessary to support this form of sharing. With base-limit registers, the use of shared partitions outside of-and potentially discontiguous to-the running process's partition requires some special provisions.

5.2.2.7 Evaluation

- Wasted memory: In fixed partitions, lot of memory is wasted due to both kinds of fragmentation.

- Access Time: Access time is not very high due to the assistance of special hardware. The translation from virtual address to physical address is done by hardware itself, thus enabling rapid access.
- Time complexity is very low because allocation/ deallocation routines are simple as the partitions are fixed.

5.2.3 VARIABLE PARTITIONED MEMORY ALLOCATION

In variable partitions, the number of partitions & their sizes are variable as they are not defined at the time of system generation. Starting with the initial state of the system, partitions are created dynamically to fit the needs of each requesting process. When a process departs, the memory manager returns the vacated space to the pool of free memory areas from which partition allocations are made. Process is allocated exactly as much memory as required.

5.2.3.1 Principles of Operation

When instructed to load a process image, the memory-management module attempts to create a suitable partition for allocation to the process in question. The first step is to locate a contiguous free area of memory, which is equal to or larger than the process's size declared with the submitted image. If a suitable free area is found, the OS carves out a partition from it to provide an exact fit to the process's needs. The leftover chunk of free memory is returned to the pool of free memory for later use by the allocation module. The partition is created by entering its base, size, & status into the system PDT. A copy of, or some link to, this information is normally recorded in the PCB. After loading the process image into the created partition, the process may be turned over to the OS module appropriate for its further processing, such as the short-term scheduler. If no suitable free area can be allocated, the OS returns an error indication.

When a resident process departs, OS returns the partition's space to the pool of free memory & invalidates the corresponding PDT entry. For swapped-out processes, the OS also invalidates the PCB field where the identity of the allocated partition is normally held.

The OS obviously needs to keep track of both partitions & free memory. Once created, a partition is defined by its base address & size. Those attributes remain essentially unchanged for as long as the related partition exists. In addition, for the purposes of process switching & swapping, it is important to know which partition belongs to a given process.

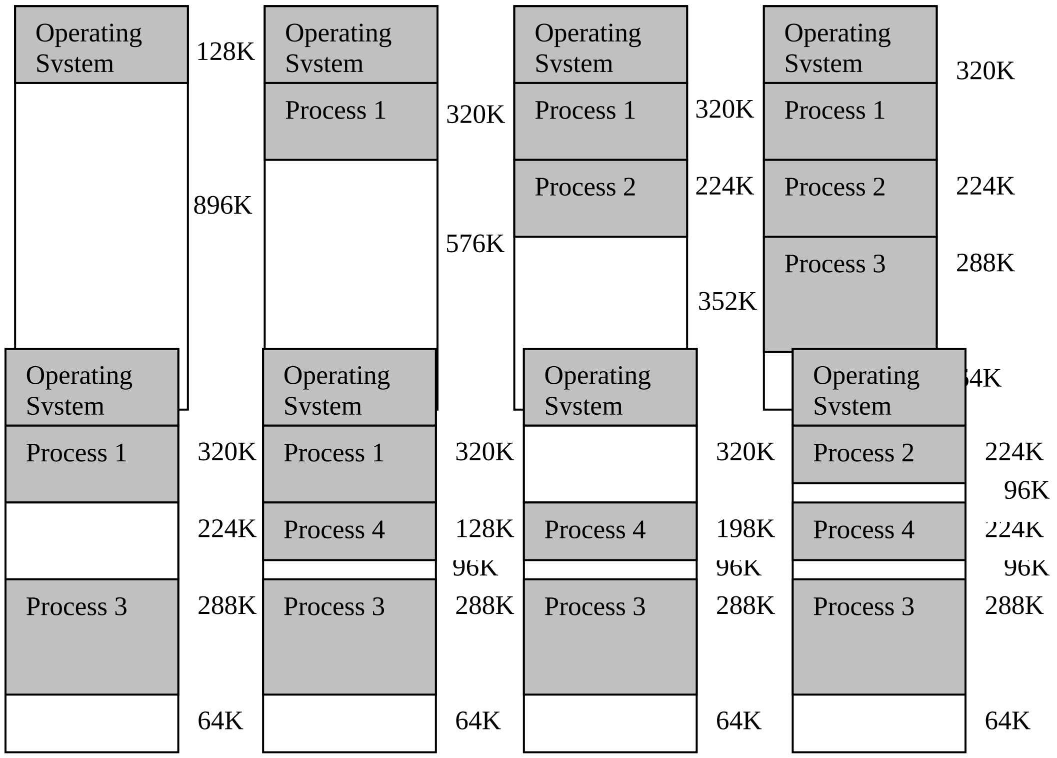


Figure 7– Partitions in dynamic memory partitioning

Free areas of memory are produced upon termination of partitions & as leftovers in the partition creation process. For allocation & for partition creation purpose, the OS must keep track of the starting address & size of each free area of memory. This information may need to be updated each time a partition is created or terminated. The highly dynamic nature of both the number & the attributes of free areas suggest the use of some sort of a linked list to describe them. It is common to conserve space by building the free list within the free memory itself. For example, the first few words of each free area may be used to indicate the size of the area & to house a link to the successor area.

Figure 6 illustrates the working of variable partitioned memory. In this example, first Process 1,Process 2 & Process 3 are allocated memory as they arrive. When Process 2 is swapped out, the memory freed by Process 2 is available for any other process. So it is allocated to Process 4 & the size of partition for process 4 also varies. Again when process 2 arrives, it is allocated memory at different location that was freed by Process 1. Moreover, the size of partition also differs from the size of partition of process 1.From the given example, it is clear that memory is allocated to processes as they arrive & on availability of memory. Partitions are created at the time of allocation according to size of process & not at the time of system generation.

Common algorithms for selection of a free area of memory for creation of a partition (Step 1) are (i) First fit & its variant, next fit, (ii) Best fit, (iii) Worst fit. Next fit is a modification of first fit whereby the pointer to the free list is saved following an allocation & used to begin the search for the subsequent allocation. as opposed to always starting from the beginning of the free list, as is the case with first fit. The idea is to reduce the search by avoiding examination of smaller blocks that tend to be created at the beginning of the free list as a result of previous allocations. In general, next fit was not found to be superior to the first fit in reducing the amount of wasted memory.

First fit is generally faster because it terminates as soon as a free block large enough to house a new partition is found. Thus, on the average, first fit searches half of the free list per allocation. Best fit, on the other hand, searches the entire free list to find the smallest free block large enough to hold a partition being created. First fit is faster, but it does not minimize wasted memory for a given allocation. Best fit is slower, & it tends to produce small leftover free blocks that may be too small for subsequent allocations. However, when processing a series of request starting with an initially free memory, neither algorithm has been shown to be superior to the other in terms of wasted memory.

Worst fit is an antipode of best fit, as it allocates the largest free block, provided the block size exceeds the requested partition size. The idea behind the worst fit is to reduce the rate of production of small holes, which are quite common in best fit. However, some simulation studies indicate that worst fit allocation is not very effective in reducing wasted memory in the processing of a series of requests.

Termination of partitions in a system with dynamic allocation of memory may be performed by means of the procedure that recombines free areas, if possible, to reduce fragmentation of memory. When first fit or best fit is used, the free list may be sorted by address to facilitate recombination of free areas when partitions are deallocated.

Buddy System: This is another method of allocation-deallocation which speeds up merging of adjacent holes. This method facilitates merging of free space by allocating free areas with an affinity to recombine. It treats entire space available as a single block of $2^{k,}$ Requests for free areas are rounded up to the next integer power of base 2. When a free block of size 2^k is requested, the memory allocator attempts to satisfy it by allocating a free block from the list of free blocks of size 2^k. If none is available, the block of the next larger size, 2^{k+1}, is split in two halves (buddies) to satisfy the request. An important property of this allocation scheme is that the base address of the other buddy can be determined given the base address & size of one buddy (for a block of size 2^k, the two addresses differ only in the binary digit whose weight is 2^k). Thus, when a block is freed, a simple test of the status bit can reveal whether its buddy is also free. If so, the two blocks can be recombined to form the twice-larger original block. In addition to the free-list links, a status field is associated with each area of memory to indicate whether it is in use or not. Free blocks of equal size are often kept in separate free lists. Advantage of Buddy System is that it coalesces adjacent buffers or holes. Its major disadvantage is that this method is very inefficient in terms of memory utilisation.

As an example, consider a system with 1MB of memory (100000H) managed using the buddy allocation scheme. An initial request for a 192 KB block in such a system would require allocation of a 256 KB block (rounded up to the size that is a power of 2). Since no such block is initially available, the memory manager would form it by splitting the 1 MB block into two 512 KB buddies, & then splitting one of them to form two 256 KB blocks. The first split produces two 512 KB blocks (buddies) with starting addresses of 00000H (H stands for hexadecimal) & 80000H, respectively. The second split of the block at 80000H yields two 256 KB blocks (buddies) that start at 80000H & A0000H, respectively. Assume that the block starting at A0000H is allocated to the user. When the partition starting as A0000H is eventually terminated, the memory manager can identify the base address of its 256 KB buddy (buddies are of the same size) by toggling the address bit in the position that corresponds to the size of the block being released. In the presented example, 256 KB = 2^{18}, & toggling of that bit yields a 0 (in this example) in bit position 18, which, with all other bits unchanged, produces the address of the original buddy, 80000H. A quick inspection of the associated status word indicates whether the buddy at that address is free or not. If it is, the two buddies are coalesced to reform the 512 KB block starting at address

80000H, which was originally used to produce the smaller blocks to satisfy the pending request.

5.2.3.2 Compaction

It is one solution to problem of external fragmentation. The goal here is to shuffle memory contents & place all free memory together in one block. Compaction is possible only if relocation is dynamic. This technique shifts the necessary process images to bring the free chunks of memory to adjacent positions to coalesce. Coalescing of adjacent free areas is a method often used to reduce fragmentation and the amount of wasted memory. However, such remedies tend to defer the impact of, rather than to prevent, the problem. The primary reason for fragmentation is that, due to different lifetimes of resident objects, the pattern of returns of free areas is generally different from the order of allocations.

When memory becomes seriously fragmented, the only way out may be to relocate some or all partitions into one end of memory & thus combine the holes into one large free area. Since affected processes must be suspended & actually copied from one area of memory into another. It is important to decide when & how memory compaction is to be performed.

Memory compaction may be performed whenever possible or only when needed. Some systems compact memory whenever a process departs, thus collecting most of the free memory into a single large area. An alternative is to compact only upon a failure to allocate a suitable partition, provided that the combined size of free areas exceeds the needs of the request at hand.

Compaction involves a high overhead, but it increases the degree of multiprogramming. That is why; OS can accommodate a process with a larger size, which would have been impossible before compaction.

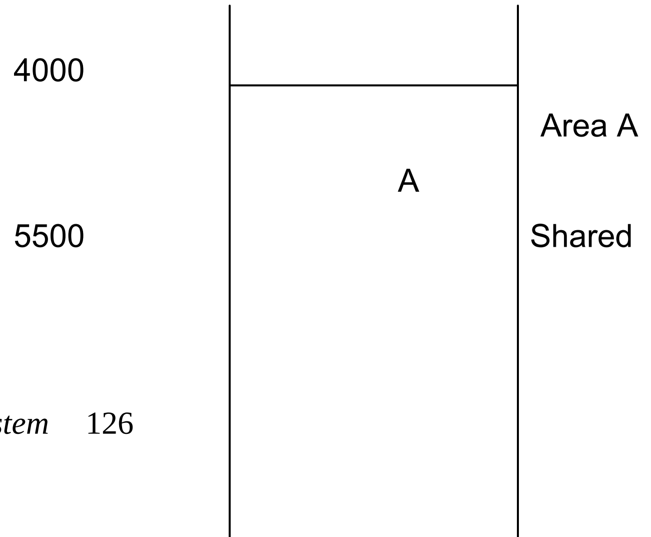
5.2.3.3 Protection

Protection & sharing in systems with dynamic partitioning of memory are not significantly different from their counterparts in static partitioning environments, since they both rely on virtually identical hardware support. One difference is that dynamic partitioning potentially allows adjacent partitions in physical memory to overlap. Consequently, a single physical copy of a shared object may be accessible from two distinct address spaces. This possibility is illustrated in Figure 8, where partitions A & B overlap to include the shared object placed in the doubly shaded area. The relevant portion of the partition of the partition definition table is also shown in Figure 8. As indicated, 500 locations starting from the physical address 5500 are shared & included in both partitions. Although perhaps conceptually appealing, this form of sharing is quite restrictive in practice. Sharing of objects is limited to two processes; when several processes are in play, one of the more involved schemes described in above must be used.

5.2.3.4 Sharing

Sharing of code is generally more restrictive than sharing of data. One of the reasons for this is that shared code must be either reentrant or executed in a strictly mutually exclusive fashion with no preemption's. Otherwise, serious problems may result if a process in the middle of execution of the shared code is switched off, & another process begins to execute the same section of the shared code. Reentrancy generally requires that variables be kept on stack or in registers, so that new activation's do not affect the state of the preempted, incomplete executions. Additional complexities in sharing of code are imposed by the need for shared code to ensure that references to itself-such as local jumps & access to local data-are mapped properly during executions on behalf of any of the participating processes. When dynamic relocation with base registers is used, this means that all references to addresses within a shared-code object from instructions within that code must reach the same set of physical addresses where the shared code is stored at run-time, no matter which particular base is used for a given relocation. This may be accomplished in different ways, such as by making the shared code position independent or by having shared code occupy identical virtual offsets in address spaces of all processes that reference it.

	Base	Size
A	4000	2000
B	5500	2500
Area		



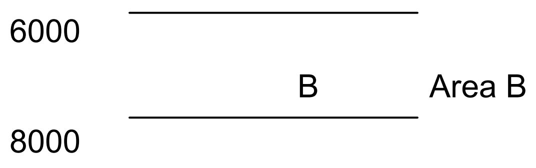


Figure 8 – Overlapping partitions

Some aspects of the issues involved in self-referencing of shared code are illustrated in Figure 9, where a subroutine SUB is assumed to be shared by two processes, PA & PB, whose respective partitions overlap in physical memory as indicated in Figure 8. Let us assume that the system in question used dynamic relocation & dynamic memory allocation, thus allowing the two partitions to overlap. The sizes of the address spaces of the two processes are 2000 & 2500 locations, respectively. The shared subroutine, SUB, occupies 500 locations, & it is placed in locations 5500 to 5999 in physical memory. The subroutine starts at virtual addresses 1500 & 0 in the address spaces of processes PA & PB, respectively. Being shared by the two processes, SUB may be linked with & loaded with either process image.

Figure 9 also shows the references to SUB from within the two processes. As indicated in Figure 9(a), the CALL SUB at virtual address 100 of process PA is mapped to the proper physical address of 5500 at run-time by adding the contents of PA's base register. Likewise the CASS SUB at virtual address 1800 in process PB is mapped to 5500 at run-time by adding PB's value of the base register. This is illustrated in Figure 9(b). Thus proper referencing of SUB from the two processes is accomplished even when the two partitions are relocated due to swapping or compaction, provided that they overlap in the same way in the new set of physical addresses. However, making references from within SUB to itself poses a problem unless some special provisions are made. For example, a jump using absolute addressing from location 50 to location 100 within SUB should read JUMP 1600 for proper transfer of control when invoked by PA, but JMP 100 if PB's invocation is to work properly. Since the JMP instruction may have only one of these two addresses in its displacement field, there is a problem in executing SUB correctly in both possible contexts.

One way to solve this problem is to use relative references instead of absolute references within shared code. For example, the jump in question may read JMP $+50, where $ denotes the address of the JMP instruction. Since it is relative to the program counter, the JMP is mapped properly when invoked by either process, that is, to virtual address 1600 or 100, respectively. At run-time, however, both references map to the same physical address, 5600, as they should. This is illustrated in Figure 9.

Code that executes correctly regardless of its load address is often referred to as position-independent code. One of its properties is that references to portions of the position-independent code itself are always relative, say, to the program counter or to a base when based addressing is used. Position-independent coding is often used for shared code, such as memory-resident subroutine libraries. In our example, use of position-independent code solves the problem of self-referencing.

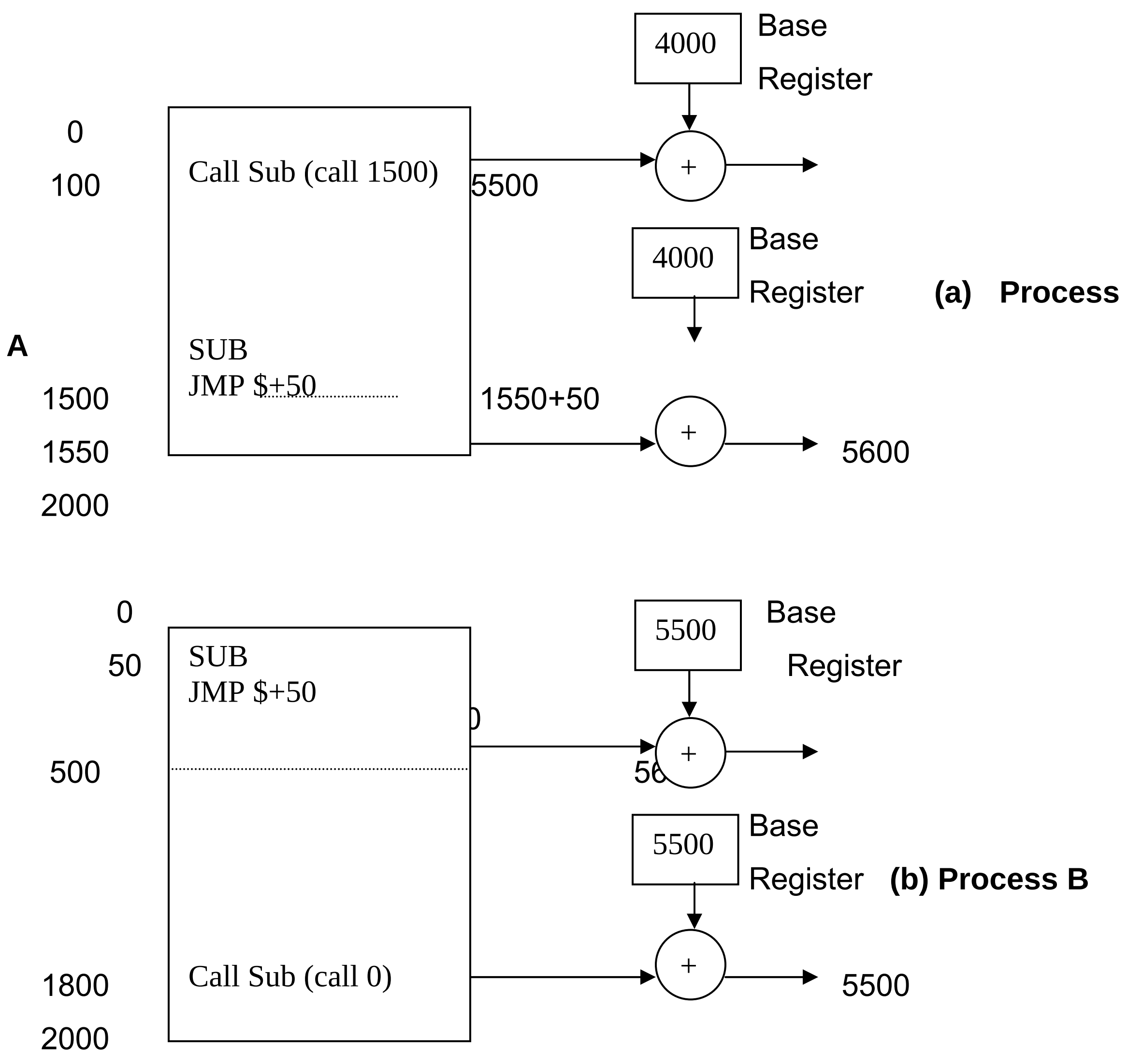


Figure 9 – Accessing shared code (a) Process A (b) Process B

Position-independent coding is one way to handle the problem of self-referencing of shared code. The main point of our example, however, is that sharing of code is more restrictive than sharing of data. In particular, both forms of sharing of code is more restrictive than sharing of data. In particular, both forms of sharing require the shared object to be accessible from all address spaces of which it is a part. ;in addition, shared code must also be reentrant or executed on a mutually exclusive basis, & some special provisions-such as position-independent coding-must be made in order to ensure proper code references to itself. Since ordinary (non-shared) code does not automatically meet these requirements, some special language provisions must be in place, or assembly language coding may be necessary to prepare shared code for execution in systems with partitioned allocation of memory.

5.2.3.5 Evaluation

- Wasted memory: This memory management scheme wastes less memory than fixed partitions because there is no internal fragmentation as the partition size can be of any length. By using compaction, external fragmentation can also be eliminated.
- Access Time: Access time is same as of fixed partitions as the same scheme of address translation using base register is used.
- Time Complexity: Time complexity is higher in variable partitions due to various data structures & algorithms used, for eg: Partition Description Table (PDT) is no more of fixed length.

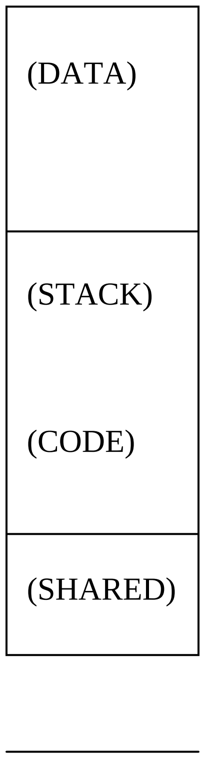
5.2.4 SEGMENTATION

The external fragmentation & its negative impact should be reduced in systems where the average size of a request for allocation is smaller. OS cannot reduce the average process size, but a way to reduce the average size of a request for memory is to divide the address space of a single process into blocks that may be placed into noncontiguous areas of memory. This can be accomplished by segmentation. Segmentation provides breaking of an address space into several logical segments, dynamic relocation & sophisticated forms of protection & sharing.

According to user's view, programs are collections of subroutines, stacks, functions etc. Each of this component is of variable length & are logically related entities. Elements within segment are identified by their offset from beginning of the segment. Segments are formed at program translation time by grouping together logically related items. For example, a typical process may have separate code, data, & stack segments. Data or code shared with other processes may be placed in their own dedicated segments.

All segments of all programs do not have to be of the same length since the segments are formed as a result of logical division. There is a maximum segment length. Although different segments may be placed in separate, noncontiguous areas of physical memory, items belonging to a single segment must be placed in contiguous areas of physical memory. Since segments are not equal, segmentation is similar to dynamic partitioning. Thus segmentation possesses some properties of both contiguous (with regard to individual segments) & noncontiguous (with regard to address space of a process) schemes for memory management.

```
DATA    SEGMENT
.           .
.                  .
datum x   dw          xx
datum y   dw             yy
                .
                .
DATA     ENDS
STACK   SEGMENT
             Ds           500
STACK    ENDS
CODE     SEGMENT
  Psub                 ….
                .
                .
  main                 …
                .
                .
CODE      ENDS
SHARED   SEGMENT
 ssub1        …
                .
                .
 ssub2
SHARED    ENDS
```



Segment Map

Segment#	Size	Type
0	d	data
1	500	stack
2	c	code
3	s	code

(a) Segment Definition **(b) Load Module**

Figure 10 - Segments

Segmentation is quite natural for programmers who tend to think of their programs in terms of logically related entities, such as subroutines & global or local data areas. A segment is essential a collection of such entities. The segmented address space of a single process is illustrated in Figure 10(a). In that particular example, four different segments are defined: DATA, STACK, CODE, & SHARED. Except for SHARED, the name of each segment is chosen to indicate the type of information that it contains. The STACK segment is assumed to consist of 500 locations reserved for stack. The SHARED segment consists of two subroutines, SSUB1 & SSUB2, shared with other processes. The definition of the segments follows the typical assembly-language notation, in which programmers usually have the freedom to define segments directly in whatever way they feel best suits the needs of the program at hand. As a result, a specific process may have several different segments of the same generic type, such as code or data. For example, both CODE & SHARED segments contain executable instructions & thus belong to the generic type "code".

4.2.4.1 Principles of Operation

Segmentation is mapping of user's view onto physical memory. A logical address space is a collection of segments. Each segment has a name & length. User specifies each address by segment name (no.) & offset within segment. Segments are numbered & are referenced by segment number. For relocation purposes, each segment is compiled to begin at its own virtual address 0. An individual item within a segment is then identifiable by its offset relative to the beginning of the enclosing segment. Thus, logical address consists of <segment no., offset>. In segmented systems, components belonging to a single segment reside in one contiguous area. But different segments belonging to same process occupy non-contiguous area of physical memory because each segment is individually relocated.

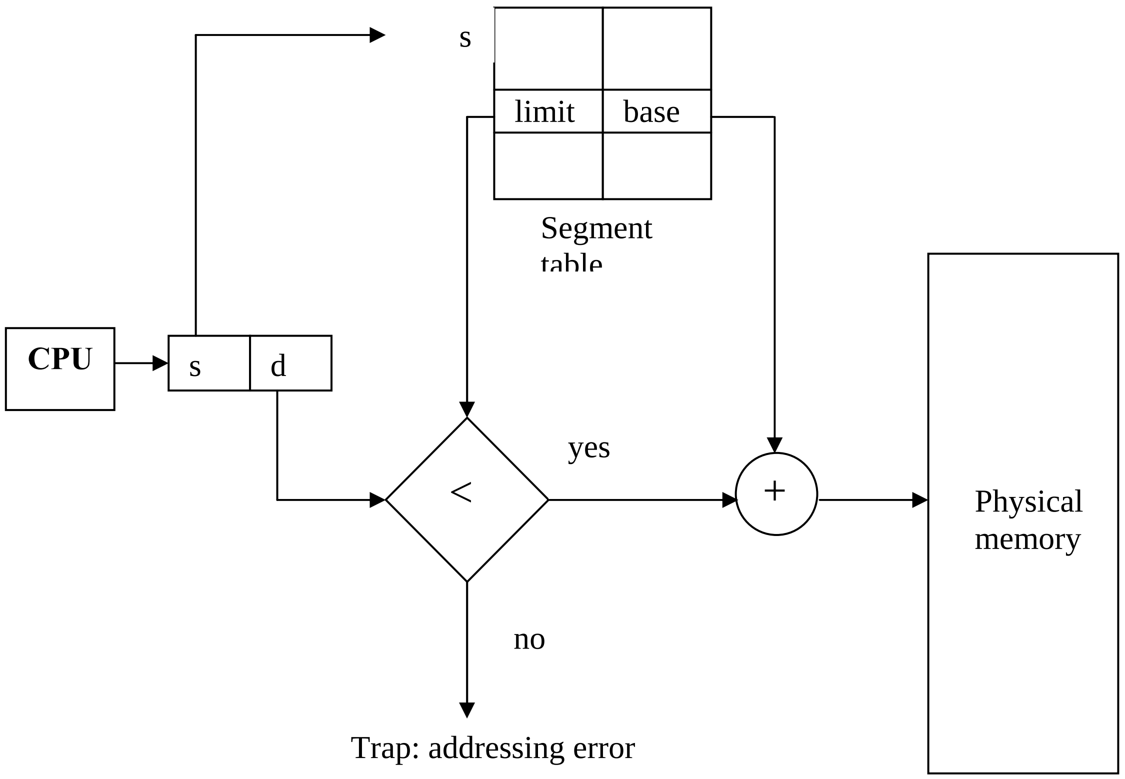
For example, the subroutine SSUB2 in segment SHARED is assumed to begin at offset 100. However, the unique designation of an item in a segmented address space requires the specification of both its segment & the relative offset therein. Offset 100 may fetch the first instruction of the subroutine SSUB2 within the segment SHARED, but the same relative offset may designate an entirely unrelated datum in the DATA segment.

To simplify processing, segment names are usually mapped to (virtual) segment numbers. This mapping is static, & systems programs in the course of preparation of process images may perform it.

A sample linker-produced load module for the segments defined in Figure 10(a) is depicted in Figure 10(b). Virtual segment numbers are shown as part of the segment map that systems programs prepare to facilitate loading of segments in memory by the OS. When segment numbers & relative offsets within the segments are defined, two-component virtual addresses uniquely identify all items within a process's address space. For example, if the SHARED segment is assigned number 3, the subroutine SSUB2 may be uniquely identified by its virtual address (3100), where 100 is the offset within the segment number 3-SHARED.

Address Translation

Since physical memory in segmented systems generally retains its linear-array organization, some address translation mechanism is needed to convert a two-dimensional virtual-segment address into its one-dimensional physical equivalent. In segmented systems, items belonging to a single segment reside in one contiguous area of physical memory. With each segment compiled as if starting from the virtual address zero, segments are generally individually relocatable. As a result, different segments of the same process need not occupy contiguous areas of physical memory.



Hardware support for Segmentation

Figure 11 – Address translation in segmented systems

When requested to load a segmented process, the OS attempts to allocate memory for the supplied segments. Using logic similar to that used for dynamic partitioning, it may create a separate partition to suit the needs of each particular segment. The base (obtained during partition creation) & size (specified in the load module) of a loaded segment are recorded as a tuple called the segment descriptor. All segment descriptors of a given process are collected in a table called the segment descriptor table (SDT). An important component of address mapping in segmented system is segment descriptor table. Two dimensional user defined address is mapped to one dimensional physical address by segment descriptor table. Each entry of this table has segment base & segment limit. Segment base contains the starting physical address of the segment & segment limit specifies the length of the segment.

Figure 11 illustrates a sample placement of the segments defined in Figure 10 into physical memory, & the resulting SDT formed by the OS. With the physical base address of each segment defined, the process of translation of a virtual, two-component address into its physical equivalent basically follows the mechanics of based addressing. The segment number provided in the virtual address is used to index the segment descriptor table & to obtain the physical base address of the related segment. Adding the offset of the desired item to the base of its enclosing segment then produces the physical address. This process is illustrated in Figure 11 for the example of the virtual address (3100). To access Segment 3, the number 3 is used to index the SDT & to obtain the physical base address, 20000, of the segment SHARED. The size field of the same segment descriptor is used to check whether the supplied offset is within the legal bounds of its enclosing segment. If so, the base & offset are added to produce the target physical address. In our example that value is 20100, the first instruction word of the shared subroutine SSUB2.

In general, the size of a segment descriptor table is related to the size of the virtual address space of a process. For example, Intel's iAPX 286 processor is capable of supporting up to 16K segments of 64 KB each per process, thus requiring 16K entries per SDT. Given their potential size, segment descriptor tables are not kept in registers. Being a collection of logically related items, the SDTs themselves are often treated as special types of segments. Their accessing is usually facilitated by means of a dedicated hardware register called the segment descriptor table base register (SDTBR), which is set to point to the base of the running process's SDT. Since the size of an SDT may vary from a few entries to several thousand, another dedicated hardware register, called the segment descriptor table limit register (SDTLR), is provided to mark the end of the SDT pointed to by the SDTBR. In this way, an SDT need contain only as many entries as there are segments actually defined in a given process. Attempts to access nonexistent segments may be detected & dealt with as nonexistent-segment exceptions.

From the OS's point of view, segmentation is essentially a multiple-base-limit version of dynamically partitioned memory. Memory is allocated in the form of variable partitions; the main difference is that one such partition is allocated to each individual segment. Bases & limits of segments belonging to a given process are collected into an SDT are normally kept in the PCB of the owner process. Upon each process switch, the SDTBR & SDTLR are loaded with the base & size, respectively, of the SDT of the new running process. In addition to the process-loading time, SDT entries may also need to be updated whenever a process is swapped out or relocated for compaction purposes. Swapping out requires invalidation of all SDT entries that describe the affected segments. When the process is swapped back in, the base fields of its segment descriptors must be updated to reflect new load addresses. For this reason, swapping out of the SDT itself is rarely useful. Instead, the SDT of the swapped-out process may be discarded, & the static segment map-such as the one depicted in Figure 10(b)-may be used for creation of an up-to-date SDT whenever the related process is loaded in memory. Compaction, when supported, requires updating of the related SDT entry for each segment moved. In such systems, some additional or revised data structures may be needed to facilitate identification of the SDT entry that describes the segment scheduled to be moved.

The price paid for segmenting the address space of a process is the overhead of storing & accessing segment descriptor tables. Mapping each virtual address requires two physical memory references for a single virtual (program) reference, as follows:

- Memory reference to access the segment descriptor in the SDT
- Memory reference to access the target item in physical memory

In other words, segmentation may cut the effective memory bandwidth in half by making the effective virtual-access time twice as long as the physical memory access time.

Segment Descriptor Caching

With performance of segmented systems so critically dependent on the duration of the address translation process, computer system designers often provide some hardware accelerators to speed the translation. Memory references expended on mapping may be avoided by keeping segment descriptors in registers. However, the potential size of an SDT & the overhead of process switching make it too costly to keep an entire SDT of the running process in registers. A reasonable compromise is to keep a few of the most frequently used segment descriptors in registers. In this way, most of the memory references may be mapped with the aid of registers. The rest may be mapped using the SDT in memory, as usual. This scheme is dependent on the OS's ability to select the proper segment descriptors for storing into registers. In order to provide the intuitive motivation for one possible implementation of systematic descriptor selection., let us investigate the types of segments referenced by the executing process.

Memory references may be functionally categorized as accesses to (i) Instructions, (ii) Data, & (iii) Stack

A typical instruction execution sequence consists of a mixture of the outline types of memory references. In fact, completion of a single stack manipulation instruction, such as a push of a datum from memory onto stack, may require all three types of references. Thus the working space of a process normally encompasses one each of code, data, & stack segments. Therefore, keeping the current code, data, & stack segment descriptors in registers may accelerate address translation. Depending on its type, a particular memory reference may then be mapped using the appropriate register. But can we know the exact type of each memory reference as the processor is making it? The answer is yes, with the proper hardware support. Namely, in most segmented machines the CPU emits a few status bits to indicate the type of each memory reference. The memory management hardware uses this information to select the appropriate mapping register.

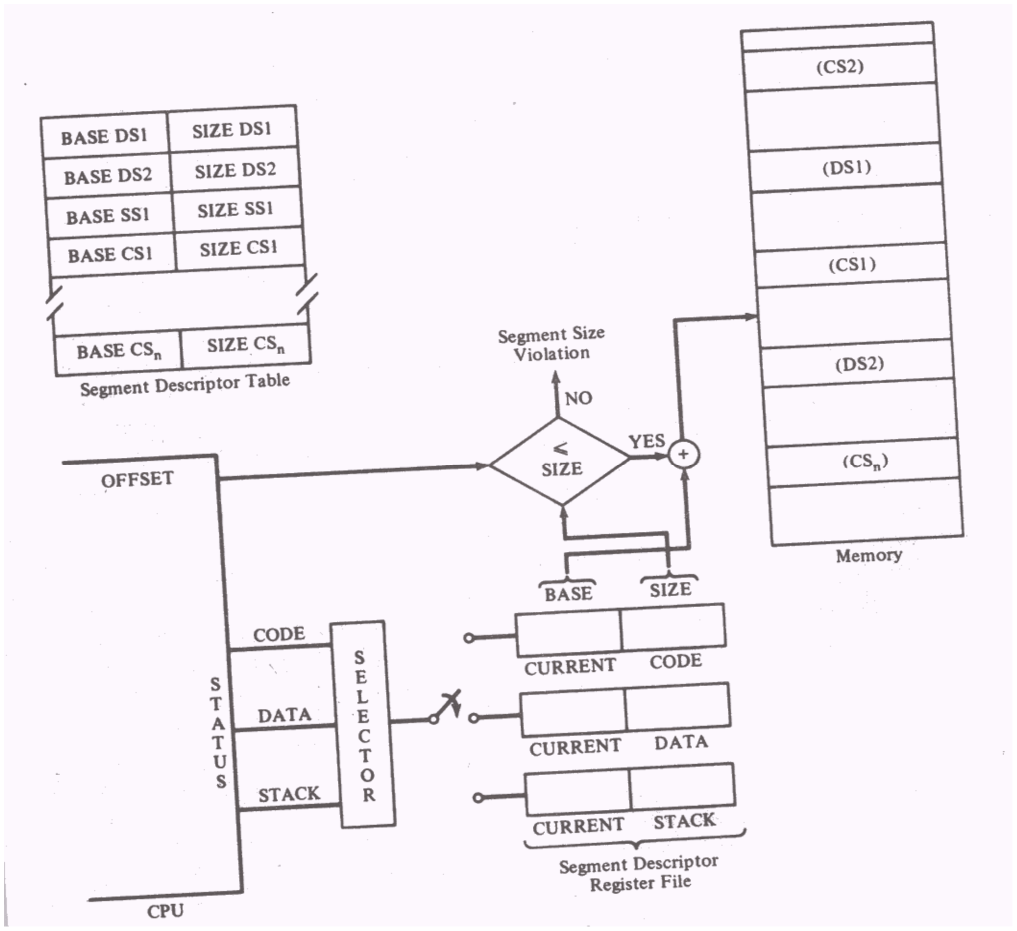


Figure 12 – Segment-descriptor cache registers

Register-assisted translation of virtual to physical addresses is illustrated in Figure 12. As indicated, the CPU status lines are used to select the appropriate segment descriptor register (SDR). The size field of the selected segment descriptor is used to check whether the intended reference is within the bounds of the target segment. If so, the base field is added with the offset to produce the physical address. By making the choice of the appropriate segment register implicit in the type of memory reference being made, segment typing may eliminate the need to keep track of segment numbers during address translations. Though segment typing is certainly useful, it may become restrictive at times. For example, copying an instruction sequence from one segment into another may confuse the selector logic into believing that source & target segments should be of type data rather than code. Using the so-called segment override of type prefixes, which allows the programmer to explicitly indicate the particular segment descriptor register to be used for mapping the memory reference in question, may alleviate this problem.

Segment descriptor registers are initially loaded from the SDT. Whenever the running process makes an intersegment reference, the corresponding segment descriptor is loaded into the appropriate register from the SDT. For example, an intersegment JUMP or CALL causes the segment descriptor of the target (code) segment to be copied from the SDT to the code segment descriptor register. When segment typing is used as described, segment descriptor register. When segment typing is used as described, segment descriptor caching becomes deterministic as opposed to probabilistic. Segment descriptors stored in the three segment descriptor registers; define the current working set of the executing process. Since membership in the working set of segments of a process changes with time, segment descriptor registers are normally included in the process state. Upon each process switch, the contents of the SDRs of the departing process are stored with the rest of its context. Before dispatching the new running process, the OS loads segment descriptor registers with their images recorded in the related PCB.

4.2.4.2 Protection

The base-limit form of protection is obviously the most natural choice for segmented systems. The legal address space of a process is the collection of segments defined by its SDT. Except for shared segments. Placing different segments in disjoint areas of memory enforces separation of distinct address space. Thus most of the discussion of protection in systems with dynamic allocation of memory is applicable to segmented environments as well.

An interesting possibility in segmented systems is to provide protection within the address space of a single process, in addition to the more usually protection between different processes. Given that the type of each segment is defined commensurate with the nature of information stored in its constituent elements, access rights to each segment can be defined accordingly. For instance, though both reading & writing of stack segments may be necessary, accessing of code segments can be permitted in execute-only or perhaps in the read-only mode. Data segments can be read-only, write-only, or read-write. Thus, segmented systems may be able to prohibit some meaningless operations, such as execution of data or modifications of code. Additional examples include prevention of stack growth into the adjacent code or data areas, & other errors resulting from mismatching of segment types & intended references to them. An important observation is that access rights to different portions of a single address space may vary in accordance with the type of information stored therein. Due to the grouping of logically related items, segmentation is one of the rare memory-management schemes that allow such finely grained delineation of access rights. The mechanism for enforcement of declared access rights in segmented systems is usually coupled with the address translation hardware. Typically, access-rights bits are included in segment descriptors. In the course of address mapping, the intended type of reference is checked against the access rights for the segment in question. Any mismatch results in abortion of the memory reference in progress, & a trap to the OS.

4.2.4.3 Sharing

Shared objects are usually placed in separate, dedicated segments. A shared segment may be mapped, via the appropriate segment descriptor tables, to the virtual-address spaces of all processes that are authorized to reference it. The deliberate use of offsets & of bases addressing facilitate sharing since the virtual offset of a given item is identical in all processes that share it. The virtual number of a shared segment, on the other hand, need not be identical in all address spaces of which it is a member. These points are illustrated in Figure 13, where a code segment EMACS is assumed to be shared by three processes. The relevant portions of the segment descriptor tables of the participating processes, SDT1, SDT2, & SDT3, are shown. As indicated, the segment EMACS is assumed to have different virtual numbers in the three address spaces of which it is part. The placement of access-rights bits in segment descriptor tables is also shown. Figure 13 illustrates the fact that different processes can have different access rights to the same shared segment. For example, whereas processes P1

& P2 can execute only the shared segment EMACS, process P3 is allowed both reading & writing.

Figure 13 also illustrates the ability of segmented systems to conserve memory by sharing the code of programs executed by many users. In particular, each participating process can execute the shared code from EMACS using its own private data segment. Assuming that EXACS is an editor, this means that a

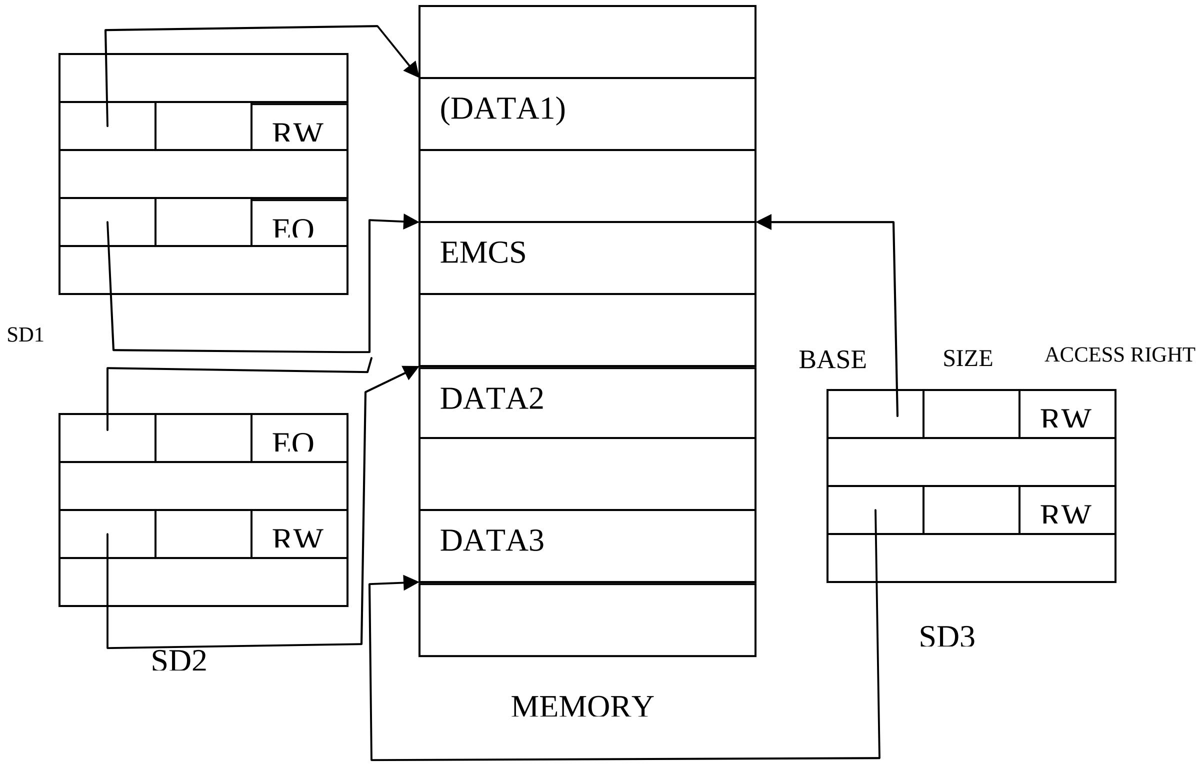


Figure 13 – Sharing in segmented systems

single copy of it may serve the entire user population of a time-sharing system. Naturally, execution of EMACS on behalf of each user is stored in a private data segment of its corresponding process. For example, users 1, 2, & 3 can have their respective texts buffers stored in data segments DATA1, DATA2, & DATA3. Depending on which of the three processes is active at a given time, the hardware data segment descriptor register points to data segment DATA1, DATA2, or DATA3, & the code segment descriptor register points to EMACS in all cases. Of course, the current instruction to be executed by the particular process is indicated by the program counter, which is saved & restored as a part of each process's state. In segmented systems, the program counter register usually contains offsets of instructions within the current code segment. This facilitates sharing by making all code self-references relative to the beginning of the current code segment. When coupled with segment typing, this feature makes it possible to assign different virtual segment numbers to the same (physical) shared segment in virtual-address spaces of different processes of which it is a part. Alternatively, the problem of making direct self-references in shared routines described in Section 3.1.3.4 of this lesson (sharing) restricts the type of code that may safely be shared.

As described, sharing is encouraged in segmented systems. This presents some problems in systems that also support swapping, which is normally done to increase processor utilization. For example, a shared segment may need to maintain its memory residence while being actively used by any of the processes authorized to reference it. Swapping in this case opens up the possibility that a participating process may be swapped out while its shared segment remains resident. When such a process is swapped back in, the construction of its SDT must take into consideration the fact that the shared segment may already be resident. In other words, the OS must keep track of shared segments & of processes that access them. When a participating process is loaded in memory, the OS is expected to identify the location of the shared segment in memory, if any, & to ensure its proper mapping from all virtual address spaces of which it is a part.

5.3 Keywords

Contiguous Memory Management: In this approach, each program occupies a single contiguous block of storage locations.

First-fit: This allocates the first available space that is big enough to accommodate process.

Best-fit: This allocates the smallest hole that is big enough to accommodate process.

Worst fit: This strategy allocates the largest hole.

External Fragmentation is waste of memory between partitions caused by scattered non-contiguous free space.

Internal fragmentation is waste of memory within a partition caused by difference between size of partition & the process allocated.

Compaction is to shuffle memory contents & place all free memory together in one block.

Program relocatability refers to the ability to load & execute a given program into an arbitrary place in memory.

4.4 SUMMARY

In this lesson, we have presented several schemes for management of main memory that are characterized by contiguous allocation of memory. Except for the single partition, which is inefficient in terms of both CPU & memory utilization, all schemes support multiprogramming by allowing address spaces of several processes to reside in main memory simultaneously. One approach is to statically divide the available physical memory into a number of fixed partitions & to satisfy requests for memory by granting suitable free partitions, if any. Fixed partition sizes limit the maximum allowable virtual-address space of any given process to the size of the largest partition (unless overlays are used). The total number of partitions in a given system limits the number of resident processes. Within the confines of this limit, the effectiveness of the short-term scheduler may be improved by employing swapping to increase the ratio of resident to ready processes. Systems with static partitioning suffer from internal fragmentation of memory.

Variable (dynamic) partitioning allows allocation of the entire physical memory, except for the resident part of the OS, to a single process. Thus, in systems with dynamic partitioning, the virtual-address space of any given process or an overlay is limited only by the capacity of the physical memory in a given system. Dynamic creation of partitions according to the specific needs of requesting processes also eliminates the problem of internal fragmentation. Dynamic allocation of partitions requires the use of more complex algorithms for de-allocation of partitions & coalescing of free memory in order to combat external fragmentation. The need for occasional compaction of memory is also a major contributor to the increased time & space complexity of dynamic partitioning.

Both fixed & variable partitioning of memory rely on virtually identical hardware support for relocation & protection. Sharing is quite restrictive in both systems.

Segmentation allows breaking of the virtual address space of a single process into separate entities (segments) that may be placed in noncontiguous areas of physical memory. As a result, the virtual-to-physical address translation at instruction execution time in such systems is more complex, & some dedicated hardware support is necessary to avoid a drastic reduction in effective memory bandwidth. Since average segment sizes are usually smaller then average process sizes, segmentation can reduce the impact of external fragmentation on the performance of systems with dynamically partitioned memory. Other advantages of segmentation include dynamic relocation, finely grained protection both within & between address spaces, ease of sharing, & facilitation of dynamic linking & loading. Unless virtual segmentation is supported, segmentation does not remove the problem of limiting the size of a process's virtual space by the size of the available physical memory.

4.5 SELF ASSESSMENT QUESTIONS (SAQ)

1. What functions does a memory manager perform?
2. How is user address space loaded in one partition of memory protected from others?
3. What is the problem of fragmentation? How is it dealt with?
4. What do you understand by program relocatability? Differentiate between static and dynamic relocation.
5. Differentiate between first fit, best fit, and worst fit memory allocation strategies. Discuss their merits and demerits.
6. How is the tracks of status of memory is kept in partitioned memory management?

7. What do you mean by relocation of address space? What problems does it cause?
8. Differentiate between internal fragmentation and external fragmentation.
9. What is external fragmentation? What is compaction? What are the merits and demerits of compaction?
10. What do you understand by segmentation? Discuss the address translation in segmented systems.
11. What are three basic approaches to sharing in systems with fixed partitioning of memory?

4.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Godbole A.S., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
5. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
6. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 6 ***Writer: Dr. Rakesh Kumar***

Memory Management - II ***Vetter: Prof. Dharminder Kr.***

6.0 OBJECTIVE

The objective of this lesson is to make the students familiar with the following concepts of (1) Non-contiguous memory management, (2) Paging, and (3) Virtual memory.

6.1 INTRODUCTION

In noncontiguous memory management memory is allocated in such a way that parts of single logical object may be placed in noncontiguous areas of physical memory, whereas the address space of a process always remains contiguous. At run time contiguous virtual address space is mapped to noncontiguous physical address space. This type of memory management is done in various ways:

4. Non-Contiguous, real memory management system
   - Paged memory management system
   - Segmented memory management system
   - Combined memory management system
5. Non-Contiguous, virtual memory management system
   - Virtual memory management system

6.2 Presentation of contents

6.2.1 Paging

6.2.1.1 Principles of Operation

6.2.1.2 Page Allocation

6.2.1.3 Hardware Support for Paging

6.2.1.4 Protection & Sharing

6.2.2 Virtual Memory

6.2.2.1 Principles of Operation

6.2.2.2 Management of Virtual Memory

6.2.2.4 Program Behavior

6.2.2.5 Replacement Policies

6.2.2.5.1 Replacement Algorithms
6.2.2.5.2 Global & Local Replacement Policies
6.2.2.6 Allocation Policies
6.2.2.7 Hardware Support & Considerations
6.2.2.8 Protection & Sharing
6.2.3 Segmentation & Paging

6.2.1 PAGING

Paging is another solution to the problem of memory fragmentation. It removes the requirement of contiguous allocation of physical memory. In it, the physical memory is conceptually divided into a number of fixed-size slots, called page frames. The virtual-address space of a process is also split into fixed-size blocks of the same size, called pages. Memory management module identifies sufficient number of unused page frames for loading of the requesting process's pages. An address translation mechanism is used to map virtual pages to their physical counterparts. Since each page is mapped separately, different page frames allocated to a single process need not occupy contiguous areas of physical memory.

6.2.1.1 Principles of Operation

Figure 1 demonstrates the basic principle of paging. It illustrates a sample 16 MB system where virtual & physical addresses are assumed to be 24 bits long each. The page size is assumed to be 4096 bytes. Thus, the physical memory can accommodate 4096 page frames of 4096 bytes each. After reserving 1 MB of physical memory for the resident portion of the OS, the remaining 3840 page frames are available for allocation to user processes. The addresses are given in hexadecimal notation. Each page is 1000H bytes long, & the first user-allocatable page frame starts at the physical address 100000H.

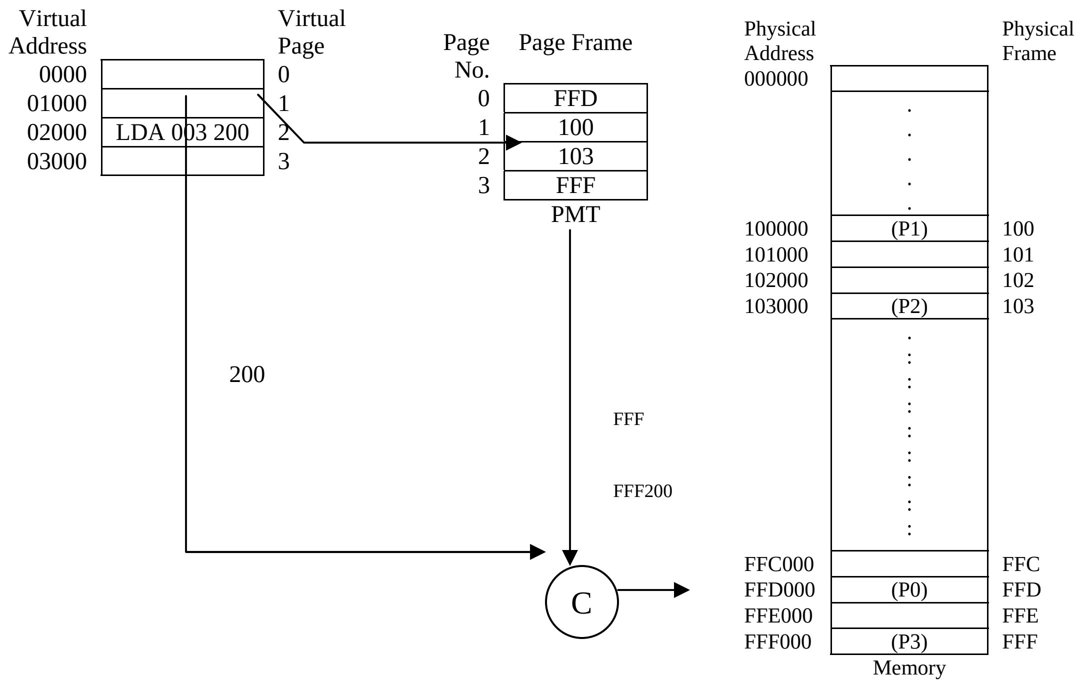


Figure 1 - Paging

The virtual-address space of a sample user process that is 14,848 bytes (3A00H) long is divided into four virtual pages numbered from 0 to 3. A possible placement of those pages into physical memory is depicted in Figure 1. The mapping of virtual addresses to physical addresses in paging systems is performed at the page level. Each virtual address is divided into two parts: the page number & the offset within that page. Since pages & page frames have identical sizes, offsets within each are identical & need not be mapped. So each 24-bit virtual address consist of a 12-bit page number (high-order bits) & a 12-bit offset within the page.

Address translation is performed with the help of the page-map table (PMT), constructed at process-loading time. As indicated in figure 1, there is one PMT entry for each virtual page of a process. The value of each entry is the number of the page frame in the physical memory where the corresponding virtual page is placed. Since offsets are not mapped, only the page frame number need be stored in a PMT entry. E.g., virtual page 0 is assumed to be placed in the physical page frame whose starting address is FFD000H (16,764,928 decimal). With each frame being 1000H bytes long, the corresponding page frame number is FFDH, as indicated on the right-hand side of the physical memory layout in Figure 1. This value is stored in the first entry of the PMT. All other PMT entries are filled with page frame numbers of the region where the corresponding pages are actually loaded.

The logic of the address translation process in paged systems is illustrated in Figure 1 on the example of the virtual address 03200H. The virtual address is split by hardware into the page number 003H, & the offset within that page (200H). The page number is used to index the PMT & to obtain the corresponding physical frame number, i.e. FFF. This value is then concatenated with the offset to produce the physical address, FFF200H, which is used to reference the target item in memory.

000	ALLOCATED
	: : :
0FF	ALLOCATED
100	ALLOCATED
101	FREE
102	FREE
103	ALLOCATED
	: : :
FFC	FREE
FFD	ALLOCATED
FFE	FREE
FFF	ALLOCATED

Figure 2 – Memory-map table (MMT)

The OS keeps track of the status of each page frame by using a memory-map table (MMT). Format of an MMT is illustrated in Figure 2, assuming that only the process depicted in Figure 1 & the OS are resident in memory. Each entry of the MMT described the status of page frame as FREE or ALLOCATED. The number of MMT entries i.e. f is computed as f = m/p where m is the size of the physical memory, & p is page size. Both m & p are usually an integer power of base 2, thus resulting in f being an integer. When requested to load a process of size s, the OS must allocate n free page frames, so that n = Round(s/p) where p is the page size. The OS allocates memory in terms of an integral number of page frames. If the size of a given process is not a multiple of the page size, the last page frame may be partly unused resulting into page fragmentation.

After selecting n free page frames, the OS loads process pages into them & constructs the page-map table of the process. Thus, there is one MMT per system, & as many PMTs as there are active processes. When a process terminates or becomes swapped out, memory is deallocated by releasing the frame holdings of the departing process to the pool of free page frames.

6.2.1.2 Page Allocation

The efficiency of the memory allocation algorithm depends on the speed with which it can locate free page frames. To facilitate this, a list of free pages is maintained instead of the static-table format of the memory map assumed earlier. In that case, n free frames may be identified & allocated by unlinking the first n nodes of the free list. Deallocation of memory in systems without the free list consists of marking in the MMT as FREE all frames found in the PMT of the departing process a time consuming operation. Frames identified in the PMT of the departing process can be linked to the beginning of the freed list. Linking at the beginning is the fastest way of adding entries to an unordered singly linked list. Since the time complexity of deallocation is not significantly affected by the choice of data structure of free pages, the free-list approach has a performance advantage as its time complexity of deallocation is not significantly affected by the choice of data structure of free pages, & is not affected by the variation of memory utilization.

6.2.1.3 Hardware Support for Paging

Hardware support for paging, concentrates on saving the memory necessary for storing of the mapping tables, & on speeding up the mapping of virtual to physical addresses. In principle, each PMT must be large enough to accommodate the maximum size allowed for the address space of a process in a given system. In theory, this may be the entire physical memory. So in a 16 MB system with 256-byte pages, the size of a PMT should be 64k entries. Individual PMT entries are page numbers that are 16 bits long in the sample system, thus requiring 128 KB of physical memory to store a PMT. With one PMT needed for each active process, the total PMT storage can consume a significant portion of physical memory.

Since the actual address space of a process may be well below its allowable maximum, it is reasonable to construct each PMT with only as many entries as its related process has pages. This may be accomplished by means of a dedicated hardware page-map table limit register (PMTLR). A PMTLR is set to the highest virtual page number defined in the PMT of the running process. Accessing of the PMT of the running process may be facilitated by means of the page-map table base register (PMTBR), which points to the base address of the PMT of the running process. The respective values of these two registers for each process are defined at process-loading time & stored in the related PCB. Upon each process switch, the PCB of the new running process provides the values to be loaded in the PMTBR & PMTLR registers.

Even with the assistance of these registers, address translations in paging systems still require two memory references; one to access the PMT for mapping, & the other to reference the target item in physical memory. To speed it up, a high-speed associative memory for storing a subset of often-used page-map table entries is used. This memory is called the translation look aside buffer (TLB), or mapping cache.

Associative memories can be searched by contents rather than by address. So, the main-memory reference for mapping can be substituted by a TLB reference. Given that the TLB cycle time is very small, the memory-access overhead incurred by mapping can be significantly reduced. The role of the cache in the mapping process is depicted in Figure 3.

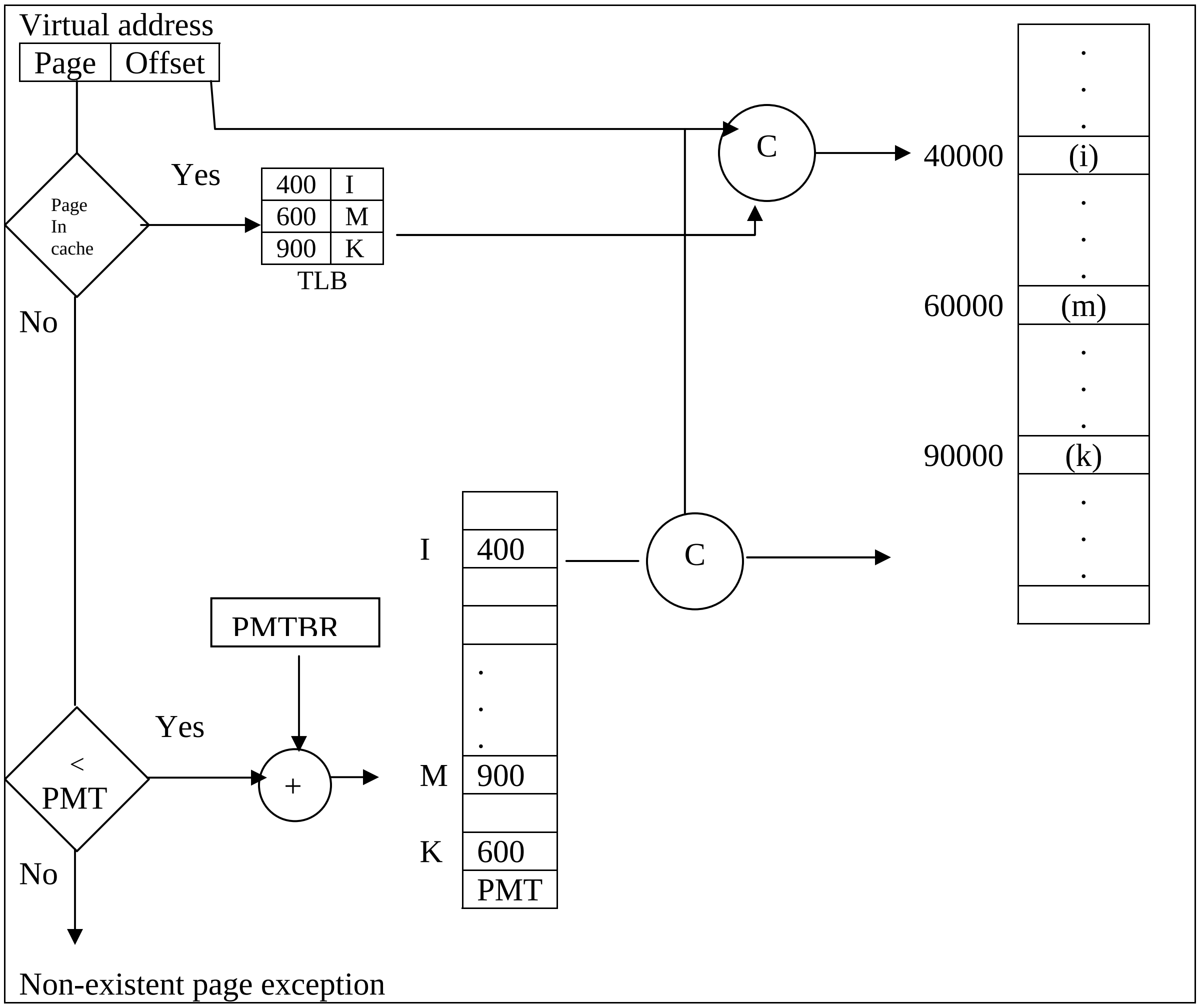


Figure 3 – Translation-lookaside buffer (TLB)

As indicated, the TLB entries contain pairs of virtual page numbers & the corresponding page frame numbers where the related pages are stored in physical memory. The page number is necessary to define each particular entry, because a TLB contains only a subset of page-map table entries. Address translation begins by presenting the page-number portion of the virtual address to the TLB. If the desired entry is found in the TLB, the corresponding page frame number is combined with the offset to produce the physical address.

Alternatively, if the target entry is not in TLB, the PMT in memory must be accessed to complete the mapping. This process begins by consulting the PMTLR to verify that the page number provided in the virtual address is within the bounds of the related process's address space. If so, the page number is added to the contents of the PMTBR to obtain the address of the corresponding

PMT entry where the physical page frame number is stored. This value is then concatenated with the offset portion of the virtual address to produce the physical memory address of the desired item.

Figure 3 demonstrates that the overhead of TLB search is added to all mappings, regardless of whether they are eventually completed using the TLB or the PMT in main memory. In order for the TLB to be effective, it must satisfy a large portion of all address mappings. Given the generally small size of a TLB because of the high price of associative memories, only the PMT entries most likely to be needed ought to reside in the TLB.

The effective memory-access time, t_{eff} in systems with run-time address translation is the sum of the address translation time, t_{TR} & the subsequent access time needed to fetch the target item from memory, t_M.

$$t_{eff} = t_{TR} + t_M$$

With TLB used to assist in address translation, t_{TR} becomes

$$T_{TR} = h\, t_{TLB} + (1 - h)\,(t_{TLB} + t_M) = t_{TLB} + (1 - h)t_M$$

where h is the TLB hit ratio, that is, the ratio of address translations that are contained the TLB over all translations, & thus $0 \leq h \leq 1$;t_{TLB} is the TLB access time; & t_M is the main-memory access time. Therefore, effective memory-access time in systems with a TLB is

$$t_{eff} = t_{TLB} + (2 - h)t_M$$

It is observed that the hardware used to accelerate address translations in paging systems (TLB) is managed by means of probabilistic algorithms, as opposed to the deterministic mapping-register typing described in relation to segmentation. The reason is that the mechanical splitting of a process's address space into fixed-size chunks produces pages. As a result, a page, unlike a segment, in general does not bear any relationship to the logical entities of the underlying program. For example, a single page may contain a mixture of data, stack, & code. This makes typing & other forms of deterministic loading of TLB entries extremely difficult, in view of the stringent timing restrictions imposed on TLB manipulation.

6.2.1.4 Protection & Sharing

Unless specifically declared as shared, distinct address spaces are placed in disjoint areas of physical memory. Memory references of the running process are restricted to its own address space by means of the address translation mechanism, which uses the dedicated PMT. The PMTLR is used to detect & to abort attempts to access any memory beyond the legal boundaries of a process. Modifications of the PMTBR & PMTLR registers are usually possible only by means of privileged instructions, which trap to the OS if attempted in user mode.

By adding the access bits to the PMT entries & appropriate hardware for testing these bits, access to a given page may be allowed only in certain programmer-defined modes such as read-only, execute-only, or other restricted forms of access. This feature is much less flexible in paging systems than segmentation. The primary difference is that paging is supposed to be entirely transparent to programmers. Mechanical splitting of an address space into pages is performed without any regard for the possible logical relationships between the items under consideration. Since there is no notion of typing, code & data may be mixed within one page. As we shall see, specification of the access rights in paging systems is useful for pages shared by several processes, but it is of much less value inside the boundaries of a given address space.

Protection in paging systems may also be accomplished by means of the protection keys. In principle, the page size should correspond to the size of the memory block protected by the single key. This allows pages belonging to a single process to be scattered throughout memory-a perfect match for paged allocation. By associating access-rights bits with protection keys, access to a given page may be restricted when necessary.

Sharing of pages is quite straightforward with paged memory management. A single physical copy of a shared page can be easily mapped into as many distinct address spaces as desired. Since each such mapping is performed via a dedicated entry in the PMT of the related process, different processes may have different access rights to the shared page. Given that paging is transparent to users, sharing at the page level must be recognized & supported by systems programs. Systems programs must ensure that virtual offsets of each item within a shared page are identical in all participating address spaces.

Like data, shared code must have the same within-page offsets in all address spaces of which it is a part. As usual, shared code that is not executed in mutually exclusive fashion must be reentrant. In addition, unless the shared code is position-independent, it must have the same virtual page numbers in all processes that invoke it. This property must be preserved even in cases when the shared code spans several pages.

6.2.2 VIRTUAL MEMORY

Virtual memory allows execution of partially loaded processes. As a consequence, virtual address spaces of active processes in a virtual-memory system can exceed the capacity of the physical memory. This is accomplished by maintaining an image of the entire virtual-address space of a process on secondary storage, & by bringing its sections into main memory when needed. The OS decides which sections to bring in, when to bring them in, & where to place them. Thus, virtual-memory systems provide for automatic migration of portions of address spaces between secondary & primary storage. Virtual memory provides the illusion of a much larger memory than may actually be available, so programmers are relieved of the burden of trying to fit a program into limited memory.

Due to the ability to execute a partially loaded process, a process may be loaded into a space of arbitrary size resulting into the reduction of external fragmentation. Moreover, the amount of space in use by a given process may be varied during its memory residence. As a result, the OS may speed up the execution of important processes by allocating them more real memory. Alternatively, by reducing the real-memory holdings of resident processes, the degree of multi-programming can be increased by using the vacated space to activate more processes.

The speed of program execution in virtual-memory systems is bounded from above by the execution speed of the same program run in a non-virtual memory management system. That is due to delays caused by fetching of missing portions of program's address space at run-time.

Virtual memory provides execution of partially loaded programs. But an instruction can be completed only if all code, data, & stack locations that it references reside in physical memory. When we reference an out-of-memory item, the running process must be suspended to fetch the target item from disk. So what is the performance penalty?

An analysis of program behavior provides an answer to the question. Most programs consist of alternate execution paths, some of which do not span the entire address space. On any given run, external & internal program conditions cause only one specific execution path to be followed. Dynamic linking & loading exploits this aspect of program behavior by loading into memory only those procedures that are actually referenced on a particular run. Moreover, many programs tend to favor specific portions of their address spaces during execution. So it is reasonable to keep in memory only those routines that make up the code of the current pass. When another pass over the source code commences, the memory manager can bring new routines into the main memory & return those of the previous pass back to disk.

6.2.2.1 Principles of Operation

Virtual memory can be implemented as an extension of paged or segmented memory management or as a combination of both. Accordingly, address translation is performed by means of PMT, segment descriptor tables, or both. The important characteristic is that in virtual-memory systems some portions of the address space of the running process can be absent from main memory.

The process of address mapping in virtual-memory systems is more formally defined as follows. Let the virtual-address space be $V = \{0, 1, \dots, v-1\}$, & the physical memory space by $M = \{0, 1, \dots m-1\}$. The OS dynamically allocates real memory to portions of the virtual-address space. The address translation mechanism must be able to associate virtual names with physical locations. In other words, at any time the mapping hardware must realize the function $f: V \rightarrow M$ such that

$$f(x) = \begin{cases} r \text{ if item } x \text{ is in real memory at location } r \\ \\ \text{missing-item exception if item } x \text{ is not in real memory} \end{cases}$$

Thus, the additional task of address translation hardware in virtual systems is to detect whether the target item is in real memory or not. If the referenced item is in memory, the process of address translation is completed.

We present the operation of virtual memory assuming that paging is the basic underlying memory-management scheme. The detection of missing items is rather straightforward. It is usually handled by adding the presence indicator, a bit, to each entry of page-map tables. The presence bit, when set, indicates that the corresponding page is in memory; otherwise the corresponding virtual page is not in real memory. Before loading the process, the OS clears all the presence bits in the related page-map table. As & when specific pages are brought into the main memory, its presence bit is reset.

A possible implementation is illustrated in Figure 4. The presented process's virtual space is assumed to consisting of only six pages. As indicated, the complete process image is present in secondary memory. The PMT contains an entry for each virtual page of the related process. For each page actually present in real memory, the presence bit is set (IN), & the PMT points to the physical frame that contains the corresponding page. Alternatively, the presence bit is cleared (OUT), & the PMT entry is invalid.

The address translation hardware checks the presence bit during the mapping of each memory reference if the bit is set, the mapping is completed as usual. However, if the corresponding presence bit in the PMT is reset, the hardware generates a missing-item exception. In paged virtual-memory systems, this exception is called a page fault. When the running process experiences a page fault, it must be suspended until the missing page is brought into main memory.

The disk address of the faulted page is usually provided in the file-map table (FMT). This table is parallel to the PMT. Thus, when processing a page fault, the OS uses the virtual page number provided by the mapping hardware to index the

FMT & to obtain the related disk address. A possible format & use of the FMT is depicted in Figure 4.

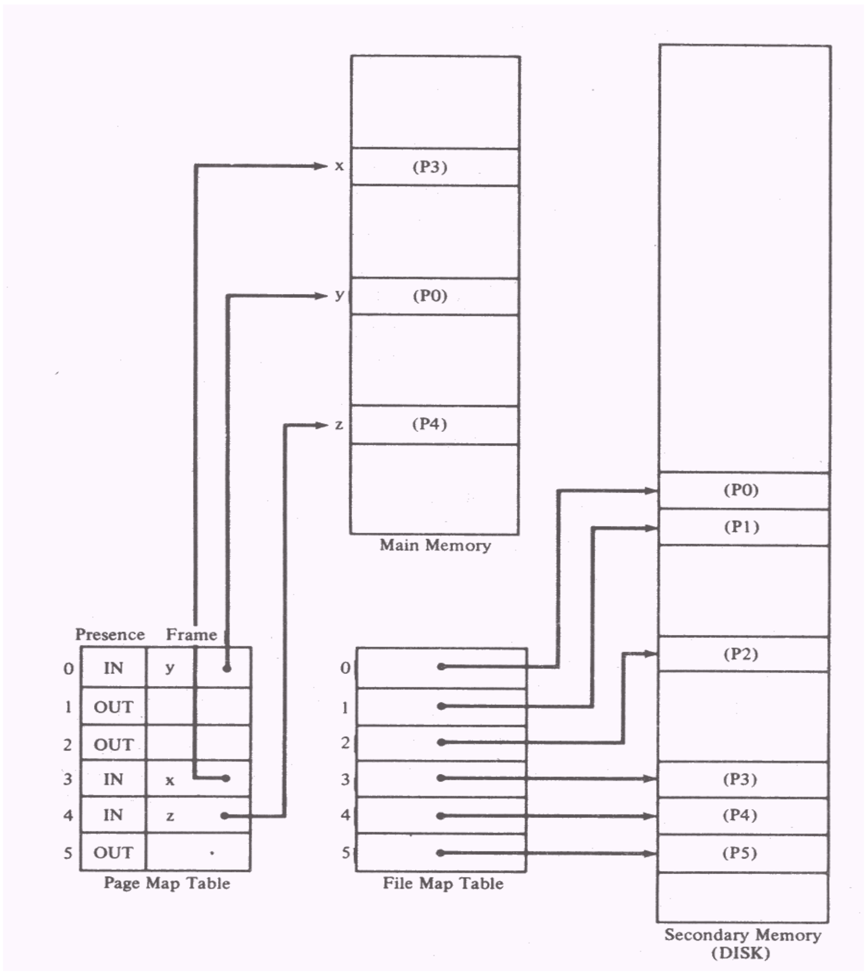


Figure 4 – Virtual memory

6.2.2.2 Management of Virtual Memory

The implementation of virtual memory requires maintenance of one PMT per active process. Given that the virtual-address space of a process may exceed the capacity of real memory, the size of an individual PMT can be much larger in a virtual than in a real paging system with identical page sizes. The OS maintains one MMT or a free-frame list to keep track of Free/allocated page frames.

A new component of the memory manager's data structures is the FMT. FMT contains secondary-storage addresses of all pages. The memory manager used the FMT to load the missing items into the main memory. One FMT is maintained for each active process. Its base may be kept in the control block of the related process. An FMT has a number of entries identical to that of the related PMT. A pair of page-map table base & page-map length registers may be provided in hardware to expedite the address translation process & to reduce the size of PMT for smaller processes. As with paging, the existence of a TLB is highly desirable to reduce the negative effects of mapping on the effective memory bandwidth.

The allocation of only a subset of real page frames to the virtual-address space of a process requires the incorporation of certain policies into the virtual-memory manager. We may classify these policies as follows:

1. Allocation policy: How much real memory to allocate to each active process
2. Fetch policy: Which items to bring & when to bring them from secondary storage into the main memory
3. Replacement policy: When a new item is to be brought in & there is no free real memory, which item to evict in order to make room.
4. Placement policy: Where to place an incoming item

6.2.2.4 Program Behavior

Program behavior is of extreme importance for the performance of virtual-memory systems. Execution of partially loaded programs generally leads to longer turnaround times due to the processing of page faults. By minimizing the number of page faults, the effective processor utilization, effective disk I/O bandwidth, & program turnaround times may be improved.

It is observed that there is a strong tendency of programs to favor subsets of their address spaces during execution. This phenomenon is known as locality of reference. Both temporal & spatial locality of reference has been observed. Spatial locality is the tendency for a program to reference clustered locations in preference to randomly distributed locations. Temporal locality is the tendency for a program to reference the same location or a cluster several times during brief intervals of time. Temporal locality of reference is exhibited by program loops. Spatial locality suggests that once an item is referenced, there is a high probability that it or its neighboring items are going to be referenced in the near future. A locality is a small cluster of not necessarily adjacent pages to which most memory references are made during a period of time. Both temporal & spatial locality of reference is dynamic properties in the sense that the identity of the particular pages that compose the actively used set varies with time. As observed, the executing program moves from one locality to another in the course of its execution. Statistically speaking, the probability that a particular memory reference is going to be made to a specific page is a time-varying function. It increases when pages in its current locality are being referenced, & it decreases otherwise. The evidence also suggests that the executing program moves slowly from one locality to another.

Locality of reference basically suggests that a significant portion of memory references of the running process may be made to a subset of its pages. These findings may be utilized for implementation of replacement & allocation policies.

6.2.2.5 Replacement Policies

If a page fault is there, then it is to be brought into the main memory necessitating creation of a room for it. There are two options for this situation:

- The faulted process may be suspended until availability of memory.
- A page may be removed to make room for the incoming one.

Suspending a process is not an acceptable solution. Thus, removal is commonly used to free the memory needed to load the missing items. A replacement policy decides the victim page for eviction. In virtual memory systems all pages are kept on the secondary storage. As & when needed, some of those pages are copied into the main memory. While executing, the running process may modify its data or stack areas, thus making some resident pages different from their disk images (dirty page). So it must be written back to disk in place of its obsolete copy. When a page that has not been modified (clean page) during its residence in memory is to be evicted, if can simply be discarded. Tracking of page modifications is usually performed in hardware by adding a written-into bit called as dirty bit, to each entry of the PMT. It indicates whether the page is dirty or clean.

6.2.2.5.1 Replacement Algorithms

First-In-First-Out (FIFO). The FIFO algorithm replaces oldest pages i.e. the resident page that has spent the longest time in memory. To implement the FIFO page-replacement algorithm, the memory manager must keep track of the relative order of the loading of pages into the main memory. One way to accomplish this is to maintain a FIFO queue of pages.

FIFO fails to take into account the pattern of usage of a given page; FIFO tends to throw away frequently used pages because they naturally tend to stay longer in memory. Another problem with FIFO is that it may defy intuition by increasing the number of page faults when more real pages are allocated to the program. This behavior is known as Belady's anomaly.

Least Recently Used (LRU). The least recently used algorithm replaces the least recently used resident page. LRU algorithm performs better than FIFO because it takes into account the patterns of program behavior by assuming that the page used in the most distant past is least likely to be referenced in the near future. The LRU algorithm belongs to a larger class of stack replacement algorithms. A stack algorithm is distinguished by the property of performing better, or at least not worse, when more real memory is made available to the executing program. Stack algorithms therefore do not suffer from Belady's anomaly.

The implementation of the LRU algorithm imposes too much overhead to be handled by software alone. One possible implementation is to record the usage of pages by means of a structure similar to the stack. Whenever a resident page is referenced, it is removed from its current stack position & placed at the top of the stack. When a page eviction is in order, the page at the bottom of the stack is removed from memory.

Maintenance of the page-referencing stack requires it's updating for each page reference, regardless of whether it results in a page fault or not. So the overhead of searching the stack, moving the reference page to the top, & updating the rest of the stack accordingly must be added to all memory references. But the FIFO queue needs to be updated only when page faults occur-overhead almost negligible in comparison to the time required for processing of a page fault.

Optimal (OPT): The algorithm by Belady, removes the page to be reference in the most distant future. Since it requires future knowledge, the OPT algorithm is not realizable. Its significance is theoretical, as it can serve as a yardstick for comparison with other algorithms.

Approximations-Clock. One popular algorithm combines the relatively low overhead of FIFO with tracking of the resident-page usage, which accounts for the better performance of LRU. This algorithm is sometimes referred to as Clock, & it is also known as not recently used (NRU).

The algorithm makes use of the referenced bit, which is associated with each resident page. The referenced bit is set whenever the related page is reference & cleared occasionally by software. Its setting indicates whether a given page has been referenced in the recent past. How recent this past is depends on the frequency of the referenced-bit resetting. The page-replacement routine makes use of this information when selecting a victim for removal.

The algorithm is usually implemented by maintaining a circular list of the resident pages & a pointer to the page where it left off. The algorithm works by sweeping the page list & resetting the presence bit of the pages that it encounters. This sweeping motion of the circular list resembles the movement of the clock hand, hence the name clock. The clock algorithm seeks & evicts pages not recently used in order to free page frames for allocation to demanding processes. When it encounters a page whose reference bit is cleared, which means that the related page has not been referenced since the last sweep, the algorithm acts as follows: (1) If the page is modified, it is marked for clearing & scheduled for writing to disk. (2) If the page is not modified, it is declared non-resident, & the page frames that it occupies are feed. The algorithm continues its operation until the required numbers of page frames are freed. The algorithm may be invoked at regular time intervals or when the number of free page frames drops below a specified threshold.

Other approximations & variations on this theme are possible. Some of them track page usage more accurately by means of a reference counter that counts the number of sweeps during which a given page is found to be un-referenced. Another possibility is to record the states of referenced bits by shifting them occasionally into related bit arrays. When a page is to be evicted, the victim is chosen by comparing counters or bit arrays in order to find the least frequently reference page. The general idea is to devise an implementable algorithm that bases its decisions on measured page usage & thus takes into account the program behavior patterns.

6.2.2.5.2 Global & Local Replacement Policies

As discussed, all replacement policies choose a victim among the resident pages owned by the process that experiences the page fault. This is known as local replacement. However, each of the presented algorithms may be made to operate globally. A global replacement algorithm processes all resident pages when selecting a victim. Local replacement tends to localize effects of the allocation policy to each particular process. Global replacement, on the other hand, increases the degree of coupling between replacement & allocation strategies. A global replacement algorithm may take pages allocated to one process by the allocation algorithm, away. Global replacement is concerned mostly with the overall state of the system, & much less with the behavior of each individual process. Global replacement is known to be sub-optimal.

6.2.2.6 Allocation Policies

The allocation policy must compromise among conflicting requirements such as reduced page-fault frequency, improved turn-around time and processor utilization, etc. Giving more real pages to a process will result in reduced page-fault frequency & improved turnaround time. But it reduces the number of active processes that may coexist in memory at a time resulting into the lower processor utilization factor. On the other hand, if too few pages are allocated to a process, its page-fault frequency & turnaround times may deteriorate.

Another problem caused by under-allocation of real pages may be encountered in systems that opt for restarting of faulted instructions. If fewer pages are allocated to a process than are necessary for execution of the restartable instruction that causes the largest number of page faults in a given architecture, the system might fault continuously on a single instruction & fail to make any real progress.

Consider a two-address instruction, such as Add @X, @Y, where X & Y are virtual addresses & @ denotes indirect addressing. Assuming that the operation code & operand addresses are encoded in one word each, this instruction needs three words for storage. With the use of indirect addressing, eight memory references are needed to complete execution of this instruction: three to fetch the instruction words, two to fetch operand addresses, two to access the operands themselves (indirect addressing), & one to store the result. In the worst case, six different pages may have to reside in memory concurrently in order to complete execution of this instruction: two if the instruction crosses a page boundary, two holding indirect addresses, & two holding the target operands. A likely implementation of this instruction calls for the instruction to be restarted after a page fault. If so, with fewer than six pages allocated to the process that executes it, the instruction may keep faulting forever. In general, the lower limit on the number of pages imposed by the described problem is architecture-dependent. In any particular implementation, the appropriate bound must be evaluated & built into the logic of the allocation routine.

While we seem to have some guidance as to the minimal number of pages, the reasonable maximum remains elusive. It is also unclear whether a page maximum should be fixed for a given system or determined on an individual basis according to some specific process attributes. Should the maximum be defined statically or dynamically, in response to system resource utilization & availability, & perhaps in accordance with the observable behavior of the specific process?

From the allocation module's point of view, the important conclusion is that each program has a certain threshold regarding the proportion of real to virtual pages, below which the number of page faults increases very quickly. At the high end, there seems to be a certain limit on the number of real pages, above which an allocation of additional real memory results in little or in moderate performance improvement. Thus, we want to allocate memory in such a way that each active program is between these two extremes.

Being program-specific, the upper & lower limits should probably not be fixed but derived dynamically on the basis of the program faulting behavior measured during its execution. When resource utilization is low, activating more processes may increase the degree of multiprogramming. However, the memory manager must keep track of the program behavior when doing so. A process that experiences a large number of page faults should be either allocated more memory or suspended otherwise. Likewise, a few pages may be taken away from a process with a low page-fault rate without great concern. In addition, the number of pages allocated to a process may be influenced by its priority (higher priority may indicate that shorter turnaround time is desirable), the amount of free memory, fairness, & the like.

Although the complexity & overhead of memory allocation should be within a reasonable bound, the use of oversimplified allocation algorithms has the potential of crippling the system throughput. If real memory is over-allocated to the extent that most of the active programs are above their upper page-fault-rate thresholds, the system may exhibit a behavior known as thrashing. With very frequent page faults, the system spends most of its time shuttling pages between main memory & secondary memory. Although the disk I/O channel may be overloaded by this activity, but processor utilization is reduced.

One way of introducing thrashing behavior is dangerously logical & simple. After observing a low processor utilization factor, the OS may attempt to improve it by activating more processes. if no free pages are available, the holdings of the already-active processes may be reduced. This may drive some of the processes into the high page-fault zone. As a result, the processor utilization may drop while the processes are awaiting their pages to be brought in. In order to improve the still-decreasing processor utilization, the OS may decide to increase the degree of multi-programming even further. Still more pages will be taken away from the already-depleted holdings of the active processes, & the system is hopelessly on its way to thrashing. It is obvious that global replacement strategies are susceptible to thrashing.

Thus a good design must make sure that the allocation algorithm is not unstable & inclined toward thrashing. Knowing the typical patterns of program behavior, we want to ensure that no process is allocated too few pages for its current needs. Too few pages may lead to thrashing, & too many pages may unduly restrict the degree of multi-programming & processor utilization.

Page-Fault Frequency (PFF)

This policy uses an upper & lower page-fault frequency threshold to decide for allocation of new page frames. The PFF parameter P may be defined as: P = 1/T Where T is the critical inter-page fault time. P is usually measured in number of page faults per millisecond. The PFF algorithm may be implemented as follows:

1. The OS defines a system-wide (or per-process) critical page-fault frequency, P.
2. The OS measures the virtual (process) time & stores the time of the most recent page fault in the related process control block.

When a page fault occurs, the OS acts as follows:

- If the last page fault occurred less than T = 1/P ms ago, the process is operating above the PFF threshold, & a new page frame is added from the pool to house the needed page.
- Otherwise, the process is operating below the PFF threshold P, & a page frame occupied by a page whose reference bit & written-into bit are not set is freed to accommodate the new page.
- The OS sweeps & resets referenced bits of all resident pages. Pages that are found to be unused, unmodified, & not shared since the last sweep are released, & the freed page frames are returned to the pool for future allocations.

For completeness, some policies need to be employed for process activation & deactivation to maintain the size of the pool of free page frames within desired limits.

6.2.2.7 Hardware Support & Considerations

Virtual memory requires: (1) instruction interruptibility & restartability, (2) a collection of page status bits associated with each page descriptor, and-if based on paging-(3) a TLB to accelerate address translations. Choice of the page size is an important design consideration in that it can have a significant impact on performance of a virtual-memory system. In most implementations, one each of the following bits is provided in every page descriptor:

- Presence bit, used to aid detection of missing items by the mapping hardware
- Written-into (modified) bit, used to reduce the overhead incurred by the writing of unmodified replaced pages to disk
- Referenced bit, used to aid implementation of the replacement policy

An important hardware accelerator in virtual-memory systems is the TLB. Although system architects & hardware designers primarily determine the details of the TLB operation, the management of TLB is of interest because it deals with problems quite similar to those discussed in the more general framework of virtual memory. TLB hardware must incorporate allocation & replacement policies so as to make the best use of the limited number of mapping entries that the TLB can hold. An issue in TLB allocation is whether to devote all TLB entries to the running process or to distribute them somehow among the set of active processes. The TLB replacement policy governs the choice of the entry to be evicted when a miss occurs & another entry needs to be brought in.

Allocation of all TLB entries to the running process can lead to relatively lengthy initial periods of “loading” the TLB whenever a process is scheduled. This can lead to the undesirable behavior observed in some systems when an interrupt service routine (ISR) preempts the running process. Since a typical ISR is only a few hundred instructions long, it may not have enough time to load the TLB. This can result in slower execution of the interrupt service routine due to the need to reference PMT in memory while performing address translations. Moreover, when the interrupted process is resumed, its performance also suffers from having to load the TLB all over again. One way to combat this problem is to use multi-context TLBs that can contain & independently manage the PMT entries of several processes. With a multi-context TLB, when a process is scheduled for execution, it may find some of its PMT entries left over in the TLB from the preceding period of activity. Management of such TLBs requires the identity of the corresponding process to be associated with each entry, in order to make sure that matches are made only with the TLB entries belonging to the process that produced the addresses to be mapped.

Removal of TLB entries is usually done after each miss. If PMT entries of several processes are in the buffer, the victim may be chosen either locally or globally. Understandably, some preferential treatment is usually given to holdings of the running process. In either case, least recently used is a popular strategy for replacement of entries.

The problem of maintaining consistency between the PMT entries & their TLB copies in the presence of frequent page moves must also be tackled by hardware designers. Its solution usually relies on some specialized control instructions for TLB flushing or for it selective invalidation.

Another hardware-related design consideration in virtual-memory systems is whether I/O devices should operate with real or virtual addresses.

A hardware/software consideration involved in the design of paged systems Is the choice of the page size. Primary factors that influence this decision are 1) memory utilization & cost & 2) page-transport efficiency. Page-transport efficiency refers to the performance cost & overhead of fetching page from the disk or, in a diskless workstation environment, across the network. Loading of a page from disk consists of two basic components: the disk-access time necessary to position the heads over the target track & sector, & the page-transfer time necessary to transfer the page to main memory thereafter. Head positioning delays generally exceed disk-memory transfer times by order of magnitude. Thus, total page-transfer time tends to be dominated by the disk positioning delay, which is independent of the page size.

Small page size reduces page breakage, & it may make better use of memory by containing only a specific locality of reference. Research results suggest that procedures in many applications tend to be smaller than 100 words. On the other hand, small pages may result in excessive size of mapping tables in virtual systems with large virtual-address spaces. Page-transport efficiency is also adversely affected by small page sizes, since the disk-accessing overhead is imposed for transferring a relatively small group of bytes.

Large pages tend to reduce table fragmentation & to increase page-transport efficiency. This is because the overhead of disk accessing is amortized over a larger number of bytes whenever a page is transferred between disk & memory. On the negative side, larger pages may impact memory utilization by increasing page breakage & by spanning more than one locality of reference. If multiple localities contained in a single page have largely dissimilar patterns of reference, the system may experience reduced effective memory utilization & wasted I/O bandwidth. In general, the page-size trade-off is technology-dependent, & its outcome tends to vary as the price & performance of individual components change.

6.2.2.8 Protection & Sharing

The frequent moves of items between main & secondary memory may complicate the management of mapping tables in virtual systems. When several parties share an item in real memory, the mapping tables of all involved processes must point to it. If the shared item is selected for removal, all concerned mapping tables must be updated accordingly. The overhead involved tends to outweigh the potential benefit or removing shared items. Many systems simplify the management of mapping tables by fixing the shared objects in memory.

An interesting possibility provided by large virtual-address spaces is to treat the OS itself as a shared object. As such, the OS is mapped as a part of each user's virtual space. To reduce table fragmentation, dedicated mapping registers are often provided to access a single physical copy of the page-map table reserved for mapping references to the OS. One or more status bits direct the mapping hardware to use the public or private mapping table, as appropriate for each particular memory reference. In this scheme, different users have different access rights to portions of the OS. Moreover, the OS-calling mechanism may be simplified by avoiding expensive mode switches between users & the OS code.

With the protection mechanism provided by mapping, a much faster CALL instruction, or its variant, may be used to invoke the OS.

6.2.3 Segmentation & Paging

It is also possible to implement virtual memory in the form of demand segmentation inheriting the benefits of sharing & protection provided by segmentation. Moreover, their placement policies are aided by explicit awareness of the types of information contained in particular segments. For example, a “working set” of segments should include at least one each of code, data, & stack segments. As with segmentation, inter-segment references alert the OS to changes of locality. However, the variability of segment sizes & the within-segment memory contiguity requirement complicate the management of both main & secondary memories. Placement strategies are quite complex in segmented systems. Moreover, allocation & deallocation of variable-size storage areas to hold individual segments on disk imposes considerably more overhead than handling of pages that are usually designed to fit in a single disk block.

On the other hand, paging is very easy for the management of main & secondary memories, but it is inferior with regard to protection & sharing. The transparency of paging necessitates the use of probabilistic replacement algorithms which virtually no guidance from users, they are forced to operate mainly on the basis of their observations of program behavior.

Both segmented & paged implementations of virtual memory have their advantages/disadvantages. Some systems combine the two approaches in order to enjoy the benefits of both. One approach is to use segmentation from the user’s point of view but to divide each segment into pages of fixed size for purposes of allocation. In this way, the combined system retains most of the advantages of segmentation. At the same time, the problems of complex segment placement & management of secondary memory are eliminated by using paging.

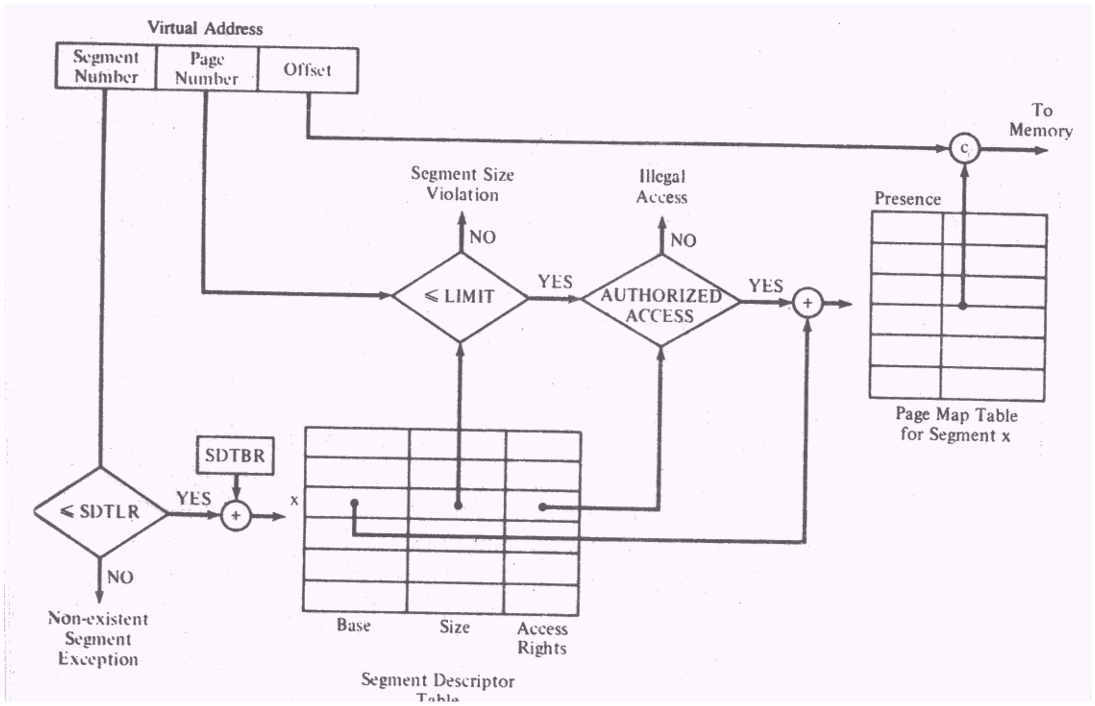


Figure 5 – Segmentation & paging

The principle of address translation in combined segmentation & paging systems is shown in Figure 5. Both segment descriptor tables & PMT are required for mapping. Instead of containing the base & limit of the corresponding segment, each entry of the SDT contains the base address & size of the PMT to be used for mapping of the related segment's pages. The presence bit in each PMT entry indicates availability of the corresponding page in the real memory. Access rights are recorded as a part of segment descriptors, although they may be placed or refined in the entries of the PMT. Each virtual address consists of three fields: segment number, page number, & offset within the page. When a virtual address is presented to the mapping hardware, the segment number is used to locate the corresponding PMT. Provided that the issuing process is authorized to make the intended type of reference to the target segment, the page number is used to index the PMT. If the presence bit is set, obtaining the page-frame address from the PMT & combining this with the offset part of the virtual address complete the mapping. If the target page is absent from real memory, the mapping hardware generates a page-fault exception, which is processed. At both mapping stages, the length fields are used to verify that the memory references of the running process lie within the confines of it address space.

Many variations of this powerful scheme are possible. For example, the presence bit may be included with entries of the SDT. It may be cleared when no pages of the related segment are in real memory. When such a segment is referenced, bringing several of lits pages into main memory may process the segment fault. In general, page re-fetching has been more difficult to implement in a way that performs better than demand paging. One of the main reasons for this is the inability to predict the use of previously un-referenced pages. However, referencing of a particular segment increases the probability of its constituent pages being referenced.

While the combination of segmentation & paging is certainly appealing, it requires two memory accesses to complete the mapping of each virtual address resulting into the reduction of the effective memory bandwidth by two-thirds. It may be too much to bear even in the face of all the added benefits. Obviously, hardware designers of such systems must assist the work of the OS by providing ample support in terms of mapping registers & look aside buffers.

6.3 Keywords

6.4 SUMMARY

The memory-management layer of an OS allocates & reclaims portions of main memory in response to requests from other users & from other OS modules, & in accordance with the resource-management objectives of a particular system. Processes are created & loaded into memory in response to scheduling decisions that are affected by, among other things, the amount of memory available for allocation at a given instant. Memory is normally freed when resident objects terminate. When it is necessary & cost-effective, the memory manager may increase the amount of available memory by moving inactive or low-priority objects to lower levels of the memory hierarchy (swapping).

Thus, the memory manager interacts with the scheduler in selecting the objects to be placed into or evicted from the main memory. The mechanics of memory

management consists of allocating & reclaiming space, & of keeping track of the state of memory areas. The objective of memory management is to provide efficient use of memory by minimizing the amount of wasted memory while imposing little storage, computational, & memory-access overhead. In addition, the memory manager should provide protection by isolating distinct address spaces, & facilitate inter-process cooperation by allowing access to shared data & code. Partitioned allocation of memory imposes relatively little overhead, but it restricts sharing & suffers from internal or external fragmentation. Segmentation reduces the impact of fragmentation & offers superior protection & sharing by dividing each process's address space into logically related entities that may be placed into dis-contiguous areas of physical memory. Contiguity of the virtual-address space is maintained by performing address translation at instruction execution time. Hardware assistance is usually provided for this operation in order to avoid a drastic reduction of the effective memory bandwidth. Paging simplifies allocation & de-allocation of memory by dividing address spaces into fixed-sized chunks. Execution-time translation of virtual to physical addresses, usually assisted by hardware, is used to bridge the gap between contiguous virtual addresses & dis-contiguous physical addresses where different pages may reside.

Virtual memory removes the restriction on the size of address spaces of individual processes that is imposed by the capacity of the physical memory installed in a given system. In addition, virtual memory provides for dynamic migration of portions of address spaces between primary & secondary memory fin accordance with the relative frequency of usage.

6.5 SUGGESTED READINGS / REFERENCE MATERIAL

6. OS Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
7. Systems Programming & OSs, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
8. OSs, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
9. OSs-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.

10. OSs, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

6.6 SELF ASSESMENT QUESTIONS (SAQ)

1. What is the common drawback of all the real memory management techniques? How is it overcome in virtual memory management schemes?
2. What is the basic difference between paging & segmentation? Which one is better & why?
3. What extra hardware do we require for implementing demand paging & demand segmentation?
4. Differentiate between internal & external fragmentation.
5. What do you understand by thrashing? What are the factors causing it?

Lesson number: 7 — ***Writer: Dr. Rakesh Kumar***

Disk Scheduling — ***Vetter: Prof. Dharminder Kr.***

7.0 Objectives

The objective of this lesson is to make the students familiar with:

(a) The characteristics of the disk that affect the performance.

(b) A number of disk scheduling algorithms to improve the access time.

(c) Disk scheduling algorithm in fixed-head storage devices.

7.1 Introduction

We know that the processor and memory of the computer should be scheduled such that the system operates more efficiently. Another very important scheduler is the disk scheduler. The disk can be considered the one I/O device that is common to every computer. Most of the processing of computer system centers on the disk system. Disk provides the primary on-line storage of information, both programs and data. All the important programs of the system such as compiler, assemblers, loaders, editors, etc. are stored on the disk until loaded into memory. Hence it becomes all-important to properly manage the disk storage and it scheduling.

7.2 Presentation of contents

7.2.1 Storage device characteristics

7.2.1.1 Seek time

7.2.1.2 Latency time

7.2.1.3 Transfer time

7.2.2 Disk Scheduling

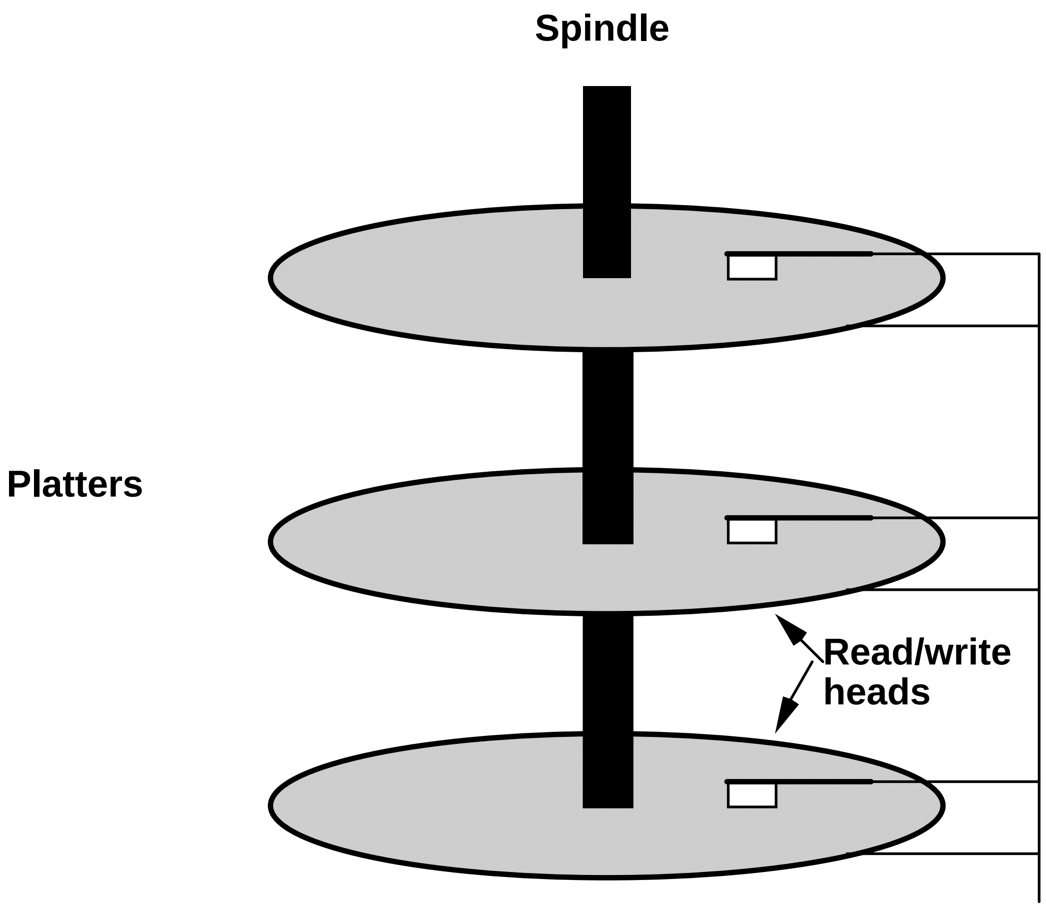
7.2.2.1 First come first serve (FCFS) scheduling

7.2.2.2 Shortest seek time first (SSTF) scheduling

7.2.2.3 Scan
7.2.2.4 C-scan (Circular scan)
7.2.2.5 Look
7.2.2.6 N-step scan
7.2.2.7 F-Scan
7.2.3 Scheduling algorithm selection
7.2.4 Sector queuing

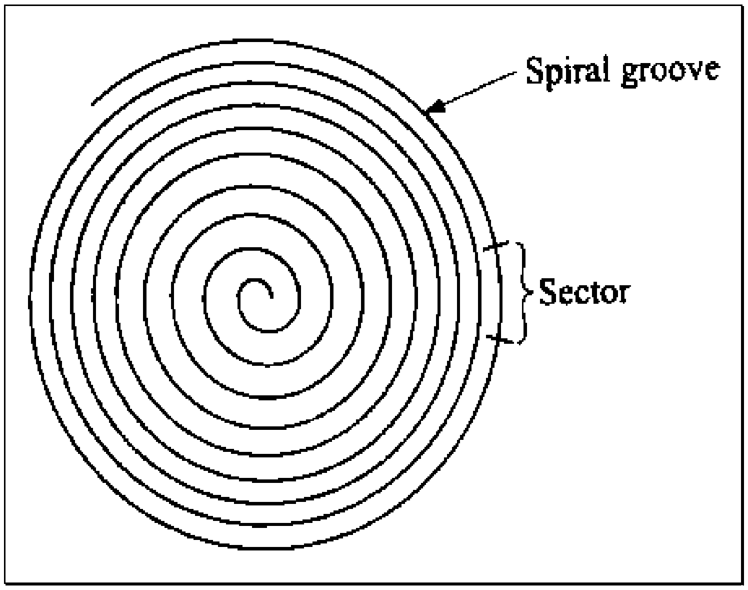
7.2.1 Storage device characteristics

Disk comes in many sizes and speeds, and information may be stored optically or magnetically. However, all disks share a number of important features. A disk is a flat circular object called a platter. Information may be stored on both sides of a platter (although some multiplatter disk packs do not use the top most or bottom most surface). Platter rotates around its own axis. The circular surface of the platter is coated with a magnetic material on which the information is stored. A read/write head is used to perform read/write operations on the disk.



Moving head disk mechanism

The read write head can move radially over the magnetic surface. For each position of the head, the recorded information forms a circular track on the disk surface. Within a track information is written in blocks. The blocks may be of fixed size or variable size separated by block gaps. The variable length size block scheme is flexible but difficult to implement. Blocks can be separately read or written. The disk can access any information randomly using an address of the record of the form (track no, record no.). When the disk is in use a drive motor spins it at high speed. The read/write head positioned just above the recording surface stores the information magnetically on the surface.



Tracks and sectors on a disk

On floppy disks and hard disks, the media spins at a constant rate. Sectors are organized into a number of concentric circles or tracks. As one moves out from the center of the disk, -the tracks get larger. Some disks store the same number of sectors on each track, with outer tracks being recorded using lower bit densities. Other disks place more sectors on outer tracks. On such a disk, more information can be accessed from an outer track than an inner one during a single rotation of the disk.

The disk speed is composed of three parts:

(a) Seek Time
(b) Latency Time
(c) Transfer time

7.2.1.1 Seek time

To access a block from the disk, first of all the system has to move the read/write head to the required position. The time consumed in this operation is known as seek time and the head movement is called seek. Seek time (S) is determined in terms of:

I: startup delays in initiating head movement.

H: the rate at which read/write head can be moved.

C: How far the head must travel.

$S = H * C + I$

7.2.1.2 Latency time

Once the head is positioned at the right track, the disk is to be rotated to move the desired block under the read/write head. This delay is known as latency time. On average this will be one-half of one revolution. Thus latency can be computed by dividing the number of revolution per minute (RPM), R, into 30.

$L = 30 / R$

7.2.1.3 Transfer time

Finally the actual data is transferred from the disk to main memory. The consumed in this operation is known as transfer time. Transfer time T, is determined by the amount of information to be read, B; the number of bytes per track, N; and the rotational speed.

$T = 60B/RN$

So the total time (A) to service a disk request is the sum of these tree i.e. seek time, latency time, and transfer time.

$A = S + L + T$

Since most of the systems depend heavily on the disk, so it become very important to make the disk service as fast as possible.

So a number of variations have been observed in disk organization motivated by the desire to reduce the access time, increase the capacity of the disk and to make optimum use of disk surface. For example there may be one head for every track on the disk surface. Such arrangement is known as fixed-head disk. In this arrangement it is very easy for computer to switch from one track to another quickly but it makes the disk very expensive due to the requirement of a number of heads. Generally there is one head that moves in and out to access different tracks because it is cheaper option.

Higher disk capacities are obtained by mounting many platters on the same spindle to form a disk pack. There is one read/write head per circular surface of a platter. All heads of the disk are mounted on a single disk arm, which moves radially to access different tracks. Since the heads are located on the identically positioned tracks of different surfaces, so such tracks can be accessed without any further seeks. So placing that data in one cylinder can speed up sequential access. Cylinder is a collection of identically positioned tracks of different surfaces.

The hardware for a disk system can be divided into two parts. The disk drive is the mechanical part, including the device motor, the read/write heads and associated logic. The other part called the disk controller determines the logical interaction with the computer. The controller takes instructions from the CPU and orders the disk drive to carry out the instructions.

Every disk drive has a queue of pending requests to be serviced. Whenever a process needs I/O to or from the disk, it issues a request to the operating system, which is placed in the disk queue. The request specifies the disk address, memory address, amount of information to be transferred, and the type of operation (input or output).

7.2.2 Disk Scheduling

For a multiprogramming system with many processes, the disk queue may often be non-empty. Thus, when a request is complete, the disk scheduler has to pick a new request from the queue and service it. As apparent, the amount of head movement needed to satisfy a series of I/O requests could affect the performance. For this reason, a number of scheduling algorithms have been proposed.

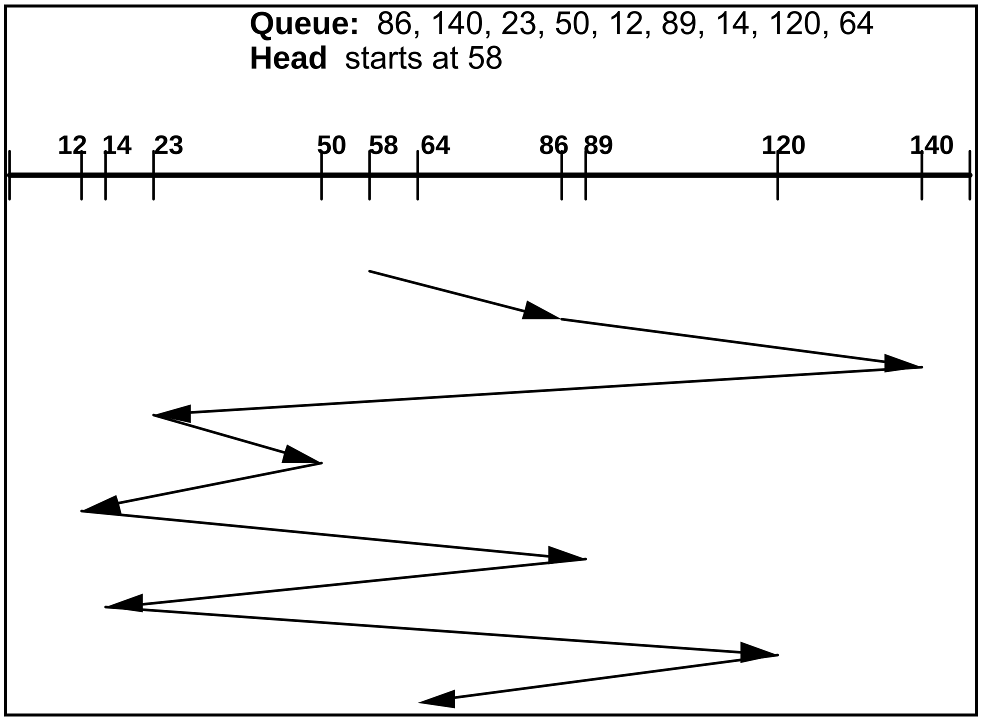
7.2.2.1 First cum first served (FCFS) scheduling

This form of scheduling is the simplest one but may not provide the best service. The algorithm is very easy to implement. In it the system picks every time the first request from the disk queue. In this scheduling the total seek time may be substantially high as evident from the following example:

Considered an ordered disk queue with requests involving tracks:

86,140, 23, 50, 12, 89, 14, 120, 64

The following figure shows the movement of read/write head in First Come First Serve scheduling.



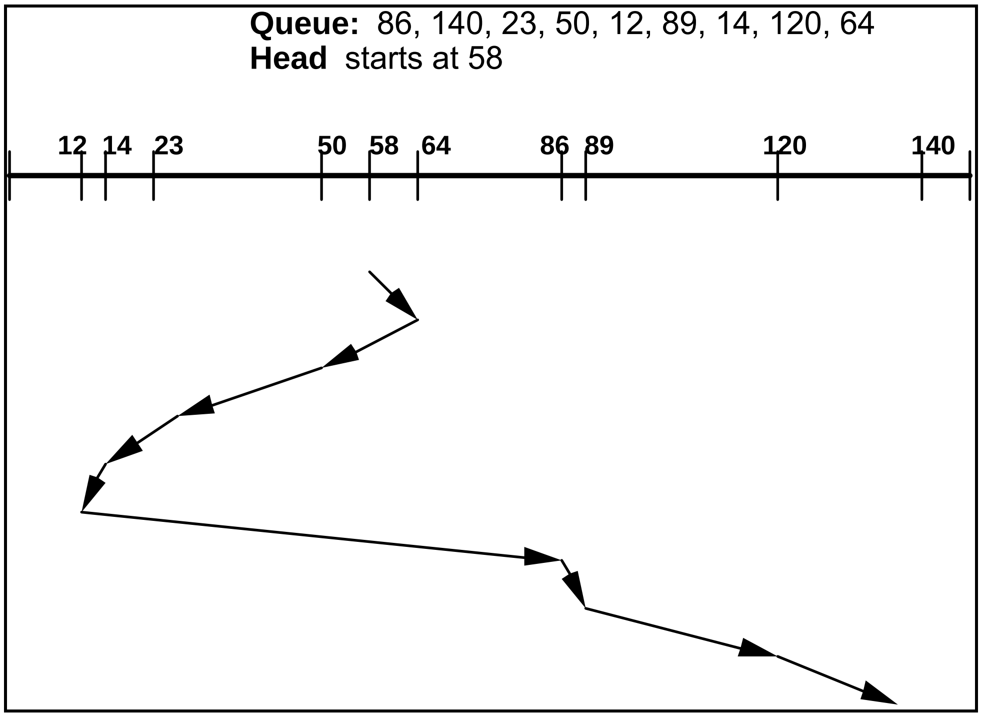
First Come First Serve Disk scheduling

7.2.2.2 Shortest seek time first (SSTF) scheduling

This scheduling algorithm services the request whose track position is closest to the current track position. Shortest-Seek-Time-First selects the request that is asking for minimum seek time from the current head position. Since seek time is generally proportional to the track difference between the requests, this approach is implemented by moving the head to the closest track in the request queue.

The following figure shows the read/write head movement in Shortest-Seek-Time-First scheduling for the above example discussed in First Come First Serve scheduling. It shows a substantial improvement in disk services i.e. reduction in the total movement of the head resulting into the reduced seek time.

Shortest Seek Time First is just like the Shortest Job First process scheduling. So it is having the limitations of Shortest Job First also. It may cause starvation of some requests.

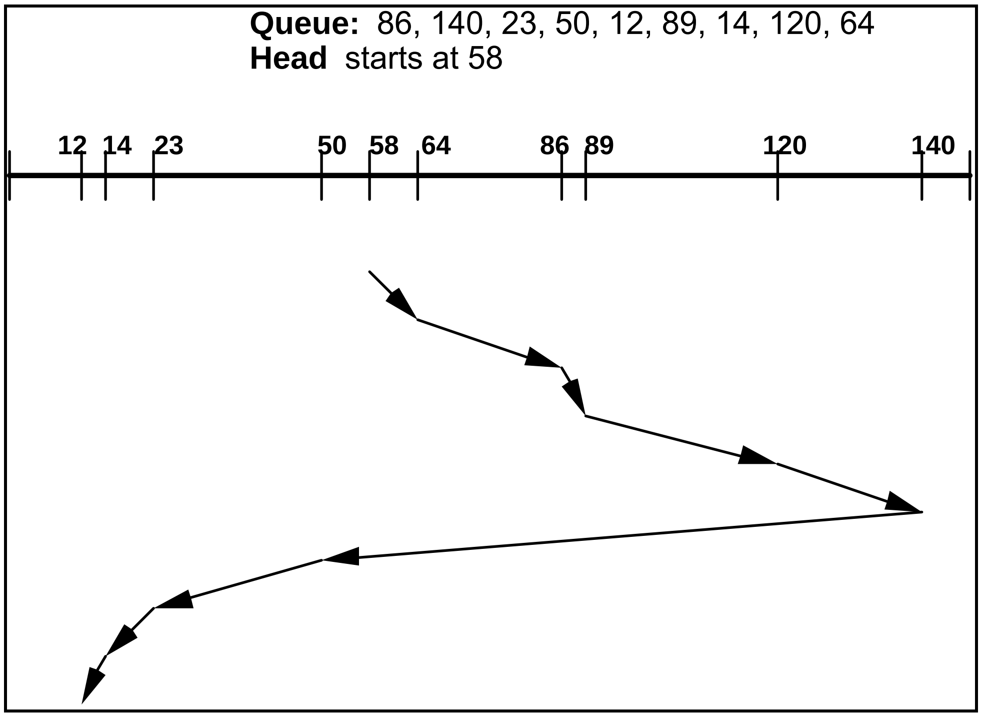


Shortest Seek Time First Scheduling

7.2.2.3 Scan

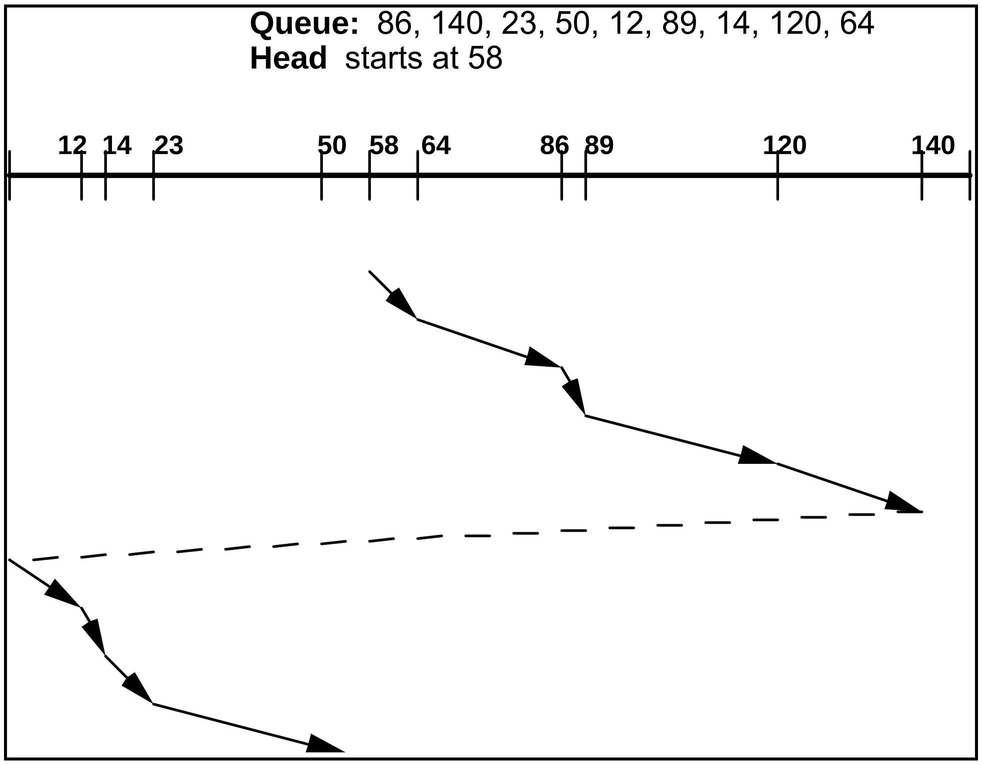
In this algorithm the read/write head moves back and forth between the innermost and outermost tracks. As the head gets to each track, satisfies all outstanding requests for that track. In this algorithm also, starvation is possible only if there are repeated requests for the current track.

The scan algorithm is sometimes called the elevator algorithm. As it is familiar to the behavior of elevators as they service requests to move from floor to floor in a building.



7.2.2.4 C-scan (Circular scan)

C-scan is a variant of scan. It is designed to provide a more uniform wait time. C-scan moves the head from one end of the disk to another, servicing requests as it goes. When it reaches the other end, however, it immediately return to the beginning of the disk, without servicing any requests on the return trip. C-scan treats the disk, as it was circular, with the last track adjacent to the first one.



7.2.2.5 Look

This algorithm is also similar to scan but unlike scan, the head does not unnecessarily travel to the innermost track and outermost track on each circuit. In this algorithm, head moves in one direction, satisfying the request for the closest track like scan in that direction. When there are no more requests in that direction the head is traveling, head reverse the direction and repeat.

7.2.2.6 N-step scan

In it the request queue is divided into sub queues with each sub queue having a maximum length of N. Sub queues are processed in FIFO order. Within a sub queue, requests are processed using Scan. While a sub queue is being serviced, incoming requests are placed in the next non-filled sub queue. N-step scan eliminates any possibility of starvation.

7.2.2.7 F-Scan

The "F" stands for "freezing" the request queue at a certain time. It is just like N-step scan but there are two sub queues only and each is of unlimited length.

While requests in one sub queue are serviced, new requests are placed in other sub queue.

7.2.3 Scheduling algorithm selection

As there are so many disk-scheduling algorithms, an important question is how to choose a scheduling algorithm that will optimize the performance. The commonly used algorithm is Shortest-Seek-Time-First and it has a natural appeal also. San and its variants are more appropriate for system with a heavy load on the disk. It is possible to define an optimal scheduling algorithm, but computational overheads required for that may not justify the savings over Shortest-Seek-Time-First and scan.

No doubt in any scheduling algorithm the performance depends on the number and types of the requests. If every time there is only one outstanding request, then the performance of all the scheduling algorithms will be more or less equivalent. Studies suggest that even First-Come-First-Serve performance will also be reasonably well.

It is also observed that performance of scheduling algorithms is also greatly influenced by the file allocation method. The requests generated by the contiguously allocated files will result in minimum movement of the head. But in case of indexed access and direct access where the blocks of a file are scattered on the disk surface resulting into a better utilization of the storage space, there may a lot of movement of the head.

In all these algorithms, to improve the performance, the decision is taken on the basis of head movement i.e. seek time. Latency time is not considered as a factor. Because it is not possible to predict latency time because rotational location cannot be determined. However, multiple requests for the same track may be serviced based on latency.

7.2.4 Sector queuing

In case of disk with one read/write head, the objective of the entire scheduling algorithm is to minimize the seek time by minimizing the movement o read/write head. The dick scheduling algorithms like First Come First Serve, Shortest-Seek-Time-First, Scan, and C-scan centers on this objective. But in case of the storage devices such as drum, which has the fixed head, this is not an issue at all. So different scheduling algorithms is used for this type of devices known as sector queuing.

In fixed-head storage devices the tracks are divided into a fixed number of blocks, known as sectors. Any I/O requests specify the address composed of track number and sector number. As seek time is zero for fixed-head storage devices, the important issue is the latency time to improve the performance.

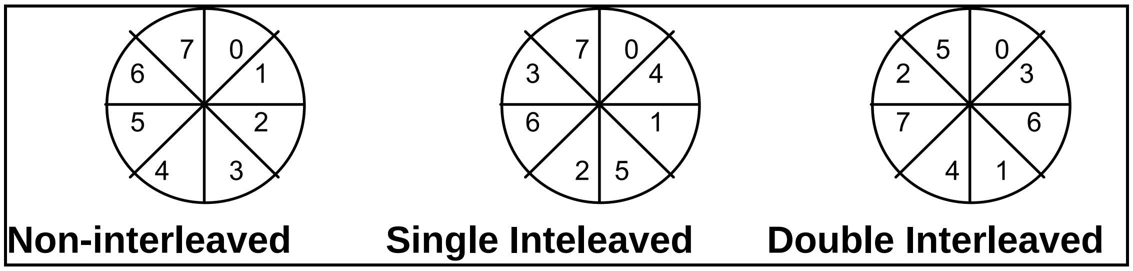
In sector queuing a separate queue is maintained for each sector of the drum. When a request arrives for sector i, it is placed in the queue for sector i. As sector i rotates beneath the read/write head, the first request in its queue is serviced.

Formatting

Before data can be written to a disk, all the administrative data must be written to the disks, organizing it into sectors. This low-level formatting or physical formatting is often done by the manufacturer. In the formatting process, some sectors may. be found to be defective. Most disks have spare sectors, and a remapping mechanism substitutes spare sectors for defective ones.

For sectors that fail after formatting, the operating system may implement a bad block mechanism. Such mechanisms are usually in terms of blocks and are implemented at a level above the device driver. The manner in which sectors are positioned on a track can affect disk performance. If disk I/O operations are limited to transferring a single sector at a time, to read multiple sectors in sequence, separate I/O operations must be performed. After the first I/O operation completes, its interrupt must be processed and the second I/O operation must be issued. During this time, the disk continues to spin. If the start of the next sector to be read has already spun past the read/write head, the sector cannot be read until the next revolution of the disk brings the cylinder by the read/write head. In a worst-case scenario, the disk must wait almost a full revolution. To avoid this problem, sectors may be interleaved. The degree of interleaving is determined by how far the disk revolves in the time from the end of one I/O operation until the controller can issue a subsequent I/O operation. The sector layout, given different degrees of interleaving, is illustrated in Fig. 5.

For most modem hard-disk controllers, interleaving is not used. The controller contains sufficient memory to store an entire track, so a single I/O operation may be used to read all the sectors on a track. Interleaving is more commonly used on less sophisticated disk systems, like floppy disks.



Interleaving

Most operating systems also provide the ability for disks to be divided into one or more virtual disks called partitions. On personal computers, Unix, Windows, and DOS all adhere to a common partitioning scheme so that they all may co-reside on a single disk.

Before files can be written to a disk, an empty file system must be created on a disk partition. This requires the various data structures required by the file system to be written to the partition, and is known as a high- level format or logical format. A file system is not required to make use of a disk. Some operating systems allow applications to write directly to a disk device. To an application, a directly accessible disk device is just a large sequential collection of storage blocks. In such a case, it is the responsibility of the application to impose an order on the information written there.

7.3 Summary

As processor and main memory speeds increase more rapidly than those of secondary storage devices, *optimizing disk performance has become important* to realizing optimal performance. Magnetic storage records data by changing the direction of magnetization of regions, each representing a 1 or a 0. To access data, a current-carrying device called a read/write head hovers above the medium as it moves.

Most modern computers use hard disks as secondary storage. As the platters spin, each read-write head sketches out a circular track of data on a disk surface. All read-write heads are attached to a single disk arm. When the disk arm moves the read/write heads to a new position, a different set of tracks becomes accessible. The time it takes for the head to move from its current cylinder to the one containing the data record being accessed is called the seek time. The time it takes for data to rotate from its current position to a position adjacent to the read/write head is called latency time. Then the record must be made to spin by the read/write head so that the data can be read from or written to the disk. Many processes can generate requests for reading and writing data on a disk simultaneously. Because these processes sometimes make requests faster than they can be serviced by the disk, queues build up to hold disk requests. Some early computing systems simply serviced these requests on a first-come-first-served (FCFS) basis. FCFS is a fair method of allocating service, but when the request rate becomes heavy, FCFS results in long waiting times.

FCFS exhibits a random seek pattern in which successive requests can cause time-consuming seeks from the innermost to the outermost cylinders. To reduce the time spent seeking records, it seems reasonable to order the request queue in some other manner than FCFS. This reordering is called disk scheduling. The two most common types of scheduling are seek optimization and rotational optimization. Disk scheduling strategies often are evaluated by comparing their throughput, mean response time and variance of response times.

Shortest-seek-time-first (SSTF) scheduling next services the request that is closest to the read-write head's current cylinder, even if that is not the first one in the queue. By reducing average seek times, SSTF achieves higher throughput rates than FCFS, and mean response times tend to be lower for moderate loads. One significant drawback is that higher variances occur on response times because of the discrimination against the outermost and innermost tracks; in the extreme, indefinite postponement of requests far from the read-write heads could occur.

The SCAN scheduling strategy reduces unfairness and variance of response times by choosing the request that requires the shortest seek distance in a preferred direction. Thus, if the preferred direction is currently outward, the SCAN strategy chooses the shortest seek distance in the outward direction. However, because SCAN ensures that all requests in a given direction will be serviced before the requests in the opposite direction, it offers a lower variance of response times than SSTF.

The circular SCAN (C-SCAN) modification to the SCAN strategy moves the arm from the outer cylinder to the inner cylinder, servicing requests on a shortest-seek basis. When the arm has completed its inward sweep, it jumps to the outermost cylinder, then resumes its inward sweep processing requests. C-SCAN maintains high levels of throughput while further limiting variance of response times by avoiding the discrimination against the innermost and outermost cylinders.

The FSCAN and N-Step SCAN modifications to the SCAN strategy eliminate the possibility of indefinitely postponing requests. FSCAN uses the SCAN strategy to service only those requests waiting when a particular sweep begins. Requests arriving during a sweep are grouped together and ordered for optimum service during the return sweep. N-Step SCAN services the first n requests in the queue using the SCAN strategy. When the sweep is complete, the next n requests are serviced. Arriving requests are placed at the end of the request queue, which prevents requests in the current sweep from being indefinitely postponed. FSCAN and N-Step SCAN offer good performance due to high throughput, low mean response times and a lower variance of response times than SSTF and SCAN.

The LOOK variation of the SCAN strategy "looks" ahead to the end of the current sweep to determine the next request to service. If there are no more requests in the current direction, LOOK changes the preferred direction and begins the next sweep, stopping when passing a cylinder that corresponds to a request in the queue. This strategy eliminates unnecessary seek operations experienced by other variations of the SCAN strategy by preventing the read/write head from moving to the innermost or outermost cylinders unless it is servicing a request to those locations.

The circular LOOK (C-LOOK) variation of the LOOK strategy uses the same technique as C-SCAN to reduce the bias against requests located at extreme ends of the platters. When there are no more requests on a current sweep, the read/write head moves to the request closest to the outer cylinder and begins the next sweep. The C-LOOK strategy is characterized by potentially lower variance of response times compared to LOOK, and by high throughput, although lower than that of LOOK. Sector queuing is a scheduling algorithm for fixed head devices such as drums.

7.4 Keywords

- **Seek time**: To access a block from the disk, first of all the system has to move the read/write head to the required position. The time consumed in moving the head to access a block from the disk is known as seek time.
- **Latency time:** the time consumed in rotating the disk to move the desired block under the read/write head.
- **Transfer time:** The consumed in transferring the data from the disk to the main memory is known as transfer time.

7.5 SELF ASSESMENT QUESTIONS (SAQ)

1. Compare the throughput of scan and C-scan assuming a uniform distribution of requests.

2. What do you understand by seek time, latency time, and transfer time? Explain.
3. Shortest Seek Time First favors tracks in the center of the disk. On an operating system using Shortest Seek Time First, how might this affect the design of the file system?
4. When there is no outstanding request in the queue, all the disk-scheduling algorithms reduce to First Come First Serve scheduling? Explain why?
5. The entire disk scheduling algorithms except First Come First Serve may cause starvation and hence not truly fair.
    - Explain why.
    - Come up with a scheme to ensure fairness.
    - Why is fairness an important goal in a time-sharing system?

7.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 8 **Writer: Dr. Rakesh Kumar**

Process Management ***Vetter: Prof. Dharminder Kr.***

8.0 Objectives

The objective of this lesson is get the students acquainted with the concepts of process synchronization. This lesson will make them familiar with:

(a) Critical section

(b) Mutual exclusion

(c) Classical coordination problems

8.1 Introduction

Since processes frequently need to communicate with other processes therefore, there is a need for a well-structured communication, without using interrupts, among processes. Processes use two kinds of synchronization to control their activities; (a) Control synchronization: it is needed if a process waits to perform some action only after some other processes have executed some action, (b) Data access synchronization: It is used to access shared data in a mutually exclusive manner. The basic technique used to implement this synchronization is to block a process until an appropriate condition is fulfilled. In this lesson synchronization in concurrent processes is discussed. Some classical coordination problems such as dining philosopher problem, producer-consumer problem etc are also discussed. These classical problems are the abstractions of the synchronization problems observed in operating systems.

8.2 Presentation of contents

8.2.1 Race Conditions

8.2.2 Critical Section

8.2.3 Mutual Exclusion

8.2.3.1 Mutual Exclusion Conditions

8.2.3.2 Proposals for Achieving Mutual Exclusion

8.2.4 Classical Process Co-Ordination Problems

8.2.4.1 The readers and writers problem
8.2.4.2 The Bounded Buffer Producers and Consumers
8.2.5 Semaphores
8.2.5.1 Producer-Consumer Problem Using Semaphores
8.2.6 The dining philosophers problem

8.2.1 Race Conditions

Consider the following extremely simple procedure

```
void deposit(int amount)
{
        balance = balance + amount;
 }
```

(Where we assume that balance is a shared variable). If two processes try to call deposit concurrently, something very bad can happen. The single statement balance = balance + amount is really implemented, on most computers, by a sequence of instructions such as

```
    Load  Reg, balance
    Add   Reg, amount
    Store Reg, balance
```

Suppose process P1 calls deposit(10) and process P2 calls deposit(20). If one completes before the other starts, the combined effect is to add 30 to the balance, as desired. However, suppose the calls happen at exactly the same time, and the executions are interleaved. Suppose the initial balance is 100, and the two processes run on different CPUs. One possible result is

P1 loads 100 into its register
P2 loads 100 into its register
P1 adds 10 to its register, giving 110
P2 adds 20 to its register, giving 120
P1 stores 110 in balance
P2 stores 120 in balance

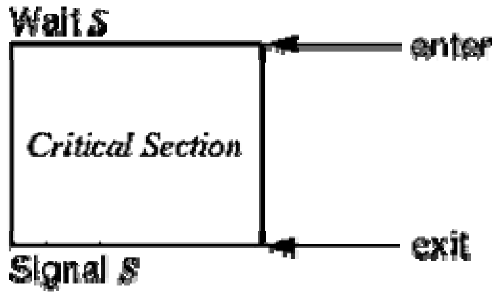
and the net effect is to add only 20 to the balance!

This kind of bug, which only occurs under certain timing conditions, is called a race condition. It is an extremely difficult kind of bug to track down (since it may disappear when you try to debug it) and may be nearly impossible to detect from testing (since it may occur only extremely rarely). The only way to deal with race conditions is through very careful coding. To avoid these kinds of problems, systems that support processes always contain constructs called synchronization primitives.

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one 'customer' thread at a time should be allowed to examine and update the shared variable. Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

So a race condition on a data item arises when many processes concurrently update its value. To maintain consistency, any time only one process should update the value. How to avoid race conditions? One solution is critical section.

8.2.2 Critical Section



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

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

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. So a critical section for a data item d is defined as a section of code, which cannot be executed concurrently with itself or with other critical sections of d.

Consider a system consisting of n processes {P_0, P_1, ..., P_{n-i}}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing 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. The critical-section problem is to design a protocol that the processes can use to co-operate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. An exit section may follow the critical section. The remaining code is the remainder section.

```
Repeat
entry section
Critical section
exit section
remainder section
until FALSE
```

Properties of critical section

The following properties are to be possessed by an implementation of critical section:

- Correctness: At most one process may execute a critical section at any given moment.
- Progress: When a critical section is not in use, one of the processes visiting to enter it will be granted entry to the critical section. If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next. Moreover, this decision cannot be postponed indefinitely. So if no process is in critical section, one can decide quickly who enters and only one process can enter the critical section so in practice, others are put on the queue.
- Bounded wait: After a process p has indicated its desire to enter a critical section, the number of times other processes gain entry to the critical section ahead of p is bounded by a finite integer. So there must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. The wait is the time from when a process makes a request to enter its critical section until that request is granted. In practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue)
- Deadlock freedom: The implementation is free of deadlock.

8.2.3 Mutual Exclusion

It 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.

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; 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 that 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.

8.2.3.1 Mutual Exclusion Conditions

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

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

8.2.3.2 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.

Problem

When one process is updating shared modifiable data in its critical section, no other process should allow 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 section and mutual exclusion achieved.

Disabling 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, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first tests 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.

```
Repeat
While Lock = 1
    Do {nothing};
{Critical section}
Lock = 0
{Remainder of the cycle}
Forever
```

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 inspects turn, finds it to be 1, and enters in its critical section. Process B also finds it to be 1 and sits in a loop continually testing 'turn' to see when it becomes 2.Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

Algorithm

```
Var num: integer
Begin
Turn=1
Concurrent begin
Repeat                          Repeat
  While turn=2                    While turn=1
      Do {nothing}                    Do {nothing}
  {critical section}              {critical section}
  turn=2                          turn=1
  {remainder of the cycle}        {remainder of the cycle}
forever                         forever
concurrent end
end.

Process 1                       Process 2
```

The shared variable turn is used to indicate which process can enter the critical section next. Let process p1 wish to enter the Critical Section. If turn=1, p1 can enter straightway. After completing the Critical Section, it sets turn to 2 so as to enable process p2 to enter the Critical Section. If p1 finds turn=2 when it wishes to enter Critical Section, it waits in the while loop until p2 exits from the critical section and executes the assignment turn=1. Thus processes may encounter a busy wait before gaining entry to the Critical Section.

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 1 finishes its critical section quickly, so both processes are now in their non-critical section. This situation violates above-mentioned condition 3(i.e. No process outside its critical section should block other processes.). E.g. let process p1 be in critical section and p2 in the remainder section. If p1 exits from the Critical Section, finishes the remainder section and wishes to enter the critical section again, it will face busy wait until after p2 uses the Critical Section. So the progress condition is violated since p1 is not able to enter although critical section is available. The solution of this problem can be:

```
Var c1, c2: integer
C1=1;c2=1;
{
repeat                                  repeat
        while c2=0 do {nothing}                 while c1=0 do {nothing}
        c1=0;                                   c2=0;
        {critical section}                      {critical section}
        c1=1;                                   c2=1;
        {remainder section}                     {remainder section}
forever                                 forever
}
process p1                              process p2
```

In this algorithm, c1 is a status flag for p1. p1 sets this flag to 0 while entering the critical section and change it to 1 upon exit. P2 checks the status of c1, if it is 1 then enter otherwise wait. This check eliminates the progress violation by enabling a process to enter critical section again any number of times if other process is not interested in entering the Critical Section. However the busy wait condition still exists.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above-mentioned solution does is this: when a process wants to enter in its critical section, it checks to see if the entry is allowed. If it is not, then it waits until it is allowed to enter. This approach waste CPU-time.

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

- ❖ 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'.

8.2.4 CLASSICAL PROCESS CO-ORDINATION PROBLEMS

There are a number of different process co-ordination problems arising in practical situations that exemplify important associated issues. These problems also provide base for solution testing for process co-ordination problems. In this section, we will see some of such classical process co-ordination problems.

8.2.4.1 The readers and writers problem

Courtois, Heymans, and Pumas posed an interesting synchronization problem called the readers-writers problem. Suppose a resource is to be shared among a community of processes of two distinct types: readers and writers. A reader process can share the resource with any other reader process but not with any writer process. A writer process acquires exclusive access to the resource whenever it requires any access to the resource.

This scenario is similar to one in which a file is to be shared among a set of processes, If a process wants only to read the file, then it may share the file with any other process that also wants to read the file. If a writer wants to modify the file, then no other process should have access to the file writer has access to it.

The correctness conditions are as follows:

1. There are two classes of processes:
   (a) Readers, which can work concurrently.
   (b) Writers, which need exclusive access.
2. Only one writer can write any time. So we must prevent 2 writers from being concurrent.
3. We must prevent a reader and a writer from being concurrent. So reading is prohibited while a writer is writing.

4. We must permit readers to be concurrent when no writer is active. So many readers can read simultaneously.
5. Perhaps we want fairness (i.e., freedom from starvation).
6. Optional variants can be:
    a. Writer should have priority over readers.
    b. Readers should have priority over writers.

The "easy way out" is to treat all processes as writers in which case the problem reduces to mutual exclusion. The disadvantage of the easy way out is that you give up reader concurrency. A possible solution is as under:

```
{                                   {
repeat                              repeat
    if a writer is writing              if readers reading writer is writing
    then {wait}                         then {wait}
    {read}                              {write}
    If writer waiting                   If reader or writer waiting
    Then                                Then
        Wake a writer if no                 Wake all readers
        Readers are reading.                Or one writer
forever                             forever
}                                   }

Readers                             Writers
```

8.2.4.2 The Bounded Buffer Producers and Consumers

As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem. In the producers and consumers problem, there are two classes of processes

- Producers, which produce items and insert them into a buffer.
- Consumers, which remove items and consume them.

These 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 from the buffer but buffer is already empty.

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

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

This approach also leads to same race conditions we 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.

Another solution is the use of monitors. Monitors is a high-level data abstraction tool that automatically generates atomic operations on a given data structure. A monitor has:

- Shared data.
- A set of atomic operations on that data.
- A set of condition variables.

Each monitor has one lock. It acquires lock when begin a monitor operation, and releases lock when operation finishes. It statically identifies operations that only read data, and then allow these read-only operations to go concurrently. Writers get mutual exclusion with respect to other writers and to readers. The advantages of using monitors are (i) it reduces probability of error (never forget to Acquire or Release the lock), (ii) biases programmer to think about the system in a certain way (is not ideologically neutral).

Bounded buffer using monitors and signals

- **Shared State** data [num] - a buffer holding produced data. num - tells how many produced data items there are in the buffer.
- **Atomic Operations** Produce (v) called when producer produces data item v. Consume (v) called when consumer is ready to consume a data item. Consumed item put into v.
- **Condition Variables** There are two condition variables (1) bufferAvail - signalled when a buffer becomes available. (2) dataAvail - signalled when data becomes available.

```
Condition *bufferAvail, *dataAvail;
int num = 0;
int data[10];
Lock *monitorLock;
Produce(v) {
monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
while (num == 10) {
    bufferAvail->Wait(monitorLock);
}
 put v into data array
num++;
 dataAvail->Signal(monitorLock);
  monitorLock->Release(); /* Release monitor lock after perform operation */
}
Consume(v) {
monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
```

```
      while (num == 0)
          dataAvail->Wait(monitorLock);
        }
      put next data array value into v
      num--;
      bufferAvail->Signal(monitorLock);
      monitorLock->Release(); /* Release monitor lock after perform operation */
      }
    }
```

8.2.5 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'.

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 operation 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 one 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.

8.2.5.1 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

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

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

8.2.6 The dining philosophers problem

The dining philosophers problem is a “classical” synchronization problem. Taken at face value, it is a pretty meaningless problem, but it is typical of many synchronization problems that you will see when allocating resources in operating systems.

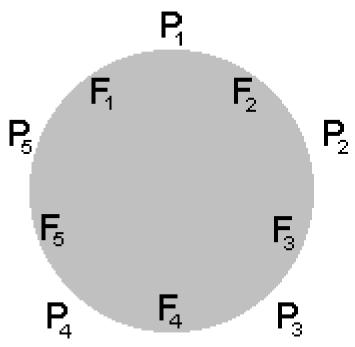
The problem is defined as follows: There are 5 philosophers sitting at a round table. Between each adjacent pair of philosophers is a chopstick. In other words, there are five chopsticks. Each philosopher does two things: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, he/she cannot eat until he/she owns the chopsticks to his/her left and right. When the philosopher is done eating he/she puts down the chopsticks and begins thinking again.

The challenge in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. every philosopher has a chopstick), and so that no philosopher starves (i.e. when a philosopher is hungry, he/she eventually gets the chopsticks). Additionally, our protocol should try to be as efficient as possible -- in other words, we should try to minimize the time that philosophers spent waiting to eat.

A simple solution to this problem can be of the form

```
Repeat
        Lift the left fork
        Lift the right fork
        {Eat}
        {Think}
Forever
```

But this solution is not acceptable since it is prone to deadlock (If all the philosophers lift their left fork).



One solution is to order the forks and require the philosophers to pick up the forks in increasing order, which mathematically eliminates the possibility of a deadlock. To illustrate this solution, label the philosophers P_1, P_2, P_3, P_4, and P_5, and label the forks F_1, F_2, F_3, F_4, and F_5. Each philosopher must pick up forks in a prescribed order and cannot pick up a fork another philosopher already has. Upon acquiring two forks, a philosopher may eat. Philosophers P_1 through P_4 follow the rule that P_x must pick up fork F_x first and then may pick up fork F_{x+1}. For example, P_1 must pick up F_1 first and F_2 second. Philosopher P_5 must, conversely, pick up fork F_1 before picking up fork F_5, to respect the deadlock-preventing fork ordering rule.

Although avoiding a deadlock, this solution is inefficient, because one can arrive to a situation where only one philosopher is eating and everybody else is waiting for him. For example philosophers P_1 to P_3 could hold forks F_1 to F_3, waiting to get forks F_2 to F_4 respectively, philosopher P_5 could wait on fork F_1 having no fork yet, while philosopher P_4 would be eating holding forks F_4 and F_5. Optimally, either philosopher P_1 or philosopher P_2 should be able to eat in such circumstances.

Preventing starvation depends on the method of mutual exclusion enforcement used. Implementations using spinlocks or busy waiting can cause starvation through timing problems inherent in these methods. Other methods of mutual exclusion that utilize queues can prevent starvation by enforcing equal access to a fork by the adjacent philosophers.

8.3 Summary

In operating systems, concurrent processes share some common storage that each process can read and write. Since processes frequently need to communicate with other processes therefore, there is a need for a well-structured communication. Processes use two kinds of synchronization to control their activities; Control synchronization, & Data access synchronization. A race condition on a data item arises when many processes concurrently update its value. One solution to race condition is critical section. If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. There are four conditions to hold to have a good solution for the critical section problem (mutual exclusion). In the approaches to mutual exclusion, if a process wants to enter in its critical section, it checks to see if the entry is allowed. If it is not allowed to enter, it waits. This approach waste CPU-time. To avoid this sleep and wakeup calls are used.

A semaphore is a protected variable whose value can be accessed and altered only by the indivisible operations P and V and initialization operation called 'Semaphoiinitislize' Semaphores helps in avoiding the race condition

8.4 Keywords

Critical section: that part of the program where the shared memory is accessed.

Mutual Exclusion: each process executing the shared data excludes all others from doing so simultaneously.

Semaphore: an object that hides an integer value and only allows three operations: initialization to a specified value, increment, or decrement.

8.5 Self assessment questions

1. What do you understand by critical section? What are the charcteristic properties of it? Explain.
2. What is mutual exclusion? Discuss the different approaches to solve the problem of mutual exclusion.
3. What do you understand by semphores? Does it satisfy the bounded wait condition? Explain.
4. What is semaphore? How does its help in avoiding the rce condition? Explain.
5. What do you understand by:
   (a) Busy waiting

(b) Bounded wait

8.6 Suggested readings/rfernce material

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 9	**Writer: Dr. Rakesh Kumar**
Deadlocks	**Vetter: *Prof. Dharminder Kr.***

Crises and deadlocks when they occur have at least this advantage that they force us to think."- Jawaharlal Nehru (1889 - 1964)

9.0 Objectives

The objectives of this lesson are to make the students acquainted with the problem of deadlocks. In this lesson, we characterize the problem of deadlocks and discuss policies, which an OS can use to ensure their absence. Deadlock detection, resolution, prevention and avoidance have been discussed in detail in the present lesson.

After studying this lesson the students will be familiar with following:

(a) Condition for deadlock.
(b) Deadlock prevention
(c) Deadlock avoidance
(d) Deadlock detection and recovery

9.1 Introduction

If a process is in the need of some resource, physical or logical, it requests the kernel of operating system. The kernel, being the resource manager, allocates the resources to the processes. If there is a delay in the allocation of the resource to the process, it results in the idling of process. The deadlock is a situation in which some processes in the system faces indefinite delays in resource allocation. In this lesson, we identify the problems causing deadlocks, and discuss a number of policies used by the operating system to deal with the problem of deadlocks.

9.2 Presentation of contents

9.2.1 Definition

9.2.2 Preemptable and Nonpreemptable Resources

9.2.3 Necessary and Sufficient Deadlock Conditions

9.2.4 Resource-Allocation Graph

9.2.4.1 Interpreting a Resource Allocation Graph with Single Resource Instances

9.2.5 Dealing with Deadlock

9.2.6 Deadlock Prevention

9.2.6.1 Elimination of “Mutual Exclusion” Condition

9.2.6.2 Elimination of “Hold and Wait” Condition

9.2.6.3 Elimination of “No-preemption” Condition

9.2.6.4 Elimination of “Circular Wait” Condition

9.2.7 Deadlock Avoidance

9.2.7.1 Banker’s Algorithm

9.2.7.2 Evaluation of Deadlock Avoidance Using the Banker’s Algorithm

9.2.8 Deadlock Detection

9.2.9 Deadlock Recovery

9.2.10 Mixed approaches to deadlock handling

9.2.11 Evaluating the Approaches to Dealing with Deadlock

9.2.1 Definition

A deadlock involving a set of processes D is a situation in which:

(a) Every process P_i in D is blocked on some event E_i.

(b) Event E_i can be caused only by action of some process (es) in D.

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant.

The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and CPU Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources A, say, a tap drive, and process 2 has be allocated non-sharable resource B, say, a printer. Now, if it turns out that process 1 needs resource B (printer) to proceed and process 2 needs resource A (the tape drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation ifs termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

9.2.2 Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

9.2.3 Necessary and Sufficient Deadlock Conditions

Coffman (1971) identified four (4) conditions that must hold simultaneously for there to be a deadlock.

1. Mutual Exclusion Condition

The resources involved are non-shareable.

Explanation: At least one resource (thread) must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait Condition

Requesting process hold already, resources while waiting for requested resources.

Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3. No-Preemptive Condition

Resources already allocated to a process cannot be preempted.

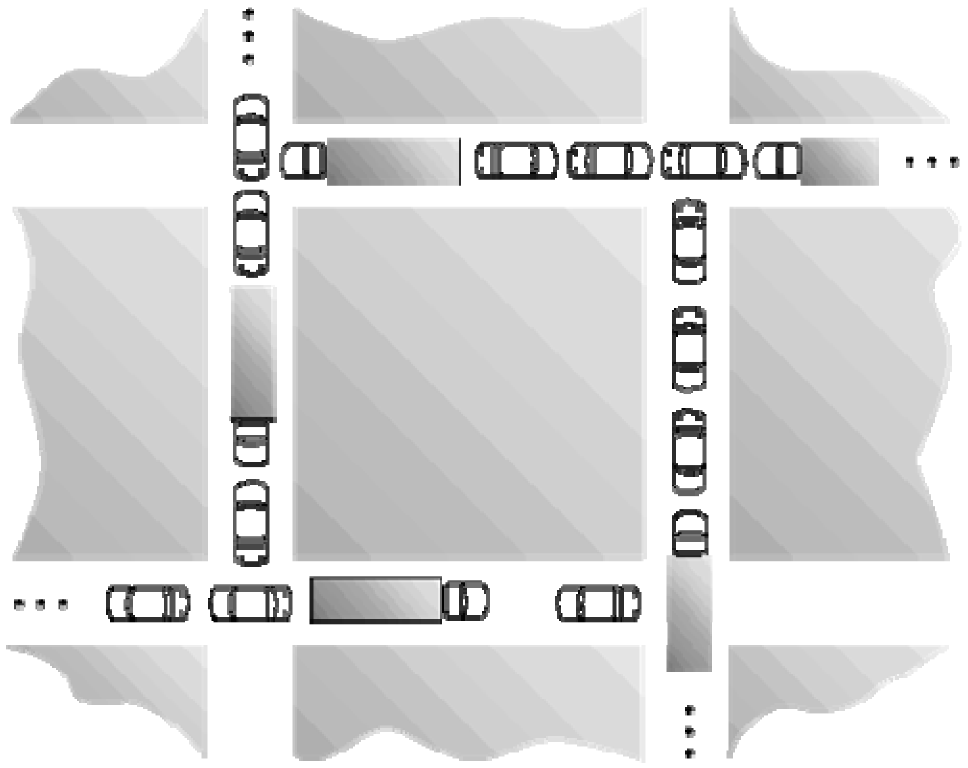
Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

4. Circular Wait Condition

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

Conditions 1 and 3 pertain to resource utilization policies, while condition 2 pertains to resource requirements of individual processes. Only condition 4 pertains to relationships between resource requirements of a group of processes.

As an example, consider the traffic deadlock in the following figure



Consider each section of the street as a resource.

1. Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
2. Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
3. No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

9.2.4 Resource-Allocation Graph

The deadlock conditions can be modeled using a directed graph called a resource allocation graph (RAG). A resource allocation graph is a directed graph. It consists of 2 kinds of nodes:

Boxes — Boxes represent resources, and Instances of the resource are represented as dots within the box i.e. how many units of that resource exist in the system.

Circles — Circles represent threads / processes. They may be a user process or a system process.

An edge can exist only between a process node and a resource node. There are 2 kinds of (directed) edges:

Request edge: It represents resource request. It starts from process and terminates to a resource. It indicates the process has requested the resource, and is waiting to acquire it.

Assignment edge: It represents resource allocation. It starts from resource instance and terminates to process. It indicates the process is holding the resource instance.

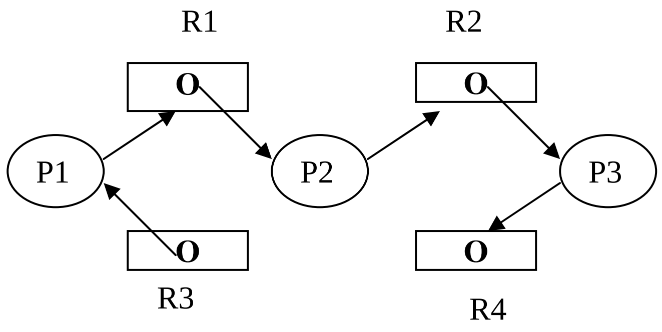
When a request is made, a request edge is added.

When request is fulfilled, the request edge is transformed into an assignment edge.

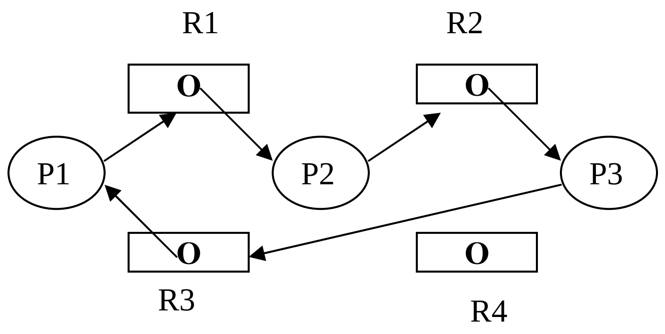
When process releases the resource, the assignment edge is deleted.

9.2.4.1 Interpreting a Resource Allocation Graph with Single Resource Instances

Following figure shows a resource allocation graph. If the graph does not contain a cycle, then no deadlock exists. Following figure is an example of a no deadlock situation.



If the graph does contain a cycle, then a deadlock does exist. As following resource allocation graph depicts a deadlock situation.



With single resource instances, a cycle is a necessary and sufficient condition for deadlock

9.2.5 Dealing with Deadlock

There are following approaches to deal with the problem of deadlock.

The Ostrich Approach — sticks your head in the sand and ignores the problem. This approach can be quite useful if you believe that they are rarest chances of deadlock occurrence. In that situation it is not a justifiable proposition to invest a lot in identifying deadlocks and tackling with it. Rather a better option is ignore it.

Deadlock prevention: This approach prevents deadlock from occurring by eliminating one of the four (4) deadlock conditions.

Deadlock detection algorithms: This approach detects when deadlock has occurred.

Deadlock recovery algorithms: After detecting the deadlock, it breaks the deadlock.

Deadlock avoidance algorithms: This approach considers resources currently available, resources allocated to each thread, and possible future requests, and only fulfill requests that will not lead to deadlock

9.2.6 Deadlock Prevention

Deadlock prevention is based on designing resource allocation policies, which make deadlocks impossible. Use of the deadlock prevention approach avoids the over- head of deadlock detection and resolution. However, it incurs two kinds of costs - overhead of using the resource allocation policy, and cost of resource idling due to the policy.

As described in earlier section, four conditions must hold for a resource deadlock to arise in a system:

- Non-shareable resources
- Hold-and-wait by processes
- No preemption of resources
- Circular waits.

Ensuring that one of these conditions cannot be satisfied prevents deadlocks. We first discuss how each of these conditions can be prevented and then discuss a couple of resource allocation policies based on the prevention approach. Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions.

9.2.6.1 Elimination of “Mutual Exclusion” Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

9.2.6.2 Elimination of “Hold and Wait” Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on “all or none” basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the “wait for” condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

9.2.6.3 Elimination of “No-preemption” Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the “no-preemptive” condition effectively.

The main drawback of this approach is high cost. When a process releases resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

9.2.6.4 Elimination of “Circular Wait” Condition

Presence of a cycle in resource allocation graph indicates the “circular wait” condition. The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.
For example, provide a global numbering of all the resources, as shown

1	≡	Card reader
2	≡	Printer
3	≡	Plotter
4	≡	Tape drive
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone. The resource ranking policy works best when all processes require their resources in the order of increasing ranks. However, difficulty arises when a process requires resources in some other order. Now processes may tend to circumvent such difficulties by acquiring lower ranking resources much before they are actually needed. In the worst case this policy may degenerate into the 'all requests together' policy of resource allocation. Anyway this policy is attractive due to its simplicity once resource ranks have been assigned.

"All requests together" is the simplest of all deadlock prevention policies. A process must make its resource requests together-typically, at the start of its execution. This restriction permits a process to make only one multiple request in its lifetime. Since resources requested in a multiple request are allocated together, a blocked process does not hold any resources. The hold-and-wait condition is satisfied. Hence paths of length larger than 1 cannot exist in the Resource Allocation Graph, a mutual wait-for relationships cannot develop in the system. Thus, deadlocks cannot arise.

9.2.7 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

9.2.7.1 Banker's Algorithm

In this analogy

Customers	≡	processes
Units	≡	resources, say, tape drive
Banker	≡	Operating System

Customers	**Used**	**Max**	
A	0	6	Available Units = 10
B	0	5	
C	0	4	
D	0	7	

In the above figure, we see four customers each of whom has been granted a number of credit units. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	**Used**	**Max**	
A	1	6	Available Units = 2
B	1	5	
C	2	4	
D	4	7	

Safe State The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure 2 is safe because with 2 units left, the banker can delay any request except C's, thus letting C finish and release all four resources. With four units in hand, the banker can let either D or B have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure 2.

We would have following situation

Customers	Used	Max	
A	1	6	
B	2	5	Available
C	2	4	Units = 1
D	4	7	

This is an unsafe state.

If all the customers namely A, B, C, and D asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later. Haberman [1969] has shown that executing of the algorithm has complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

9.2.7.2 Evaluation of Deadlock Avoidance Using the Banker’s Algorithm

There are following advantages and disadvantages of deadlock avoidance using Banker’s algorithm.

Advantages:

- There is no need to preempt resources and rollback state (as in deadlock detection & recovery)
- It is less restrictive than deadlock prevention

Disadvantages:

- In this case maximum resource requirement for each process must be stated in advance.
- Processes being considered must be independent (i.e., unconstrained by synchronization requirements)
- There must be a fixed number of resources (i.e., can't add resources, resources can't break) and processes (i.e., can't add or delete processes)
- Huge overhead — Operating system must use the algorithm every time a resource is requested. So a huge overhead is involved.

9.2.8 Deadlock Detection

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is O(N^2) where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

9.2.9 Deadlock Recovery

Once you've discovered that there is a deadlock, what do you do about it? One thing to do is simply re-boot. A less drastic approach is to yank back a resource from a process to break a cycle. As we saw, if there are no cycles, there is no deadlock. If the resource is not preemptable, snatching it back from a process

may do irreparable harm to the process. It may be necessary to kill the process, under the principle that at least that's better than crashing the whole system.
Sometimes, we can do better. For example, if we checkpoint a process from time to time, we can roll it back to the latest checkpoint, hopefully to a time before it grabbed the resource in question. Database systems use checkpoints, as well as a technique called logging, allowing them to run processes “backwards,” undoing everything they have done. It works like this: Each time the process performs an action, it writes a log record containing enough information to undo the action. For example, if the action is to assign a value to a variable, the log record contains the previous value of the record. When a database discovers a deadlock, it picks a victim and rolls it back.
Rolling back processes involved in deadlocks can lead to a form of starvation, if we always choose the same victim. We can avoid this problem by always choosing the youngest process in a cycle. After being rolled back enough times, a process will grow old enough that it never gets chosen as the victim--at worst by the time it is the oldest process in the system. If deadlock recovery involves killing a process altogether and restarting it, it is important to mark the “starting time” of the reincarnated process as being that of its original version, so that it will look older that new processes started since then.
When should you check for deadlock? There is no one best answer to this question; it depends on the situation. The most “eager” approach is to check whenever we do something that might create a deadlock. Since a process cannot create a deadlock when releasing resources, we only have to check on allocation requests. If the OS always grants requests as soon as possible, a successful request also cannot create a deadlock. Thus we only have to check for a deadlock when a process becomes blocked because it made a request that cannot be immediately granted. However, even that may be too frequent. As we saw, the deadlock-detection algorithm can be quite expensive if there are a lot of processes and resources, and if deadlock is rare, we can waste a lot of time checking for deadlock every time a request has to be blocked.

What's the cost of delaying detection of deadlock? One possible cost is poor CPU utilization. In an extreme case, if all processes are involved in a deadlock, the CPU will be completely idle. Even if there are some processes that are not deadlocked, they may all be blocked for other reasons (e.g. waiting for I/O). Thus if CPU utilization drops, that might be a sign that it's time to check for deadlock. Besides, if the CPU isn't being used for other things, you might as well use it to check for deadlock!

On the other hand, there might be a deadlock, but enough non-deadlocked processes to keep the system busy. Things look fine from the point of view of the OS, but from the selfish point of view of the deadlocked processes, things are definitely not fine. If the processes may represent interactive users, who can't understand why they are getting no response. Worse still, they may represent time-critical processes (missile defense, factory control, hospital intensive care monitoring, etc.) where something disastrous can happen if the deadlock is not detected and corrected quickly. Thus another reason to check for deadlock is that a process has been blocked on a resource request “too long.” The definition of “too long” can vary widely from process to process. It depends both on how long the process can reasonably expect to wait for the request, and how urgent the response is. If an overnight run deadlocks at 11pm and nobody is going to look at its output until 9am the next day, it doesn't matter whether the deadlock is detected at 11:01pm or 8:59am. If all the processes in a system are sufficiently similar, it may be adequate simply to check for deadlock at periodic intervals (e.g., one every 5 minutes in a batch system; once every millisecond in a real-time control system).

9.2.10 Mixed approaches to deadlock handling

The deadlock handling approaches differ in terms of theirv usage implications. Hence it is not possible to use a single deadlock handling approach to govern the allocation of all resources. The following mixed approach is found useful:

1. **System control block:** Control blocks like JCB, PCB etc. can be acquired in a specific order. Hence resource ranking can be used here. If a simpler

strategy is desired, all control blocks for a job or process can be allocated together at its initiation.

2. **I/O devices files:** Avoidance is the only practical strategy for these resources. However, in order to eliminate the overheads of avoidance, new devices are added as and when needed. This is done using the concept of spooling. If a system has only one printer, many printers are created by using some disk area to store a file to be printed. Actual printing takes place when a printer becomes available.
3. **Main memory:** No deadlock handling is explicitly necessary. The memory allocated to a program is simply preempted by swapping out the program whenever the memory is needed for another program.

9.2.11 Evaluating the Approaches to Dealing with Deadlock

- The Ostrich Approach — ignoring the problem

It is a good solution if deadlock is not frequent.

Deadlock prevention — eliminating one of the four (4) deadlock conditions

This approach may be overly restrictive and results into the under utilization of the resources.

- Deadlock detection and recovery — detect when deadlock has occurred, then break the deadlock

In it there is a tradeoff between frequency of detection and performance / overhead added.

- Deadlock avoidance — only fulfilling requests that will not lead to deadlock

It needs too much a priori information and not very dynamic (can’t add processes or resources), and involves huge overhead

9.3 Summary

- A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. Processes compete for physical and logical resources in the system. Deadlock affects the progress of processes by causing indefinite delays in resource allocation.
- *There are four Necessary and Sufficient Deadlock Conditions (1) Mutual Exclusion Condition: The resources involved are non-shareable, (2) Hold and Wait Condition:*

Requesting process hold already, resources while waiting for requested resources,(3) No-Preemptive Condition: Resources already allocated to a process cannot be preempted,(4) Circular Wait Condition: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

- The deadlock conditions can be modeled using a directed graph called a resource allocation graph (RAG) consisting of boxes (resource), circles (process) and edges (request edge and assignment edge). The resource allocation graph helps in identifying the deadlocks.
- There are following approaches to deal with the problem of deadlock: (1) The Ostrich Approach — stick your head in the sand and ignore the problem, (2) Deadlock prevention — prevent deadlock from occurring by eliminating one of the 4 deadlock conditions, (3) Deadlock detection algorithms — detect when deadlock has occurred, (4) Deadlock recovery algorithms — break the deadlock, (5) Deadlock avoidance algorithms — consider resources currently available, resources allocated to each thread, and possible future requests, and only fulfill requests that will not lead to deadlock
- There are merits/demerits of each approach. The Ostrich Approach is a good solution if deadlock is not frequent. Deadlock prevention may be overly restrictive. In Deadlock detection and recovery there is a tradeoff between frequency of detection and performance / overhead added, Deadlock avoidance needs too much a priori information and not very dynamic (can't add processes or resources), and involves huge overhead

9.4 Keywords

Deadlock: A deadlock is a situation in which some processes in the system face indefinite delays in resource allocation.

Preemptable resource: A preemptable resource is one that can be taken away from the process with no ill effects.

Nonpreemptable resource: It is one that cannot be taken away from process (without causing ill effect).

Mutual exclusion: several processes cannot simultaneously share a single resource

9.5 SELF-ASSESMENT QUESTIONS (SAQ)

1. What do you understand by deadlock? What are the necessary conditions for deadlock?
2. What do you understand by resource allocation graph (RAG)? Explain using suitable examples, how can you use it to detect the deadlock?
3. Compare and contrast the following policies of resource allocation:

(a) All resources requests together.

(b) Allocation using resource ranking.

(c) Allocation using Banker's algorithm

On the basis of (a) resource idling and (b) overhead of the resource allocation algorithm.

4. How can pre-emption be used to resolve deadlock?
5. Why Banker's algorithm is called so?
6. Under what condition(s) a wait state becomes a deadlock?
7. Explain how mutual exclusion prevents deadlock.
8. Discuss the merits and demerits of each approach dealing with the problem of deadlock.
9. Differentiate between deadlock avoidance and deadlock prevention.
10. A system contains 6 units of a resource, and 3 processes that need to use this resource. If the maximum resource requirement of each process is 3 units, will the system be free of deadlocks for all time? Explain clearly.
    If the system had 7 units of the resource, would the system be deadlock-free?

9.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.
6. Operating Systems, A Concept-based Approach, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

Lesson Number: 10 | ***Writer: Dr. Rakesh Kumar***

Case Study of UNIX | ***Vetter: Prof. Dharminder Kr.***

10.0 Objectives

The objective of this lesson is

(A) To give an overview of the important features of UNIX operating system to the students.

(B) To make the students familiar with some important UNIX commands.

10.1 Introduction

UNIX is written in a high-level language giving it the benefit of machine independence, portability, understandability, and modifiability. Multi-tasking (more than one program can be made to run at the same time) and multi-user (more than one user can work at the same computer system at the same time) are the two most important characteristics of UNIX helping it in gaining widespread acceptance among a large variety of users. It was the first operating system to bring in the concept of hierarchical file structure. It uses a uniform format for files called the byte stream making the application programs to be written easily. UNIX treats every file as a stream of bytes so the user can manipulate his file in the manner he wants. It provides primitives that allow more complex and complicated programs to be built from the simpler ones. It provides very simple user interface both character-based and graphical based. It hides the machine architecture from the user. This helps the programmer to write different programs that can be made to run on different hardware configurations. It provides a simple, uniform interface to peripheral devices.

10.2 Presentation of contents

10.2.1 Versions
10.2.2 UNIX Architecture
10.2.3 Features of UNIX
10.2.3.1 Portability
10.2.3.2 Machine Independent
10.2.3.3 Multi-user Capability
10.2.3.4 Multitasking Capability
10.2.3.5 Software Development Tools
10.2.3.6 Built-in Networking
10.2.3.7 Security
10.2.4 Implementation of Operating System Functions
10.2.4.1 Process management functions
10.2.4.2 Memory Management
10.2.4.3 Device and File functions
10.2.5 UNIX Kernel
10.2.5.1 Assumptions about Hardware
10.2.5.2 Interrupts and Exceptions
10.2.5.3 Processor Execution Levels
10.2.6 File System and Internal Structure of Files
10.2.6.1 Representation of Data in a File
10.2.6.2 Directories
10.2.6.3 Blocks and Fragments
10.2.7 UNIX Shell
10.2.8 User Interaction with UNIX Operating System
10.2.8.1 Steps to Login
10.2.8.2 Changing your Password
10.2.8.3 UNIX Command Structure
10.2.9 Common UNIX Commands
10.2.10 File System, Permissions Changing Order and Group
10.2.11 UNIX Editors

10.2.1 Versions

Some popular versions of UNIX are AIX IBM, XENIX, ULTRIX, Sun OS, and BSD. The original version of UNIX came in the late 1960's designed by Ken Thompson at AT&T Bell Laboratories. At that time, Bell Labs were busy in designing an operating system called Multics with an objective to provide a very sophisticated and complex multi-user operating system that had support for many advanced features. However, Multics failed because the state of art provided by it at that time was too complex. So, Bell Labs had to withdraw themselves from the Multics project. Ken Thompson then started working on a simpler project and he named it UNIX. Dennis Ritchie rewrote the source code of UNIX operating system in C language.

By the year 1977, UNIX I system found its major contribution in the telephone companies, providing a good environment for program development, network transaction services and real time services. A large number of institutions and universities were provided licenses of UNIX system. In the year 1977, the UNIX system was first ported from a PDP to a non-PDP machine.

As the popularity of UNIX grew, many other companies came out with their own versions of UNIX and ported it onto other new machines. From the year 1977 to 1982, Bell Laboratories combined many AT & T variants into a single system and gave it a name UNIX System III. Bell Laboratories in this version brought out many new features and advancements. It was given the name UNIX System V. The people at University of California at Berkeley developed a variant to the UNIX System. Its recent version is called 4.3 BSD for VAX machines.

By the beginning of 1984, UNIX system was installed at about 1,00,000 different computer sites. It ran on a wide range of computers ranging from a mini computer to a mainframe.

10.2.2 UNIX Architecture

The high level architecture of the UNIX system is shown in Figure 10.1.

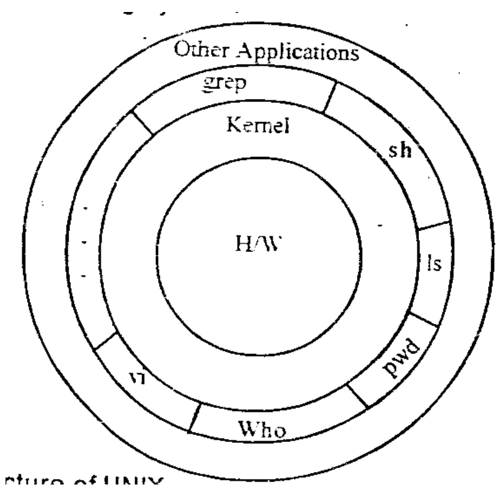


Figure 10.1 System Architecture of UNIX

The UNIX system seems to be organized as a set of layers. The Kernel surrounds the hardware. The user programs are independent of the hardware on which they are running. The programs such as the shell and editors interact with the Kernel by invoking a well-defined set of system calls. The system calls get various actions done from the Kernel for the calling program. They interchange data between the Kernel and the program. There are many other programs in this layer which from a part of the standard system configurations. These programs are known as commands. There are several other user created programs present in the same layer. The outer most layer contains other application programs which can be build on top of lower level programs. For instance, the C compiler appears in the outermost layer of the figure. It invokes a C preprocessor, compiler, assembler and link loader. These are all separate lower level programs. The programming style offered by the UNIX system helps us to fulfill a task by combining the existing programs.

10.2.3 Features of UNIX

The popularity of UNIX is due to the following reasons:

10.2.3.1 Portability

UNIX is its portable i.e. it can run successfully on all types of computers. The reason of this is that it is written in a high-level language. PCs, Macintoshes, Workstations, Minicomputers, Super Computers and Mainframes run the UNIX operating system with equal ease.

10.2.3.2 Machine Independent

The UNIX system does not expose the machine architecture to the user. Thus, it becomes very easy to write applications that can run on micros, minis or mainframes.

10.2.3.3 Multi-user Capability

UNIX is a multi-user system in which the same computer resources like hard disk, memory etc can be used or accessed by many users simultaneously. Each user is given a terminal. Each terminal is an input and an output device for the user. All the terminals are connected to the main computer. So, a user sitting at any terminal can not only uses the data or the software of the main computer but also the peripherals like printers attached to it. The main computer is called the server or the console. The number of terminals that can be connected to the server depends upon the number of parts present in the controller card.

10.2.3.4 Multitasking Capability

UNIX has the facility to carry out more than one job at the same time. Multitasking is achieved by dividing the CPU time in the order of milliseconds/microseconds for execution between all the jobs that are being carried out. Each job is carried out according to its priority number. It gives the impression that the tasks are being carried out simultaneously.

10.2.3.5 Software Development Tools

UNIX offers an excellent environment for developing new software. It provides a variety of tools ranging from editing a program to maintenance of software. It exploits the power of hardware to the maximum extent of effectiveness and efficiency.

10.2.3.6 Built-in Networking

UNIX has got built in networking support with a large number of programs and utilities. It also offers an excellent media for communication with other users. The users have the liberty of exchanging mail, data, programs, etc. You can send your data at any place irrespective of the distance over a computer network.

10.2.3.7 Security

UNIX enforces security at three levels.

(a) Each user is assigned a login name and a password. So, only the valid users can have access to the files and directories.

(b) Each file is bound around permissions (read, write, execute). The file permissions decide who can read/modify/execute a particular file. The permissions once decided for a file can also be changed from time to time.

(c) Then file encryption comes into picture. It encodes file in a format that cannot be very easily read. So, if anybody happens to open file, even then he will not be able to read the text of the file. However, you can decode the file for reading its contents.

10.2.4 Implementation of Operating System Functions

UNIX operating system performs following designated functions:

(a) **Process management functions**: creating, destroying and manipulating processes.

(b) **Memory management functions:** allocating, de-allocating and manipulating memory.

(c) **Input/Output functions:** communicating and controlling I/O device and file system.

(d) **Miscellaneous functions:** Network functions etc.

The UNIX System V offers somewhere around 64 system calls, which carry very simple options with them. So, it becomes easy to make use of these system calls. The body of the Kernel is formed by the set of system calls and the internal algorithms that implement them. Kernel provides all the services to the application programs in the UNIX system. In UNIX, the programs don't have any knowledge of the internal format in which the Kernel stores file data.

10.2.4.1 Process management functions

The behavior of a UNIX process is defined by its text segment, data segment and stack segment as shown in Figure 10.2.

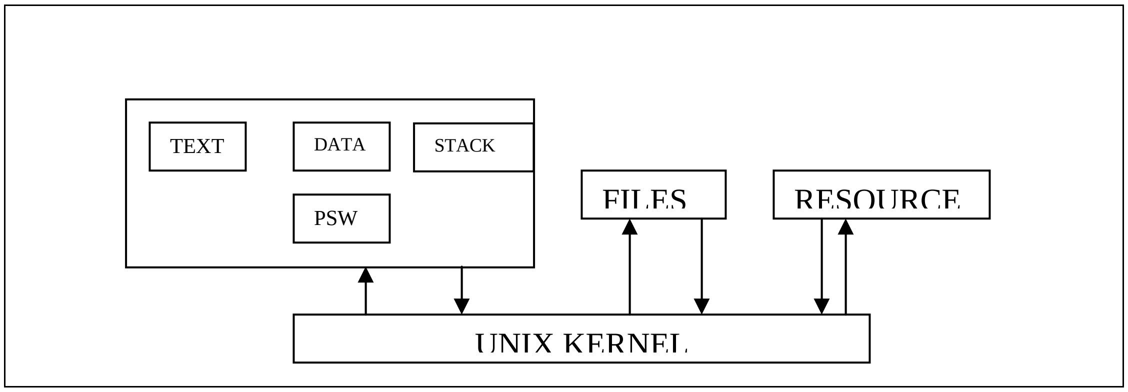


Figure 10.2: A Process in UNIX

The text segment contains the compiled object instructions, the data segment contains static variables, and the stack segment holds the runtime stack used to store temporary variables. A set of source file that is compiled and linked-into an executable form is stored in a file with the default name of a. out. If the program references statically define data, such as C static variables, a template for the data segment is maintained in the executable file. The data segment will be created and initialized to contain values and space for variables when the executable file is loaded and executed. The stack segment is used to allocate storage for dynamic elements of the program, such as automatic C variables that are created when they come into scope and are destroyed when pass out of scope.

The compiler and linker create the executable file. These utilities do not define a process; they define only the program text and a template for the data component that the process will use when it executes the program. When the loader loads a program into the computer's memory, the system creates appropriate data and stack segments, called a process.

A process has a unique process identifier (PID), a pointer to a table of process descriptors used by the UNIX OS kernel to reference the process's descriptor. Whenever one process references another process in a system call, it provides the pointer of the target process. The UNIX pa command lists each pm associated with the user executing the command. The pm of each process appears as a field in the descriptor of each process.

UNIX command for creating a new process is the fork system call. Whenever a process calls fork, a child process is created with its descriptor, including its own copies of the parent's program text, data, and segments, and access to all open file descriptors (in the kernel). The child and parent processes execute in their own separate address spaces. This means that even though they have access to the same information, both the child and its parent each reference their own copy of the data. No part of the address space of either process is shared.

Hence, the parent and child cannot communicate by referencing variables stored at the same address in their respective address space. UNIX systems also provide several forms of the execve system call to enable a process to reload its address space with a different program:

execve (char *path, char *avgv[], char *envp[]);

This system call causes the load module stored in the file at path to replace the program currently being executed by the process. After execve has completed executing, the program that called it, is no longer loaded. Hence, there is no notion of returning from an execve call, since the calling program is no longer loaded in memory. When the new program is started, it is passed the argument list, angv, and the process uses a new set of environment variables, envp.

UNIX also provides a system call, wait (and a variant, waitpid), to enable a parent process to detect when one of its child processes terminates. Details of the terminating child's status may be either returned to the parent via a value parameter passed to wait or ignored by the parent. The waitpid allows the parent to wait for a particular child process (based on its PID) to terminate, while the wait command does not discriminate among child processes. When a process exits, its resources, including the kernel process descriptor, are released. The operating system signals the parent that the child has died, but it will not release the process descriptor until the parent has received the signal. The parent executes the wait call to acknowledge the termination of a child process.

Following are the rest of UNIX system calls related to process management:

- acct enable/disable process accounting alarm set a process alarm clock exit terminate a process fork create a new process
- getpid get process, process group and parent process ID

- getuid get real user, effective user real group and effective group ID
- kill send a signal to a process or group of processes
- msgctl message control operation msgop message operation
- nice change priority of a process pause suspend until
- pipe create an inter-process channel
- profil execution time profile ptrace process trace
- semctl semaphor control operations
- semget get set of semaphor
- semop semaphor operations
- setpgrp: set process group ID
- setuid set group and user ID
- signal specify what to do when a signal is received
- stime set time
- sync update super block time get time
- times get process and child process times
- ulimit get user upper limits
- uname get the name of the current operating system
- ulink remove directory entry
- Wait wait for the child process to stop or terminate

10.2.4.2 Memory Management

Kernel resides in the main memory so long as computer is operational. When a program is compiled, a set of addresses is generated in the program by the compiler. These represent addresses of variables or addresses of instructions such as functions. The addresses generated by the compiler are for a virtual machine. The addresses are not absolute in terms of memory addresses where they will be loaded eventually. It assumes that no other program will be executing concurrently.

However, when you run the program the Kernel allocates some space to it in the main memory. But the virtual addresses generated by the compiler might not resemble the physical addresses occupied in the machine. Then the Kernel maps the compiler-generated address with the physical machine addresses.

UNIX divides the available memory into system memory and application memory. It loads itself into system memory and creates data structures it will use in its operation into this area of memory. The application memory area contains the user's programs. The application memory area "is divided into global stack, local / stack and heap. The global and static type of variables, functions etc. are assigned space in these memory areas. UNIX provides system calls to affect loading and unloading of programs into and out of the processes.

- Internally UNIX uses paging with segmentation methods to manage memory. In addition to these primitive operations, UNIX provides library functions like malloc, for allocating memory to a process and objects dynamically. Other memory related system calls are:
- brk change data segment space allocation
- shmop shared memory operations
- shmctl shared memory control operation
- shmget get shared memory segment
- plock lock process, text or data memory addresses,
- msgget get message queue

10.2.4.3 Device and File functions

For each device it has device drivers for low-level communication. UNIX treats every device the same way as it treats the files. Device drivers are intended to be accessed by user space code. If an application accesses a driver, it uses one of two standardized interfaces: the block device interface or the character device interface. Both interfaces provide a fixed set of functions to the user programs.

When a user program calls the driver, it performs a system call. The kernel searches the entry point for the device in the block or character in direct reference table (the jump table) and then calls the entry point. The exact semantics of each function depends on the nature of the device and the intent of the driver design. Hence, the function names suggest only a purpose for each. The logical contents of the jump table are kept in the file system in the dev directory.

A Unix driver has three parts:

- Code to initiate derive operations

- Device interrupt handlers

The initialisation code is run when the system is booted or started first time. It tests for the physical presence of respective devices and then initializes them. The API implements functions for a subset of the entry points. This part of the code also provides information to the kernel as to which functions are implemented. The system interrupt handler that corresponds to the physical device causing the interrupt calls the device interrupt handler.

System administrators are responsible for installing devices and drivers. The information necessary to install a driver can be incorporated into a configuration file by the administrator and then processed by the configuration builder tool /etc/conf.

UNIX has system calls to effect I/O manipulation. Some of them are:

- write - write on a file
- utime - set file access and modification times
- fstat - get file system statistics
- ulink - remove directory entry
- umount - unmount a file system
- umask - set and get file creation mask
- stat - get file status
- read - read from a file
- open - open for reading or writing h
- mount - mount a file system
- mknod - make a directory or special or ordinary file
- lseek - move read/write pointer link link to a file
- bfcntl - file control
- nexec - execute a file
- lytodup - duplicate an open file descriptor
- dbycreat - create a new file or rewrite an existing one
- the close close a file descriptor let of chroot change to root
- chown - change owner or group of file entry

- chmod - change mode of file
- chdir - change directory device access determine accessibility of a file

10.2.5 UNIX Kernel

The services provided by the Kernel are given below:

1. It controls the fate and state of various processes such as their creation, termination and I/O suspension.
2. The Kernel allocates main memory for an executing process. The Kernel allows the processes to share portions of their address space. It keeps the private space of processes secure and doesn't allow tampering from other processes. However, if the free memory is low with the system, then the Kernel frees out some memory by writing a process temporarily to secondary memory. In case the Kernel writes all the processes to the secondary memory, it is called a swapping system. However, if only the pages of memory are written onto the secondary memory, then it is called the paging system.
3. The Kernel schedules processes for execution on the CPU. The time-sharing concept allows the processes to share the CPU. When the time of a process has finished, the Kernel suspends it and puts some other ready process for execution in the CPU. It is again the work of the Kernel to reschedule the suspended process.
4. The Kernel permits different processes to make use of the peripheral devices such as terminals, tape drives, disk drives and network devices as and when requested.
5. The Kernel allocates the secondary memory for efficient storage and retrieval of user data. The Kernel allocates secondary storage for user files, organizes the file system in a well-planned manner and provides security to user files from illegal access.
6. The services provided by the Kernel are absolutely transparent to the user. For instance, the Kernel formats the data present in a file for internal storage. However, it hides the internal format from user processes. Similarly, it makes a distinction between the regular file or a device but hides the distinction from user processes. Finally, the Kernel provides the services so that the user level

processes can support the services they must provide. For instance, the Kernel provides the services that the shell requires to act as a command interpreter. Therefore, the Kernel allows the shell to read terminal input, to create pipes and to redirect I/O. The computer users can also create private versions of the shell so that they can create an environment according to their own requirements without disturbing the other users.

10.2.5.1 Assumptions about Hardware

Whenever the user on the UNIX system executes a process, it is divided into two levels: User level and Kernel level. So, as and when a process executes a system call, the execution mode of the process changes from the user mode to Kernel mode. The Kernel tries to process the requests made by the user. It returns an error message if the process fails. However, if no requests are given to the operating system to service, even then the operating system keeps itself busy with other operations such as handling interrupts, scheduling processes, managing memory and so on. The main differences between the user mode and the Kernel mode are given below:

1. Process in a user mode can access their own instructions and data but they cannot access the instructions and data of the Kernel. But all the processes present in the Kernel can have the access to both the Kernel and the user addresses.
2. Some machine instructions give an error message when executed in user mode. For instance, a machine may contain an instruction that manipulates the processor status register. This instruction is not allowed to be executed in user mode. Processes executing in user mode should not have this capability otherwise they may corrupt the kernel loaded.

It is very true that the system runs in either the user mode or the Kernel mode. However, the Kernel runs on behalf of the user process. The Kernel is not a separate process running parallel to user processes. The Kernel forms a part of each user process.

10.2.5.2 Interrupts and Exceptions

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU abruptly. Whenever the Kernel receives the interrupt, it saves the current work it is doing and services the interrupt. After the interrupt is processed, the Kernel resumes the interrupted work and proceeds as if nothing had happened. The hardware gives a priority weightage according to the order in which the interrupts should be handled. Thus, when the Kernel looks into an interrupt, it keeps the lower priority interrupts waiting and services the higher priority interrupts.

The term exception is different from the term interrupt. An exception is a condition in which a process causes an unexpected event. For instance, dividing a number by zero, illegal address, out of memory, etc. Exceptions occur in the middle of the execution of an instruction and are the similar to interrupts. The system tries to start the instruction again after handling the exception. However, interrupts are considered to happen between the executions of two instructions. The system continues working on the next instruction after servicing the interrupt.

10.2.5.3 Processor Execution Levels

Sometimes, the Kernel must stop the interrupt from occurring during critical activity preventing data corruption. For instance, the Kernel might not want to handle an interrupt when it is working with linked lists because handling the interrupt at this point of time might lead to corruption of pointers. Therefore, a better technique has been worked out. The processor execution levels can be set with the help of certain instructions. If you set the processor execution level to certain value, then it can keep away the interrupt from that level and lower levels. It will only allow the high level interrupts to disturb the process.

10.2.6 File System and Internal Structure of Files

Kernel does not impose any structure on files, and no meaning is attached to its contents - the meaning of bytes depends solely on the program that interprets the file. This is not true of just disc files but of peripherals devices as well. Magnetic tapes, mail messages, character typed on the keyboard, line printer output, data flowing in pipes - each of these is just a sequence of bytes as far as the system and the programs in it are concerned.

Files are organized in tree-structured directories. Directories are themselves files that contain information on how to find other files. A path name to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically it contains of individual file name elements separated by the slash character.

The UNIX file system supports two main objects: files and directories. Directories are nothing but files, which have a special format.

10.2.6.1 Representation of Data in a File

All the data entered by the user is kept in files. Internally the data blocks take up most of the data that has been put in files. Each block on the disk is addressable by a number. Associated with each file in UNIX is a little table called inode, which contains the table of contents to locate a file's data on disk. The table of contents consists of a set of disk block numbers. An inode maintains the attributes of a file, including the layout of its data on disk. Disk inodes consists of the following fields:

- Last modification date
- Last access date
- Time the file last read
- Last inode modification
- Time the file was last modified
- Reference count
- Block reference pointer and indirect pointer to blocks in the file

The data on a file is not stored in a contiguous section of the disk. The reason behind is that the Kernel will have to allocate and reserve continuous space in the file system before allowing operations that would increase the file size. For instance, let us suppose that there are three files A, B and C. Each file consists of 10 blocks of storage and supposes the system allocated storage for the three files contiguously as shown in Figure 10.3.

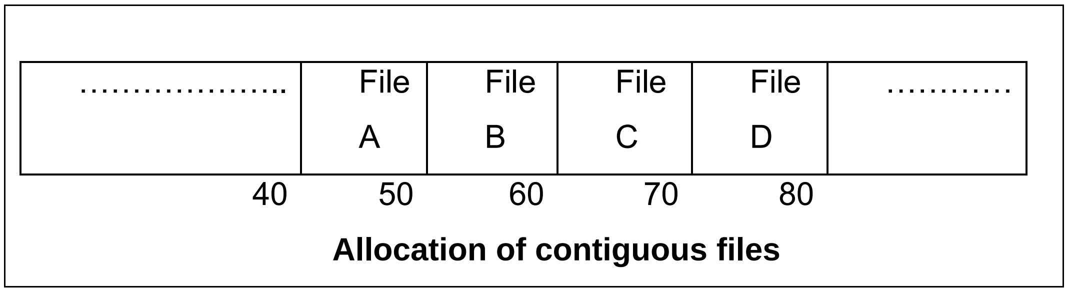


Figure 10.3

However, if the user now wishes to add 5 blocks of data in the file B, then the Kernel will have to copy the file to such a place where a file of 15 blocks can be accommodated. Moreover, the previously occupied disk block by file B's data can only be used in a case where the files have data less than 10 blocks.

The Kernel allocates the file space of one block at a time. This allows the data to be spread with throughout the file system. In this case, locating the data of a file becomes a complicated process. If a block contains 10K bytes, then such a file would need an index of 100 block numbers and as the block of 100K bytes would need an index of 1000 block numbers. Thus, the size of the inode would keep varying according to the size of the file.

10.2.6.2 Directories

The directories are files that give the file system a hierarchical structure. In a directory the data is put in a sequence of entries. Each such entry contains an inode number and the name of a file present in the directory .The pathname is a null terminated character string. The pathname is divided into separate parts by the / (slash) character. Each component of the pathname should hold the name of a directory. However, the very last component can be a non-directory file. The component names can have a maximum of 14 characters, with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes. Each directory contains the file names dot and dot-dot. The inode numbers of these directories are those of the directory and its parent directory respectively. The inode number of "." in "\etc" directory is present at offset 0 in the file and its value is 83. The inode number of "..” is present at the offset 16 and its value is 2. Any directory entry can also be kept empty. Its inode number is indicated by 0.

The data stored by the Kernel for a directory is similar to the data stored for an ordinary file. For the directory also the Kernel makes use of the inode structure and direct and indirect blocks. The access permission of a directory has the following meaning: the read permission allows a process to read a directory. Write permission allows a process to create new directory entries and remove the old directory entries. It accounts for altering the contents of a directory. The execute permission allows a process to search the directory for a filename.

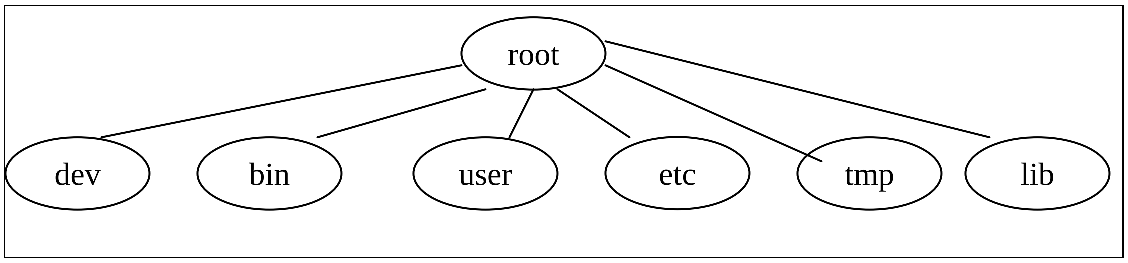


Figure 10.4

Figure 10.4 shows a typical UNIX File System. The file system is organized as a tree with a single root node called the root (written "/ "); every non-leaf node of the file system structure is a directory, and leaf nodes of the tree are either directories or regular files or special devices.

- The /bin directory contains the executable files for most UNIX commands.
- The /etc directory contains other additional commands that related to system maintenance and administration. It also contains several files, which store the relevant information about the users of the system, the terminals and devices connected to the system.
- The /lib directory contains all the library functions provided by UNIX for the programmers.
- The /dev directory stores files that are related to the devices. UNIX has a file associated with each of the I/O devices.
- The /user directory is created for each user to have a private work area where the user can store his files. This directory can be given any name. Here it is named as "user".
- The /tmp directory is the directory into which temporary files are kept. The files stored in this directory are deleted as soon as the system is shutdown and restarted.

Create, open, read, write are system calls, which are used for basic file manipulation. The create system call; given a path name creates an empty file. An existing file opened by the open system call, which takes a path name and a node and returns a small descriptor which may then be passed to a read or write system call to perform data transfer to or from the file.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files. Each read or write updates the current offset into the file, which is associated with file table entry and is used to determine the position in the field for the next read or write.

10.2.6.3 Blocks and Fragments

Most of the file system is taken up by data blocks, which contain whatever the users have put in their files. The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for a speed. However, because UNIX file system usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1 BSD file system was limited to 1024-byte block. The 4.2 BSD solution is to use two block sizes for files which have no indirect blocks: all the blocks of the file are large block size except the last. The last block is an appropriate multiple of a smaller fragment size to fill out the file. Thus, a file of size 18000 bytes would have two 8K blocks and one 2K block fragment.

The block and fragment sizes are set during the file creation according to the intended use of the file system: if many small files are expected, the fragment size should be small; if repeated transfer of large files are expected, the basic block size should be large.

10.2.7 UNIX Shell

A shell is the user-interface to the UNIX. A shell could use many different strategies to execute a user's computation. The approach used in modern shells is to create a new process (or thread) to execute new computation. For example, if a user decides to compile a program, the process interacting with the user creates a new child process to carry out the compilation took to execute the compiler program. The initial process (the OS) can use this same technique when it decides to service a new interactive user in a timesharing environment.

That is, when the user attempts to establish an interactive session, the OS treats this as a new computation. It awakens a previously created process for the login port or creates a new process to handle the interaction with the user.

This idea of creating a new process to execute a computation may seem like overkill, but it has a very important characteristic. When the original process decides to execute a new computation, it protects itself from any fatal errors that might arise during that execution. If it did not use a child process to execute the command, a chain of fatal errors could cause the initial process to fail, thus crashing the entire system.

The Bourne shell and others accept a command line from the user, parse the command line, and then invoke the OS to run the specified command with the specified arguments. When a user passes a command line to the shell, it is interpreted as a request to execute a program in the specified file - even if the file contains a program that the user wrote. That is, a programmer can write an ordinary C program, compile it, then have the shell execute it just like it was a normal UNIX command.

For example, you could write a C program in a file named main.c, then compile and execute it with shell commands like

```
$ cc main.c
$ a.out
```

The shell finds the cc command (the C compiler) in the /bin directory, then passes it the string "main.c" when it creates a child process to execute the cc program. The C compiler, by default, translates the C program that is stored in main.c, then writes the resulting executable program into a file named a.out in the current directory. In the second command, the command line is just the name of the file to be executed, a.out (without any parameters). The shell finds the a.out file in the current directory, then executes it.

Consider the detailed steps that a shell must take to accomplish its job:

- Printing a prompt. There is a default prompt string, sometimes hard coded into the shell, e.g., the single character string "%", "#", ">" or other. When the shell is started, it can look up the name of the machine on which it is running, and prepare this string name to the standard prompt character, for example giving a prompt string such as "kio-wa$". The shell can also be designed to print the current directory as part of the prompt, meaning that each time the user employs cd to change to a different directory, the prompt string is redefined.

- Once the prompt string is determined, the shell prints it to screen whenever it is ready to accept a command line.
- Getting the command line. To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be blocked until the user types a command line in response to the prompt. When the command has been provided by the user and terminated with a NEWLINE character, the command line string is returned to the shell.
- Parsing the command. The syntax for the command line is trivial. The parser begins at the left side of the command line and scans until it sees a white space character. The first such word is treated as the command name, and subsequent words are treated as parameter string.
- Finding the file. The shell provides a set of environment variables for each user-this variable is first defined in the user's login file, though it can be modified at any time with the set command. The PATH environment variable is an ordered list of absolute pathnames that specifies where the shell should search for command files. If the login file has a line such as

```
set path=(.:/bin:/usr/bin)
```

The shell will first look in the current directory (since the first pathname is "." for the current directory), then in /bin, and finally in /usr/bin. If there is no file with the same name as the command in any of the specified directories, the shell responds to the user that it is unable to find the command.

- Preparing the parameters. The shell simply passes the string parameters to the command as the argv array of pointers to strings.
- Executing the command. Finally the shell must execute the binary object program in the specified file. UNIX shells have always been designed to protect the original process from crashing when it executes a program. That is, since a command can be any executable file, the process that is executing the shell must protect itself in case the executable file has a fatal error in it. Somehow, the

shell wants to "launch" the executable so that even if the executable contains a fatal error (which destroys the process executing it), the shell will remain unharmed.

The Bourne shell uses multiple processes to accomplish what the UNIX-style system calls fork, execve, and wait. This system call creates a new process, which is a copy of the calling process except that it has its own process identification (with the correct relationships to the other processes) and its own pointers to shared kernel entities such as file descriptors. After fork has been called, two processes will execute the next statement after the fork in their own address spaces - the parent and the child. If the call succeeds in the parent process, fork returns the process identification of the newly created child process, and in the child process, fork() returns a zero value.

execve. This system call is used to change the program that the process is currently executing. It has the form execve (char *path, char *argv[], char *envp[]. The path argument is the pathname of a file that contains the new program to be executed. The argv array is a list of parameter strings, and the envp array is a list of environment variable strings and values that should be used when the process begins executing the new program. When a process encounters the execve system call, the next instruction it executes will be the one at the entry point of the new executable file. This means that the kernel performs a considerable amount of work in this system call. It must find the new executable file, load it into the address space currently being used by the calling process (overwriting area and discarding the previous program), set the argv array and environment variables for the new program execution, then start the process executing at the new program’s entry point.

There are various versions of execve available at the system call interface; they differ in the way parameters are specified.

wait. A process uses this system call to block itself until the kernel signals the process to execute again. For example, because one of its children processes has terminated. When the wait call returns as a result of a child process terminating, the status of the terminated child is returned as a parameter to the calling process.

10.2.8 User Interaction with UNIX Operating System

To interact with UNIX, the first step is login process in which we use a name and password initially assigned by the system administrator.

10.2.8.1 Steps to Login

Logging in is a procedure that tells the UNIX System who you are; the system responds by asking you the password. So, in order to login, first, connect your PC to the UNIX system. After a successful connection is established, you would find the following prompt coming in the on the screen.

```
Login:
```

Each user on the UNIX system is assigned an account name, which identifies him as a unique user. The account name is any combination of eight or less characters. Now, at the login prompt, enter your account name. Press Enter Key. Type your account name in lowercase letters. UNIX treats uppercase letters differently from lowercase letters.

```
Login: pankaj
Password: ******
```

Once the login name is entered, UNIX prompts you to enter a password. While you are entering your password, it will not be shown on the screen. If you give either the login name or the password wrong, then UNIX denies you the permission to access its resources. The system then shows an error message on the screen, which is given below:

```
Login: pankaj
Password: ******
Login incorrect:
Login:
```

Many UNIX systems give you three or four chances to enter your login and password correct. Once you have successfully logged on by giving a correct login and password, you are given some information about the system, some news for users and a message whether you have an electronic mail or not and followed y $ prompt. The dollar sign is the UNIX's method of telling that it's ready to accept commands from the user. You can have a different prompt also in a case where your system is configured for showing a different prompt. By default a $ is shown for the Korn or Bourne Shells.

At this point, you are ready to enter your first UNIX command. Now, when you are done working on your UNIX system and decide to leave your terminal - then it is always a good idea to log off the system. In order to log off the system, type the following command:

```
$ exit
login:
```

The above command will work if you are using a Bourne or a Korn shell. However, if you are working on C shell, you can give another command to log off.

```
$ logout
login:
```

UNIX system is very particular not to allow the unauthorized users to access the system. So, when a message like 'Login denied' comes on the screen, it does not tell you what was wrong with your login.

10.2.8.2 Changing your Password

You can change your password with the 'passwd' command. The procedure of changing passwords is very simple. In order to change your password, you first have to log on to the UNIX system. Then issue the 'passwd' command at the UNIX prompt.

Syntax: passwd [user-name] Options

-d Deletes your password

-x days. This sets the maximum number of days that the password will be date active. After the specified number of days you will be required to give a new password.

-n days. This sets the minimum number of days the password has to be active, before it can be changed.

-s: This gives you the status of the user's password.

Only the superuser can use the above options.

Example:

```
$passwd
```

```
Changing password for shefali
Enter old password: *******
Enter new password: *******
Re-type new password: *******
```

Thus UNIX wants you to type in your old password. Then it asks for the new password. Finally, UNIX confirms your new password by asking you to type in the new password again. If by any means, any mismatch happens, then the UNIX system warns you that the information provided by you is inconsistent as shown below:

```
$ passwd:
Changing password for shefali
Enter old password:
Enter new password:
Re-type new password:
Mismatch-password unchanged
```

UNIX also offers a variety of tools to maintain security .One such tool is the usage of the 'lock' command. The lock command locks your keyboard till the time you enter a valid password, as shown below:

```
$lock
Password: Sorry
Password:
```

10.2.8.3 UNIX Command Structure

There are a few of UNIX commands, that you can type them standalone. For example, ls, date, pwd, logout and so on. But UNIX commands generally require some additional options and/ or arguments to be supplied in order to extract more information. Let us find out the basic UNIX command structure. The UNIX commands follow the following format: **Command [options] [arguments]**

The options/arguments are specified within square brackets if they are optional. The options are normally specified by a “-“ (hyphen) followed by letter, one letter per option.

10.2.9 Common UNIX Commands

Some commonly used UNIX commands are discussed below:

cal Command

The cal command creates a calendar of the specified month for the specified year, if you do not specify the month, it creates a calendar for the entire year. By default this command shows the calendar for the current month based on the system date. The cal writes its output to the standard output.

Syntax: cal [[mm] yy]

where mm is the month, an integer between 1 and 12 and yy is the year; an integer between 1 and 9999, For current years, a 4-digit number must be used, '98' will not produce a calendar of 1998.

Options: None

date Command

It shows or sets the system date and time. If no argument is specified, it displays the current date and the current time.

Syntax: date [+options] Options:

%d displays date as mm/dd/yy

%a displays abbreviated weekday (Sun to Sat)

%t displays time as HH:MM:SS

%r displays time as HH:MM:SS(A.M/P.M.)

%d displays only dd

%m displays only mm

If you are working in the superuser mode, you can set the date as shown below:

$ date MMddhhmm[yy]

where MM = Month (1-12)

dd = day (1-31)

hh = hour (1-23)

mm = minutes (1-59)

yy = Year

who Command

The who command lists the users that are currently logged into the system.

Syntax: who [options]

Options:

- u - lists the currently logged-in users.
- t - gives the status of all logged-in users, am
- i - this lists login-id and terminal of the user invoking this command.

finger Command

In larger system, you may get a big list of users shown on the screen. The finger command with an argument gives you more information about the user. The finger command followed by an argument can give a complete information for a user who is not logged onto the system.

Syntax: finger [user-name] Options: none

Examples

(i) $ finger xyz

If you want to know about everyone currently logged onto the system, give the following command:

$finger

The ls Command

The ls command is used for listing information about files and directories.

Syntax: ls [-options] [filename]

Options:

-a - List all directory entries including dot (.) entries.

-d - Give the name of directories only.

-9 - Print group id only in the long listing.

-i - It Print inode number of each file in the first column.

-l - Lists in the long or detailed format owner's id only in the long listing.

-s - This lists the disk blocks (of 512 bytes each), occupied by a file.

Sort file names by time since last access.

-t - Sort file names by time since last modified.

-r - Recursively lists all subdirectories.

-f - Marks type of each file.

You can make use of more than one option at a given time, just group them together and precede them with "-".

The cp Command

The cp command creates a duplicate copy of a file.

Syntax: cp file1 file2

Options: None

Here, the file1 is copied as file2. If file2 already exists, the new file overwrites it. The file names specified may be full path names or just the name (current working directory, will then be assumed).

The mv Command

This command moves or renames files.

Syntax: mv file1 file2

Here file1 refers to the source filename and 'file2' refers to the destination filename. Moving a file to another within the same directory is equivalent to renaming the file. Otherwise also, mv doesn't really move the file, it just renames it and changes directory entries.

The In Command

The 'ln' command adds one or more links to a file. Syntax: In filel file2

The In command here establishes a link to an existing file. File name 'file1' specifies the file that has to be linked and file name 'file2' specifies the directory into which the link has to be established. If the 'file2' is in the same directory as file1 then the file seems to carry names, but physically there is only one copy. If you use the ls -li command, you will find that the link count has been incremented by one and that both the files have the same inode number, as they refer to the same data blocks in the disk. Any changes that are made to one file will be reflected in the other. And if 'file2' specifies a different directory, the file will be physically present at one place but will appear as if it is present in other directory, thereby allowing different users to access the file. It saves a lot of disk space because the file is not duplicated.

But you should note that you should have write permission to the directory under which the link is to be created.

(i) $ln /usr/mkt/mkt.c /usr/mktl/new-mkt.c

This will create a link for file mkt.c in 'mkt' directory to 'mktl' directory by the name 'new-mkt.c'.

(ii) $ In myfile.prg newfile.prg

The above command links the file 'myfile.prg' as 'new-file.prg' in the same directory. You can see these files by giving the ls command.

The rm Command

The 'rm' command removes files or directories. Syntax: rm [options] file(s)

When you remove a file, you are actually removing a link. The space occupied by the file on the disk is freed, only when you remove the last link to a file.

The Options:

c -confirms on each file before deleting it.

f - removes only those files which do not have write permission.

r - deletes the directory and its contents along with all the sub-directories and their contents.

The cat Command

The cat writes the contents of one of more files, onto the screen in the sequence specified. If you do not specify an input file, cat reads its data from the standard input file, generally the keyboard.

Syntax: cat file.

Examples

$ cat /usr/mkt/new-mkt.c

This command will display the contents of ‘c’ program file ‘new-mkt.c’ onto the screen.

chmod command

Only a superuser can change permissions for any file on the system.

g - the group to which the file owner belongs

o - other users, not part of the group

a - all users

'operation' denotes the options to be done, and can be:

+ add permission

- remove permission

= assign permission

‘permission' can be:

r - read permission

w - write permission

x - execute permission

‘filename(s)’ can be the files on which you want to carry out this command.

Examples - First see the file permissions using the ls -1 command for mkt.c as shown below:

```
$ls -l mkt.c
Output:  -rwx- -x- -2 root other 1428 May 1507:34 mkt.c
```

i.e. user has rwx, group has x and all others also have x permission.

(i) Now use the chmod command as illustrated below:

```
$chmod u-x g+w o+r mkt.c
```

The above command remove execute (x) permission for user, give write (w) permission to group and give read (r) permission to all others.

(ii) Then again use the ls -l command to verify, if the permissions have been or not set.

```
$1s -1 mkt.c
Output:   -rw- -wxr-x 2 root other 1428 May 1507:34 mkt.c
```

Alternatively, you could have also used the following commands to do the same work:

```
$chmod u=rw g=wx o=rx mkt.c
```

If we use

```
$ chmod a=rwx mkt.c "
$ ls -l mkt.c
Output:-rwxrwxrwx 2 root other 1428 May 15 07:34 mkt.c
```

In the above command, a=rwx assigns read, write and execute permission to all users.

The chown Command

The chown command changes the owner of the specified file(s).

Syntax: chown new-owner filename.

This command requires you to be in the superuser mode. The new owner can be the user ID of the new owner or the new owner's user number. You can also specify the owner by his name. But the new owner should have an entry in the /etc/passwd file. The filename is the name(s) of the file(s), whose owner is to be changed.

Options: None

Examples

```
$chown bobby sales.c
```

The above command now makes bobby the owner of sales.c file.

The chgrp Command

The chgrp is used to change the group of a file.

Syntax: chgrp group filename.

Only the superuser can use this command. This command changes the group ownership of a file. Here group denotes the new group-ID and filename denotes the file whose group-ID is desired to be changed.

10.2.10 File System, Permissions Changing Order and Group

File is a unit of storing information. All utilities, applications and data are represented as files. The file may contain executable programs, texts or databases. They are stored on secondary memory storage such as a disk or magnetic tape.

Naming Files

You can give filenames up to 14 characters long. The name may contain alphabets, digits and a few special characters. Files in UNIX do not have the concept of primary or secondary name as in DOS, and therefore file names may contain more than one period(.). Therefore, the following file names are all valid filenames: mkt.c, name2.c, .star, a.out

However, UNIX file names are case sensitive. Therefore, the following names represent two different files in UNIX. mkt.c and Mkt.c

Types

The files under UNIX can be categorized as follows:

1. Ordinary files.
2. Directory files.
3. Special files.
4. FIFO files.

We are discussing about these files below.

Ordinary Files

Ordinary files are the one, with which we all are familiar. They may contain executable programs, text or databases. You can add, modify or delete them or remove the file entirely.

Directory Files

Directory files as discussed earlier also represent a group of files. They contain list of file names and other information related to these files. Some of the commands, which manipulate these directory files, differ from those for ordinary files.

Special Files

Special files are also referred to as device files. These files represent physical devices such as terminals, disks, printers and tape-drives etc. These files are read from or written into just like ordinary files, except that operation on these files activates some physical devices. These files can be of two types Character device files and block device files. In character device files data is handled character by character, as in case of terminals and printers. In block device files, data is handled in large chunks of blocks, as in the case of disks and tapes.

FIFO Files

FIFO (first-in-first-out) are files that allow unrelated processes to communicate with each other. They are generally used in applications where the communication path is in only one direction, and several processes need to communicate with a single process. For an example of FIFO file, take the pipe in UNIX. This allows transfer of data between processes in a FIFO manner. A pipe takes the output of the first process as the input to the next process, and so on.

File Names and Meta characters

In UNIX, we can refer to a group of files with the help of METACHARACTERS. The valid meta characters are ?, [, and]. It replaces any number of characters including a null character.

? - used in place of one and only one character.

[] - brackets are used to specify a set of a range of characters.

Examples

(i) $ls []c

It will list all files starting with any character or characters and ending with the character c.

(ii) $ ls robin[]

It will list all the files starting with robin and ending with any character.

(iii) $ ls x?yz[]

It will list all those files, in which the first character is x, the second character can be anything; the third and fourth characters should be respectively y and z and that the rest of the name can be anything.

(iv) $ls l[abc]mn

It will list all those files, in which the first character is l, the second character can be either a, b or c the last two characters i.e., 3rd and 4th should be m and n respectively.

Alternatively the above command can also be given in the following manner.

$ ls l[a-c]mn

File Security and Ownership

The data is centralized on a system working with UNIX. The first step towards data security is the usage of passwords. The next step should be to guard the data among users. If the number of users is small, it is not much of a problem. But it can be problematic on a large system supporting many users.

UNIX can thus differentiate files belonging to an individual, the owner of a file or group of users or the others with different limited accesses, as the case may be. The different ways to access a file are:

Read(r) - You can just look through the file.

Write(w) - You can also modify it.

Execute(x) - You can just execute it

Therefore if you have a file called vendor.c and that you are the owner of it, you may provide; yourself with all the rights rwx [read, write and execute]. You can provide rx (read, and execute) rights to the members of your group and only the x (execute) right to all others.

Normally, when you create a file, you are the owner of the file and your group becomes the group id for the file. The system assigns a default set of permissions for file, as set by the system administrator. The user can also change these permissions at his will. But only a super user can change these permissions (rwx), ownership and group id's of any file in the system.

10.2.11 UNIX Editors

UNIX operating system provides several text editors can be classified into two types -Line Editors and Screen Editors.

(a) Line Editors: The early UNIX editors that edit processes one line at a time are called Line Editors. So with a line editor, you are required to give many commands to display or edit a set of lines. The common examples of UNIX line editors are ed and ex.

(i) ed - ed was the first line editor of UNIX which is still used sometimes, though it is not popular. ed was popular in those days when most UNIX commands consisted of only two or three letters. It has become outdated now due to the use of screen editors that provides much more features.

(ii) ex - ex is more powerful and comprehensive than ed line editor. Some ex line-oriented commands are also used in few screen editors (such as vi).

(b) Screen Editors: Editors that make use of the whole screen for editing or processing more than one line at a time, are called screen editors. With screen editors, you can display and edit many lines by giving a single command. The common examples of screen editors are vi and emacs.

- vi -vi stands for 'Visual editor'. vi is the standard full-screen UNIX tool and is the only editor available on SCO UNIX.
- emacs - emacs is another popular screen editor of UNIX. Although most vendors distribute emacs with UNIX system, emacs is not a part of UNIX.

10.3 Keywords

Multi-tasking: More than one program can be made to run at the same time.

Multi-user: More than one user can work at the same computer system at the same time.

Kernel: This is the actual operating system, a single large program that always resides in memory. Sections of the code in this program are executed on behalf of users to do needed tasks. Strictly speaking, the kernel is UNIX.

UNIX shell: A shell is the user-interface to the UNIX.

10.4 Summary

Unix is a multi-user, multi tasking operating system written in high-level language. It is portable, modifiable and understandable. Some popular versions of UNIX are AIX IBM, XENIX, ULTRIX, Sun OS, and BSD. The UNIX is organized as a set of layers. In the center, Kernel surrounds the hardware, which is surrounded by shell and editors that interact with the Kernel by invoking a well-defined set of system calls. Another important feature of UNIX is its security implemented by password, file permissions, and encryption. Files are organized in tree-structured directories. Directories are also files that contain information on how to find other files. Kernel does not impose any structure on files; the meaning of bytes depends solely on the program that interprets the file. This is not true of just disc files but of peripherals devices as well, each of these is just a sequence of bytes.

UNIX provides a number of line and screen editors such as ed, ex, and vi.

10.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. How many types of files there can be in UNIX?
2. How can security of files be maintained?
3. What are different types of users in UNIX for files?
4. What is function of ls command? Give various options.
5. Differentiate between cp & mv Command.
6. What is rm command used for?
7. Differentiate between chmod & chown Command.
8. What is meant by a multi-user and multi-tasking operating system? Is UNIX multi-user, how?
9. What are various Features of UNIX? What makes UNIX portable and secure?
10. Give different layers of UNIX Architecture. Explain the intended purposes of each.
11. What do you understand by Kernel? What are the functions provided by it?
12. How can the exceptions be resolved in UNIX?
13. How are file organized in UNIX? What is the difference between a directory & a file in Unix?

14. What do you understand by Editor? Why it is needed? How many types of Editors are present?

10.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. The Design of the UNIX Operating System, Bach M.J., PHI, New Delhi, 2000.
2. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
3. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
5. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
6. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson Number: 11 ***Writer: Dr. Rakesh Kumar***

Case Study of MS-DOS ***Vetter: Prof. Dharminder Kumar***

11.0 Objectives

The objective of this lesson is

(C) To give an overview of the important features of MS-DOS operating system.

(D) To make familiar with some important MS-DOS commands.

11.1 Introduction

MS-DOS is a single-user operating system. It is designed to operate on machines using the Intel line of 8086 microprocessors. These processors include the 8088, 8086, 80286, 80386, 80486, and the new Pentium (80586). PCs that are called 386s or 486s; are based on their processor names, 80386 and 80486. The 80586 was so different from the previous versions that it was given the name Pentium as a distinction. MS-DOS cannot support a large network of users. Because PCs and MS-DOS became so popular, many network operating systems were designed in the mid-1980s to network MS-DOS machines together; working around the operating system limitations through software. MS-DOS is the preferred operating system for most of the Intel processor (currently Pentium) based PCs of the world. MS-DOS does not break out into neat compartments as easily as some operating systems do. This is partially due to its simplicity -- no multi-user or multitasking ability. It is also partly due to the way MS-DOS has evolved over the years.

So in many respects DOS was a primitive OS. It was based on previous systems, of course, and echoes of UNIX and CP/M can be seen in it. But if you read Stephenson's article above, you will realize that it had hidden power, as well, because you could interact more directly with the components of the computer than you can with more modern operating systems. It is this power that makes it valuable to know DOS today. The majority of computer users today use graphical systems, but users of Windows, for instance, have the ability to interact with the computer through DOS (or in the case of NT, with a DOS-analogue) to do things which are difficult or impossible to accomplish through the graphical interface. For this reason, familiarity with DOS is still considered essential for anyone supporting Intel-based machines running Windows.

11.2 Presentation of contents

11.2.1 Kernel
11.2.2 COMMAND.COM
11.2.3 MS-DOS Services
11.2.4 Startup Files
11.2.5 Limitations
11.2.6 Security Shortcomings
11.2.7 Ease of Use
11.2.8 DOS Variations
11.2.9 The user's view of MS-DOS
    11.2.9.1 MS-DOS Commands
    11.2.9.2 Managing Input and Output
    11.2.9.3 Setting up the Environment
    11.2.9.4 Batch Files
    11.2.9.5 Working with Disks
    11.2.9.6 Working with Directories
    11.2.9.7 Managing Files
    11.2.9.8 Device drivers
    11.2.9.9 Memory Management
    11.2.9.10 Controlling Program Flow
11.2.10 System calls - The programmer's view
    11.2.10.1 Working with Disks
    11.2.10.2 Working with Directories

11.2.10.3 Managing Files
11.2.10.4 Managing Input and Output
11.2.10.5 Managing Memory
11.2.10.6 Controlling Program Flow

11.2.1 Kernel

MS-DOS uses two hidden files at boot time. These are io.sys and msdos.sys. For all practical purposes, these files, in conjunction with the firmware BIOS (Basic Input Output Services) built into every PC, make up the MS-DOS kernel (basic operating system). They load at the time of startup and allow the command processor to run. These files are not rebuildable or alterable. Software to run printers or CD-ROMs (device drivers) can be installed in MS-DOS but the kernel cannot be changed.

11.2.2 COMMAND.COM

This file starts the command processor. The command processor uses the commands you enter at the C:> prompt. When you run application programs and then return to MS-DOS, the system must be able to find COMMAND.COM and reload it back into memory (RAM). The command processor also supports a command language knows as the DOS Batch language. The Batch language files have a .BAT extension and are considered executable by the command processor. The Batch language is not as powerful as some command languages but does support conditional statements and variables (if time = next_day then...). In the early years of PCs, the Batch language was used for many tasks. Now, low cost utilities often provide many times the functions in addition to direct support. However, batch files are still very common and very useful.

11.2.3 MS-DOS Services

Because MS-DOS is not able to execute more than one task at a time, it cannot have multiple jobs working in the background. However, MS-DOS does use services that allow programs to interact with the computer. These services include basic input/output services (BIOS), print services, and file services. MS-DOS services use what is known as an interrupt to the microprocessor. This interrupt concept allows terminate-and-stay-resident (TSR) programs to function. TSRs can sit quietly in the background and appear when the user requests them. TSRs provide MS-DOS with the appearance of running more than one task at a time.

11.2.4 Startup Files

MS-DOS uses two changeable startup files. They are CONFIG.SYS and AUTOEXEC.BAT. The CONFIG.SYS has the task of loading installable device drivers and other system parameters which must run at boot time. The device drivers might be for a CD-ROM drive or a sound board. The AUTOEXEC.BAT starts applications automatically for the user, handles logins for network software, and places information in the PC’s environment. In some cases, there is not a clear-cut line that exists between which commands go into the CONFIG.SYS and which go into the AUTOEXEC.BAT. In general, the CONFIG.SYS holds only device drivers, special commands provided with MS-DOS, and memory managers.

11.2.5 Limitations

MS-DOS is limited in its memory usage. From the beginning, MS-DOS was designed to allow only the first 640 kilobytes of memory (RAM) to be used for application programs, even though only 1 megabyte of memory was addressable. To current MS-DOS users, 640 KB has proven to be inadequate as well. The latest version (as of this writing), MS-DOS 6.0, has special memory management tools and techniques built into allow programs additional memory. Additionally, and entire counter-culture of memory management tools have been developed which all programs to use more memory under previous versions of MS-DOS. These include Quarterdeck’s QEMM386, Qualitas Inc. 386 M, and Pharlap for the 486.

Closely tied to the limitation on memory is MS-DOS's inability to run more than one program at a time. The nature of MS-DOS requires one program to end before another one begins. Again, the ingenuity of programmers has worked numerous tricks to get around this limitation. For one, TSRs can load into memory and stay throughout a session. These are background programs available for use by other programs. However, many programmers have created poorly-behaved TSRs, which are unstable, or can cause multiple problems with other programs. Lack of standards has given TSRs a bad reputation. Sometimes a TSR itself is reliable, but does not work well with other TSRs. A second solution to the single application shortcoming has been task-switching programs like Software Carousel and the original Microsoft Windows. These allow you to quickly move between different applications that are in RAM or located on a hard disk swap space. Though certainly not multitasking, they are much better than nothing. Nevertheless, the only solution that attempts multitasking is Microsoft Windows. You should understand that no matter what Microsoft says, Microsoft Windows is not an operating system. MS-DOS remains the operating system and Windows is simply a user interface with lots of flexibility.

11.2.6 Security Shortcomings

MS-DOS has no concern for security or networking. There are no passwords, login files, encryption, networking hooks, mail programs, communication programs, or any of the other tidbits that users have come to take for granted. This security shortage presents a particularly difficult problem in a networked environment. Most network operating systems have put security measures on top of MS-DOS. These typically involve logins and access permissions to the programs on the server. MS-DOS now comes with a program called SHARE that controls access to the same file by multiple users over a network. Still, each individual MS-DOS machine is inherently susceptible to viruses, rogues, and security leaks.

11.2.7 Ease of Use

How did MS-DOS become so popular? The answer is a combination of marketplace luck, good advertising, low cost, and ease of use. Precisely because MS-DOS had no restraints and because IBM put MS-DOS on its PCs, anyone could easily buy a machine and put MS-DOS on their desk. The DOS batch language allowed nonprogrammers to write simple programs and interfaces. MS-DOS itself is not that easy to use, but the programs created to run under MS-DOS were themselves understandable to the computer illiterate individuals of the world.

11.2.8 DOS Variations

Though MS-DOS is the most popular, Microsoft does not make the only versions of DOS. IBM has a license to create DOS, which runs on IBM PS/2 computers and is called PC-DOS. Up until DOS 4.0, there were only minor differences between MS-DOS and PC-DOS. Since that time there has been more divergence between the two companies. In the late 1980s, a number of DOS multi-user versions appeared on the market including VM-386 and CMOS. The only major competitor to MS-DOS in the single machine market is DR-DOS from Digital Research (DRI), that is the same company, now owned by Novell Inc., which was involved in the original discussions with IBM that crashed and burned. Essentially, DR-DOS has jumped a release ahead of MS-DOS when it comes to new capabilities and technologies. DRI guarantees the compatibility of DR-DOS with programs designed for MS-DOS.

11.2.9 THE USER'S VIEW OF MS-DOS

MS-DOS provides an interface between the user and the computer system in form of a set of commands, which are available through a command interpreter. COMMAND.COM file is the MS-DOS command interpreter. The totality of DOS commands may be divided in three classes, viz., internal commands, external commands and batch command. In MS-DOS there are two types of commands. An Internal command, which is a command embedded into the command.com file, and an external command, which is not embedded into command.com and therefore requires a separate file to be used.

12.2.9.1 MS-DOS commands

COMMAND	DESCRIPTION
ANSI.SYS	Defines functions that change display graphics, control cursor movement, and reassign keys.
APPEND	Causes MS-DOS to look in other directories when editing a file or running a command.
ARP	Displays, adds, and removes arp information from network devices.
ASSIGN	Assign a drive letter to an alternate letter.
ASSOC	View the file associations.
AT	Schedule a time to execute commands or programs.
ATMADM	Lists connections and addresses seen by Windows ATM call manager.
ATTRIB	Display and change file attributes.
BATCH	Recovery console command that executes a series of commands in a file.
BOOTCFG	Recovery console command that allows a user to view, modify, and rebuild the boot.ini
BREAK	Enable / disable CTRL + C feature.
CACLS	View and modify file ACL's.
CALL	Calls a batch file from another batch file.
CD	Changes directories.
CHCP	Supplement the International keyboard and character set information.
CHDIR	Changes directories.
CHKDSK	Check the hard disk drive running FAT for errors.
CHKNTFS	Check the hard disk drive running NTFS for errors.
CHOICE	Specify a listing of multiple options within a batch file.

CLS	Clears the screen.
CMD	Opens the command interpreter.
COLOR	Easily change the foreground and background color of the MS-DOS window.
COMMAND	Opens the command interpreter.
COMP	Compares files.
COMPACT	Compresses and uncompress files.
CONTROL	Open Control Panel icons from the MS-DOS prompt.
CONVERT	Convert FAT to NTFS.
COPY	Copy one or more files to an alternate location.
CTTY	Change the computers input/output devices.
DATE	View or change the systems date.
DEBUG	Debug utility to create assembly programs to modify hardware settings.
DEFRAG	Re-arrange the hard disk drive to help with loading programs.
DEL	Deletes one or more files.
DELETE	Recovery console command that deletes a file.
DELTREE	Deletes one or more files and/or directories.
DIR	List the contents of one or more directory.
DISABLE	Recovery console command that disables Windows system services or drivers.
DISKCOMP	Compare a disk with another disk.
DISKCOPY	Copy the contents of one disk and place them on another disk.
DOSKEY	Command to view and execute commands that have been run in the past.
DOSSHELL	A GUI to help with early MS-DOS users.
DRIVPARM	Enables overwrite of original device drivers.
ECHO	Displays messages and enables and disables echo.
EDIT	View and edit files.

EDLIN	View and edit files.
EMM386	Load extended Memory Manager.
ENABLE	Recovery console command to enable a disable service or driver.
ENDLOCAL	Stops the localization of the environment changes enabled by the setlocal command.
ERASE	Erase files from computer.
EXIT	Exit from the command interpreter.
EXPAND	Expand a Microsoft Windows file back to it's original format.
EXTRACT	Extract files from the Microsoft Windows cabinets.
FASTHELP	Displays a listing of MS-DOS commands and information about them.
FC	Compare files.
FDISK	Utility used to create partitions on the hard disk drive.
FIND	Search for text within a file.
FINDSTR	Searches for a string of text within a file.
FIXBOOT	Writes a new boot sector.
FIXMBR	Writes a new boot record to a disk drive.
FOR	Boolean used in batch files.
FORMAT	Command to erase and prepare a disk drive.
FTP	Command to connect and operate on a FTP server.
FTYPE	Displays or modifies file types used in file extension associations.
GOTO	Moves a batch file to a specific label or location.
GRAFTABL	Show extended characters in graphics mode.
HELP	Display a listing of commands and brief explanation.
IF	Allows for batch files to perform conditional processing.
IFSHLP.SYS	32-bit file manager.
IPCONFIG	Network command to view network adapter settings and assigned values.
KEYB	Change layout of keyboard.

LABEL	Change the label of a disk drive.
LH	Load a device driver in to high memory.
LISTSVC	Recovery console command that displays the services and drivers.
LOADFIX	Load a program above the first 64k.
LOADHIGH	Load a device driver in to high memory.
LOCK	Lock the hard disk drive.
LOGON	Recovery console command to list installations and enable administrator login.
MAP	Displays the device name of a drive.
MD	Command to create a new directory.
MEM	Display memory on system.
MKDIR	Command to create a new directory.
MODE	Modify the port or display settings.
MORE	Display one page at a time.
MOVE	Move one or more files from one directory to another directory.
MSAV	Early Microsoft Virus scanner.
MSD	Diagnostics utility.
MSCDEX	Utility used to load and provide access to the CD-ROM.
NBTSTAT	Displays protocol statistics and current TCP/IP connections using NBT
NET	Update, fix, or view the network or network settings
NETSH	Configure dynamic and static network information from MS-DOS.
NETSTAT	Display the TCP/IP network protocol statistics and information.
NLSFUNC	Load country specific information.
NSLOOKUP	Look up an IP address of a domain or host on a network.
PATH	View and modify the computers path location.
PATHPING	View and locate locations of network latency.
PAUSE	Command used in batch files to stop the processing of a command.
PING	Test / send information to another network computer or network

	device.
POPD	Changes to the directory or network path stored by the pushd command.
POWER	Conserve power with computer portables.
PRINT	Prints data to a printer port.
PROMPT	View and change the MS-DOS prompt.
PUSHD	Stores a directory or network path in memory so it can be returned to at any time.
QBASIC	Open the Qbasic.
RD	Removes an empty directory.
REN	Renames a file or directory.
RENAME	Renames a file or directory.
RMDIR	Removes an empty directory.
ROUTE	View and configure windows network route tables.
RUNAS	Enables a user to execute a program on another computer.
SCANDISK	Run the scandisk utility.
SCANREG	Scan registry and recover registry from errors.
SET	Change one variable or string to another.
SETLOCAL	Enables local environments to be changed without affecting anything else.
SETVER	Change MS-DOS version to trick older MS-DOS programs.
SHARE	Installs support for file sharing and locking capabilities.
SHIFT	Changes the position of replaceable parameters in a batch program.
SHUTDOWN	Shutdown the computer from the MS-DOS prompt.
SMARTDRV	Create a disk cache in conventional memory or extended memory.
SORT	Sorts the input and displays the output to the screen.
START	Start a separate window in Windows from the MS-DOS prompt.
SUBST	Substitute a folder on your computer for another drive letter.
SWITCHES	Remove add functions from MS-DOS.

SYS	Transfer system files to disk drive.
TELNET	Telnet to another computer / device from the prompt.
TIME	View or modify the system time.
TITLE	Change the title of their MS-DOS window.
TRACERT	Visually view a network packets route across a network.
TREE	View a visual tree of the hard disk drive.
TYPE	Display the contents of a file.
UNDELETE	Undelete a file that has been deleted.
UNFORMAT	Unformat a hard disk drive.
UNLOCK	Unlock a disk drive.
VER	Display the version information.
VERIFY	Enables or disables the feature to determine if files have been written properly.
VOL	Displays the volume information about the designated drive.
XCOPY	Copy multiple files, directories, and/or drives from one location to another.

11.2.9.2 Managing Input and Output

MS-DOS normally reads input from the standard input, and normally writes output to the standard output. The standard input or output may be redirected to a file or a device. MS-DOS treats devices as files. The output of a command or program can be piped to another command or program. Following DOS commands are relevant for I/O manipulation: CLS, CTTY, FIND, GRAFTABL, GRAPHICS, MODE, MORE, PRINT, SORT, and TYPE.

11.2.9.3 Setting up the Environment

The user can tailor the working environment via the CONFIG.SYS file that defines operating characteristics of an MS-DOS system. When MS-DOS starts, it automatically executes the commands in the CONFIG.SYS file if one is available.

(a) ANSI.SYS - This is a device driver that implements the American National Standards Institute (ANSI) standard escape codes for screen and keyboard control.

(b) BREAK - Control how often MS-DOS checks for a Control-C interrupt. (When MS- DOS encounters a Control-C, it interrupts the current process and returns to the system prompt.)

(c) BUFFERS - Specify the number of in-memory disk buffers that MS-DOS allocates each time it starts.

(d) COUNTRY - Specify the country, thus indicating the appropriate date, decimal sign and currency symbol to be used.

(e) DEVICE - Install a device driver.

(f) DRIVPARM - Redefine the default characteristics defined by a device driver for a block device.

(g) FCBS - Specify the maximum number of FCB-controlled files that can be open while file sharing is in effect.

(h) FILES - Specify the maximum number of open files controlled by file handles.

(i) LASTDRIVE - Determine the maximum number of drives by setting the highest drive letter that MS-DOS recognizes.

(j) SHELL - Specify a command processor to be used in place of COMMAND.COM.

(k) VDISK.SYS-Create a memory-resident virtual disk. This is a file containing the device driver that actually creates the virtual disk in RAM.

(l) VDISK.SYS is placed in the CONFIG.SYS file with the DEVICE command. A virtual disk is much faster than a real disk, but the data is lost when the power is turned off.

(m) CHKDSK reports the storage allocated to the virtual disk.

11.2.9.4 Batch Files

A batch file is a file that contains a number of DOS commands, each of which could be run individually from the command prompt. By putting them into a batch file, they can be run as a group by simply running the batch file. Note that the commands execute in the order they appear in the batch file, and that anything that causes a command to halt will also halt the batch file.

You create a batch file by using an ASCII text editor, such as DOS EDIT, or Windows Notepad. When you have created the batch file, you save it with a file name, and give it the extension *.bat. Note that you must not use a name that is the same as any DOS commands or any other program or utility you are likely to run. If you use a DOS command name, trying to run your batch file will not work because the DOS command will execute first. If your name matches some other program or utility, you may never be able to run that program again because your batch file will run before the program runs. So pick something that is not likely to match any command name or program file name.

Virtually all internal and external commands can be used in a batch file. The few exceptions are the commands that are intended only for configuration which are used in the CONFIG.SYS file. Examples of these include BUFFERS, COUNTRY, DEVICE, etc.

When you create a batch file, you are beginning to write a program, essentially. DOS batch files may not have the power of a structured programming language, but they can be very handy for handling quick tasks. The one good habit for any programmer is to put comments in the program that explain what the program is doing. To do so place REM at the beginning of a comment line. The OS will then ignore that line entirely when it executes the program, but anyone who looks at the "source code" in the batch file can read your comments and understand what it is doing.

To have a batch file executed automatically every time MS-DOS starts, create an AUTOEXEC.BAT file in the root directory. Common commands used in batch file are:

(a) CALL - Call another batch file with parameters.

(b) ECHO - Display command names or messages as commands are executed from a batch file.

(c) FOR - Execute an MS-DOS command iteratively for each file in a set of files.

(d) GOTO - Execute a command in a batch file other than the next in sequence.

(e) IF - Test a condition and execute a command in a batch file, depending on the result.

11.2.9.5 Working with Disks

Formatting disks with the FORMAT command prepares them to be read or written by MS-DOS routines. The FORMAT command simply calls the device driver, which contains the code to perform the formatting of its particular device. Newly installed device drivers must contain the code to format the devices they control.

A volume is a floppy disk or a partition of a fixed disk. Volumes are divided into logical sectors. Logical sectors reside on parts of physical tracks. The tracks are determined by the various fixed positions of the read-write heads. Sectors are grouped into clusters; the file allocation table chains the clusters to define the contents of the file. Each volume has an identifying volume label. Volume labels may be created, modified, or deleted with the LABEL command. Each logical volume includes five control areas and a files area as follows:

(a) Boot sector

(b) Reserved area

(c) File allocation table #1

(d) File allocation table #2

(e) Root directory

(f) Files area

The boot sector contains all the information MS- DOS needs to interpret the structure of the disk. The FAT allocates disk clusters to files, each cluster may contain many sectors, but this number is fixed when the disk is formatted and it must be a power of 2. A file's clusters are chained together in the FAT. There is an extra copy of the FAT is for integrity and reliability. The root directory organizes the files on the disk into directories and subdirectories, The files area is where the files' information is stored. File space is allocated one cluster at a time as needed.

The FORMAT command causes the following information to be placed in the BIOS parameter block (BPB) in the boot sector:

i. sector size in bytes
ii. sectors on the disk
iii. sectors per track
iv. cluster size in bytes
v. number of FATs
vi. sectors per FAT
vii. number of directory entries
viii. number of heads
ix. hidden sectors
x. reserved sectors
xi. media identification code.

The boot sector is logical sector 0 of every logical volume and is created when the disk is formatted. It contains OEM identification, the BIOS parameter block, and bootstrap loader. When the system is booted, a ROM bootstrap program reads in the first sector of the disk (this contains the disk bootstrap) and transfers control to it. The boot strap loader reads MS-DOS BIOS to memory and transfers control to it.

CHKDSK compares the two FATs to make sure they are identical. MS-DOS maintains copy of the FAT in memory to be able to search it quickly. Each cluster entry in the FAT contains codes to indicate the following:

(a) The cluster is available.
(b) The cluster is in use.
(c) The cluster is bad (i.e., it has a bad sector); CHKDSK reports these.
(d) The cluster is a file's last one.

(e) A pointer (cluster number) to the file's next cluster. (The directory points to the file's first cluster.)

Clusters are allocated to files one at a time on demand. Because of the continuing addition, modification, and deletion of files, the disk tends to become fragmented with free clusters dispersed throughout the disk. Clusters are allocated sequentially, but in-use clusters are skipped. No attempt is made to reorganize the disk so that files would consist of sequential clusters, but this can be accomplished via the MS-DOS commands.

MS-DOS uses memory buffers with disk input/output. With file handle calls, the location of the buffer may be specified in the call. With FCB calls, MS-DOS uses a preset buffer called the disk transfer area, which is normally in a program's program segment and is 118 bytes long.

11.2.9.6 Working with Directories

Initial versions of MS-DOS had a simple linear directory listing all files. Version 2.0 incorporated a hierarchical file structure with directories and subdirectories. A root directory and subdirectories form the hierarchical structure, and directory entries describe individual files.

The 32-byte directory entry contains the following:

i. File name (8 bytes) Extension (3 bytes)
ii. File attribute byte (1 byte)
iii. Reserved (10 bytes)
iv. Time created or last updated (2 bytes)
v. Date created or last updated (2 bytes)
vi. Starting cluster (2 bytes)
vii. File size (4 bytes)

The first byte of the file name may contain special codes indicating the following:

i. A directory entry that has never been used
ii. The entry is a subdirectory
iii. File has been used, but is now erased.

The file attribute byte may indicate the following:

i. Read-only file (an attempt to open the file for writing or deletion will fail)
ii. Hidden file (excluded from normal searches)
iii. System file (excluded from normal searches)
iv. Volume label (can exist only in the root directory)
v. Subdirectory (excluded from normal searches)
vi. Archive bit (set to "on" whenever a file is modified)

Each disk is formatted to have at least one directory, but there can be many directories. A directory may include other directories. Formatting creates an initial root directory. Each fixed disk partition has its own root directory. With versions 2.0 and higher, the root directory may contain directories in addition to files. Directories may contain one or more file entries or more directories, thus generating a hierarchical file system directory structure. The volume label, if present, appears in the root directory.

MS-DOS function calls are available to search directories for files in a hardware independent manner. FCB functions require that files be specified by a pointer to an unopened FCB; these functions do not support the hierarchical file system. With the file handle function request, it is possible to search for a file by specifying an ASCII string; it is possible to search within any subdirectory on any drive, regardless of the current subdirectory, by specifying a pathname.

11.2.9.7 Managing Files

Data is organized into files whether that data is in memory or on disk. Files may be data files or executable files. A file is simply a string of bytes; no record structure is assumed. Applications impose their own record structure on the string of bytes MS-DOS requires only a pointer to the data buffer, and a count of the number of bytes to be read or written in order to do I/O. Executable files must be in either the .COM format or the .EXE format. Object modules are maintained in the Intel Corporation object-record format.

Files may be accessed with FCB calls or file handle calls. File handle calls are designed to work with a hierarchical file system. File handle calls are preferable, but FCB calls are provided for compatibility with previous versions of DOS. File handle calls support record locking and sharing. Users interested in writing programs that will be compatible with future versions of MS-DOS should use handle calls rather than the FCB calls.

The FCB function requests are as follows:

i. Open file with FCB
ii. Close file with FCB
iii. Delete file with FCB
iv. Read block sequentially
v. Write block sequentially
vi. Create file with FCB
vii. Rename file with FCB
viii. Read block directly
ix. Write block directly
x. Get file size with FCB
xi. Set relative record field
xii. Read multiple blocks directly
xiii. Write multiple blocks directly

The FCB data structure occupies a portion of the application's memory and contains bytes of data as follows:

i. Drive identifier (1 byte) (0=default drive, 1=drive A, 2=drive B, and so on)
ii. File name (8 bytes)
iii. Extension (3 bytes)
iv. Current block number (2 bytes) (for sequential reads and writes)
v. Record size (2 bytes) (MS-DOS sets to 128, but this may be changed.)
vi. File size (4 bytes)
vii. Date created/updated (2 bytes)
viii. Time created/updated (2 bytes)
ix. Reserved (8 bytes)
x. Current record number (1 byte) (for sequential reads and writes)
xi. Relative record number (4 bytes)

A file handle is a 16-bit integer (usually 0 to 20) created by MS-DOS and returned to a program that creates or opens a file or device. A file may be opened with a handle by using a pathname and attribute. The program saves the handle and uses it to specify the file in future operations on the file. An FCB-like data structure is built by MS-DOS for the file, but this is strictly controlled by the operating system. The file handle function calls are as follows:

(a) Create file
(b) Open file
(c) Close file
(d) Read from file or device
(e) Write to file or device
(f) Move file pointer

(g) Duplicate file handle (creates a new handle that refers to the same file as an existing handle)
(h) Match file handle (cause one handle to refer to the same file as another)
(i) Create temporary file
(j) Create file if name is unique (fails if a file with the same name already exists)

MS-DOS maintains a table that relates handles to files or devices. Eight handles are normally available to programs, but this can be increased to 20 via the CONFIG.SYS file. Five of the handles are pre-assigned to standard devices as follows:

Handle	Standard Device	Device Name	Description
0	Standard input device	CON	reads char from the keyboard
1	Standard output device	CON	writes characters to the VDU
2	Standard error device	CON	writes characters to the VDU
3	Standard auxiliary device	AUX	controls serial port I/O
4	Standard printer device	PRN	controls parallel port output

A COM file consists of absolute machine code and data. It is not relocatable. It loads faster than an .EXE file. An .EXE file is a relocatable and executable file; it contains a header with relocation information, and the actual load module. The .EXE files separate the code and data portions. This feature is designed to facilitate the sharing of “pure procedures” among several tasks in evolving MS-DOS systems that support concurrent tasks. The Microsoft Macro Assembler translates source code files into object files with the .OBJ extension.

11.2.9.8 Device drivers

Device drivers are programs that control input and output. They manage communication between the hardware devices and the operating system. Regardless of the specific details of performing input/output on particular devices, device drivers communicate in standard manner with the rest of the operating system. Five resident device drivers control the standard devices. Other device drivers may be installed at the command level as needed. The two types of devices are character devices that handle one byte at a time, and block devices that can access blocks of information on random access devices such as disks. Devices are treated as files. The IO.SYS file consists of the five resident device drivers that form the MS-DOS BIOS.

New or replacement device drivers may be installed by using the DEVICE command followed by the driver's file name in a CONFIG.SYS file on the boot disk. Thus, the input/output system may be reconfigured at the command level. MS-DOS has character device drivers and block device drivers. CON, AUX, and PRN are character device drivers. The CONFIG.SYS file may be used to notify the operating system of hardware device changes such as additional hard disks, a clock chip, a RAM disk, or additional memory.

A device driver is composed of a device header, a strategy routine, and an interrupt routine. The device header contains a pointer to the next device driver in the chain of device drivers in IO.SYS, and it points to the strategy routine and the interrupt routine. The request header is the data structure that MS-DOS uses to give the driver the information necessary to perform a requested input/output operation. The strategy routine saves a pointer to the request header data structure. The interrupt routine performs the I/O, passes status and completion information back to MS-DOS in the request header. Each driver has a strategy routine entry point and an interrupt routine entry point. For asynchronous I/O, the strategy routine is called upon to enqueue a request and return to the caller quickly. The interrupt routine then performs the I/O when it can.

A filter is a program that processes input data in some particular way to produce output directed to some file or device. Three filters are built in to MS-DOS. SORT sorts text data. FIND searches for a character string. MORE displays one screen of data at a time. Other filters may be created. Filters may be piped together (with the | symbol) so that the output of one forms the input to another. Inputs or outputs may be redirected.

Background programs are dormant until they are activated by a signal from the keyboard. A hot key, or combination of keys, signals a program to take control of keyboard input.

11.2.9.9 Memory Management

Memory is organized as follows in MS-DOS (from low memory to high memory locations):

i. Interrupt vector table
ii. Optional extra space (used by IBM for ROM data area
iii. IO.SYS
iv. MSDOS.SYS
v. Buffers, control areas, and installed device drivers
vi. Resident part of COMMAND.COM
vii. External commands or utilities (.COM and .EXE files are loaded here)
viii. User stack for .COM Files (256 bytes)
ix. Transient part of COMMAND.COM

The interrupt vector table contains the addresses of the interrupt handler routines. IO.SYS is the basic input/output system; it is the MS-DOS/hardware interface. MSDOS.SYS contains most of the interrupt handlers and function requests. The resident part of COMMAND.COM contains certain interrupt handlers and the code that reloads the transient part of COMMAND.COM as needed; the transient part of COMMAND.COM includes the batch processor, the internal commands, and the command processor.

MS-DOS begins the user's program segment in the lowest address free memory. The program segment prefix (PSP) occupies the first 256 bytes of the program segment area. The PSP points to various memory locations the program requires as it executes.

MS-DOS creates a memory control block at the start of each memory area it allocates. This data structure specifies the following:

i. the size of the area
ii. the program name (if a program owns the area)
iii. a pointer to the next allocated area of memory

MS-DOS may allocate a new memory block to a program, free a memory block, or change the size of an allocated memory block. If a program tries to allocate a memory block of a certain size, MS-DOS searches for an appropriate block. If such a block is found, it is modified to belong to the requesting process. If the block is too large, MS-DOS parcels it into an allocated block and a new free block. When a block of memory is released by program, MS-DOS changes the block to indicate that it is available. When a program reduces the amount of memory it needs, MS-DOS creates a new memory control block for the memory being freed. The first memory block of a program always begins with program segment prefix. Normally when a program terminates, its memory is released. The program can retain its memory by issuing function 31, TERMINATE BUT STAY RESIDENT.

11.2.9.10 Controlling Program Flow

COMMAND.COM uses the EXEC function to load and execute program files. A program issuing EXEC causes MS-DOS to allocate memory, write a program segment prefix, loads the new program, and transfer control to it. The calling program is the parent program; the called program is the child program. A child program may use EXEC to load and execute its own child programs. Each child automatically inherits its parent's active handles, so it can access its parent's active files. MS-DOS allows programs to load and execute overlays.

MS-DOS handles hardware interrupts from devices and software interrupts caused by executing instructions. An interrupt vector table causes control to transfer to the appropriate interrupt handlers. Users may write their own interrupt handlers. The Control-C interrupt normally terminates the active process and returns control to the command interpreter.

Certain types of critical errors may occur that prevent a program from continuing. For example, if a program tries to open a file on a disk drive without a disk, or a disk drive whose door is open, a critical error is signaled. MS-DOS contains a critical error handler, but user users may wish to provide their own routines for better control over errors in specific situations.

11.2.10 SYSTEM CALLS - THE PROGRAMMER'S VIEW

MS-DOS provides the programmer with a rich collection of system calls. These calls accomplish various manipulations that would be extremely time-consuming and difficult to program and debug if the operating system were not available. The calls manipulate files, disks and directories; return information about the hardware and software environment; and obtain additional memory and relinquish memory no longer needed. By using MS-DOS calls, programmers realize several advantages. Applications programming becomes faster and less error prone, and applications that use system calls are easier to upgrade to new versions of the operating system. Microsoft is committed to providing upward compatibility for programs that use system calls properly. The next several sections enumerate most of MS-DOS's system calls.

11.2.10.1 Working with Disks

- Check Status of Verify Flag - Returns the value of the Verify Flag, which determines whether or not DOS verifies write operations to the disk.
- Get Default Drive Data - Returns information about the current default drive, namely the number of sectors per cluster, information about the type of disk in the current default drive, bytes per sector, and clusters per drive.
- Get Disk Transfer Address - Returns the pointer to the current Disk Transfer Area (the DT A is the buffer that DOS uses to transfer data to-and-from the disk).
- Get Free Disk Space - Determines how many bytes are free on the specified drive. Get Specified Drive Data-Returns information about the specified drive, namely the number of sectors per cluster, the offset to the FAT ID byte, the number of bytes per sector, and the number of clusters per drive.
- Identify Current Drive - Returns the default drive ID.
- Read Absolute Disk Sectors - Reads the specified number of disk sectors from the disk drive, starting at a given location.
- Reset Disk - Flushes all buffers to the media and marks them "free."
- Select Drive - Sets the default drive to the specified drive.
- Set Disk Transfer Address - Sets the Disk Transfer Address to the specified value.
- Set or Reset Verify Flag - Sets the value of the Verify Flag to a specified value. The Verify Flag determines whether or not DOS verifies write operations to the disk.
- Write Absolute Disk Sectors - This interrupt handler writes the specified number of sectors to the disk drive, starting at a given location.

11.2.10.2 Working with Directories

- Change Current Directory - Changes the current directory to the directory in the user's pathname.
- Create Directory - Creates a directory using the name in the user's pathname.
- Get Current Directory - Returns the pathname of the current directory on a

specified drive.

- Remove Directory - Removes the directory specified in the user's pathname.

11.2.10.3 Managing Files

- Change File Attributes - Gets or sets the attributes of the file specified in the user's pathname.
- Close File - Closes the specified handle.
- Close File with FCB - Closes the file pointed to by the user's FCB.
- Create File-Creates and assigns a handle to the file in the user's pathname.
- Create File if Name is Unique - Creates a new file if a file by that name does not exist in the specified directory.
- Create File with FCB - Creates a file in the current directory using the file name in the user's FCB.
- Create Temporary File - Uses the clock device to provide a unique file name and appends it to the pathname provided by the user.
- Delete File - Deletes the directory specified in the user's pathname.
- Delete File with FCB - Deletes a file named in the user's FCB, and removes the specified file from the directory buffer for the drive specified in the user's FCB.
- Duplicate File Handle - Copies the handle of an open file.
- Find First Matching File - Searches the specified' or current directory for the first entry that matches the user's pathname.
- Find First Matching File with FCB - Searches the current directory for the first file name matching the file name contained in the user's FCB.
- Find Next Matching File - Searches for the next entry that matches the name and attributes specified in a previous Find First Matching File.
- Find Next Matching File with FCB - Searches for the file named in the user's FCB. The file name was used originally to conduct the "Find First Matching File with FCB" function.
- Get File Size with FCB - Returns the size of the file specified in the FCB.
- Lock or Unlock Region of File - Lock (deny access to) or unlock a specified region of a file.

- Match File Handle - Forces a specified handle to refer to the same file as another handle already associated with an open file.
- Move File Pointer - Moves the read/write pointer of the open file associated with the specified handle.
- Open File - Opens a file and assigns it a handle using the file name specified in the user's pathname.
- Open File with FCB - Opens the file named in the user's FCB.
- Parse File Name - Searches the specified string for a valid name. If the file name is valid, the function returns a pointer to an unopened FCB.
- Read Block Directly - Reads the record pointed to by the user's FCB Relative-Record field.
- Read Block Sequentially - Reads the next block from the file named in the user's FCB.
- Read from File or Device - Reads from the specified open file handle.
- Read Multiple Blocks Directly - Reads a specified number of records from the file starting at the block named in the user's FCB relative-record field.
- Rename File - Renames the file specified in the user's pathname by changing the directory entry.
- Rename File with FCB - Renames a file in the current directory using the two file names in the user's modified FCB; the first name is the file in the current directory, and the second is the new name for the file.
- Set Relative Record Field - Sets the relative record field in the user's FCB to the value' and of the current record field in the FCB. The relative record field is used by the Read (or Write) Block Directly functions.
- Write Block Directly - Writes the record pointed to by the relative record field (in the user's FCB), to the file named in the user's FCB.
- Write Block Sequentially - Writes to the file named in the user's FCB. The FCB also contains the current block and the record to be written.
- Write Multiple Blocks Directly - Writes a specified number of records to the file named in the user's FCB. The FCB also contains the current relative record number where writing will start.

11.2.10.4 Managing Input and Output

- Check Keyboard Status - Returns a code to indicate if characters are available from standard input.
- Display Character - Sends the specified character to standard output. Display String-Sends the specified string to standard output.
- I/O Through Standard Device - Either reads or sends a character to standard output, depending on the specified value.
- Input from Auxiliary Device - Waits for a character to be read from standard al and returns the character.
- IOCTL - The IOCTL function is a group of 16 related sub functions that manipulates character and block-device control data. These sub functions can be used to set or reset the attributes of a certain device, send or receive control data with either character or block devices, check the status of a device, or check if a device or file handle has been redirected with Microsoft Networks.
- Output to Auxiliary Device - Sends the specified character to the standard auxiliary device.
- Print Character - Sends the specified character to the standard printer device.
- Read Keyboard - Waits for a character to be read from the standard input device and returns the character.
- Read Keyboard after Clearing Buffer - First clears the input buffer and then executes the specified function.
- Read Keyboard and Echo - Waits for a character to be read from the standard device, then echoes it to the standard output device and returns the character.
- Read Keyboard to Buffer - Gets a string (terminated with a carriage return) from standard input device and puts it in the user-specified buffer.
- Read Keyboard without Echo - Waits for a character to be read from the standard put device and returns the character.

11.2.10.5 Managing Memory

- Allocate Memory - Attempts to allocate the specified amount of memory to current process. DOS coalesces disowned blocks, if necessary, to provide enough for the user's requested block size.

- Change Memory Allocation Block Size - Puts the user's specified block back into memory pool, coalesces it with other unowned blocks, and then tries to allocate requested block size; the function returns the size of the resultant block to the user.
- Free Allocated Memory - Releases the specified block of memory.
- Get or Set Memory Allocation Strategy - Gets or sets the strategy used by MS-DOS allocate memory when a process requests it; the three different strategies are first best fit, and last fit.

11.2.10.6 Controlling Program Flow

- Abort Program - Aborts the current process.
- Check Control-C - Either returns or sets DOS's internal Control-C checking flag; the Control-C flag is used by DOS to determine before which function requests it should check for a Control-C.
- Create New Program Segment Prefix - Creates a new program segment prefix (PSP) at the specified address. (This is an obsolete function except for compatibility with pre-2.0 versions of DOS.)
- Execute Program - Loads and executes a program.
- Get Current Program Segment Prefix - Returns the contents of the DOS "Current PSP" variable. (A PSP is a block of memory that immediately precedes a .COM or .EXE program.)
- Get Extended Error Codes - Returns an extended error code for the function that immediately preceded it. The extended error code contains details about the type and location of the error, and the recommended course of action to take when the error occurs.
- Get Interrupt Vector - Returns the address of the specified interrupt handler.
- Get Return Code of Child Process - Returns the exit code specifying the reason for termination, such as normal, Control-C, or critical device error.
- Set Interrupt Vector - Sets the address of the specified interrupt in the Interrupt

Vector Table.

- Terminate But Stay Resident - Returns control to the parent process (usually COMMAND.COM), but keeps the current process resident after it terminates.
- Terminate But Stay Resident - Terminates a program of up to 64K in size, but keeps it in memory. (This interrupt is obsolete.)
- Terminate on Control-C - Used by DOS to deal with a Control-C being typed on the keyboard.
- Terminate on Fatal Error - Used by DOS if a critical error occurs during I/O.
- Terminate Process - This is the proper way for a program to terminate when finished. Terminate Program - Terminates the current process and returns control to the parent process. (This is an obsolete interrupt.)
- Terminate Program at Address - Used by DOS to transfer control after a process terminates.

11.3 Keywords

Batch file: It is an executable file that contains a group of commands that the user wants to execute in sequence.

Device drivers: These are programs that control input and output.

Internal commands: These are commands, such as COPY and DIR that can be handled by the COMMAND.COM program.

11.4 Summary

MS-DOS is a single-user, single-process operating system. Due to confinement of device-independent code into lone layer, porting of MS-DOS is theoretically reduced to writing of the BIOS code for the new hardware. Although early versions of MS-DOS show resemblance to the CP/M operating system, later releases of MS-DOS have Unix-like features. At the command level, MS-DOS provides a hierarchical file system, 1/0 redirection, pipes and filters. User-written commands can be invoked in the same way as standard system commands, thus giving the appearance of extending the basic system functionality.

MS-DOS provides both device-dependent and device-independent versions of system calls for input/output and file manipulation. Being a single-user system, MS-DOS provides only rudimentary file protection and access control. Disk space is allocated in terms of clusters of consecutive sectors. A variant of chaining that allows for relatively fast random access to files is used for keeping track of both file blocks and free space.

11.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. Discuss history and different versions of MS-DOS.
2. Discuss various internal and external MS-DOS commands.
3. Discuss memory management in MS-DOS.
4. What do you understand by batch files? Discuss some common commands used in it.
5. Differentiate between CONFIG.SYS and AUTOEXEC.BAT files.
6. What are the limitations of DOS? Explain.
7. What do you understand by Terminate and Stay Resident programs (TSR)? Explain.

11.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. The Design of the UNIX Operating System, Bach M.J., PHI, New Delhi, 2000.
2. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
3. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
5. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.

6. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson Number: 12 ***Writer: Dr. Rakesh Kumar***

Case Study of MS-WINDOWS NT ***Vetter: Prof. Dharminder Kr.***

12.0 Objectives

The objectives of this lesson are:

(a) To provide a brief overview of the history of Windows operating system.

(b) To discuss the key features of Windows NT operating system.

12.1 Introduction

Microsoft Windows is the name of several families of software operating systems by Microsoft. Microsoft first introduced an operating environment named Windows in November 1985 as an add-on to MS-DOS in response to the growing interest in graphical user interfaces (GUI). The term Windows collectively describes several generations of Microsoft (MS) operating system (OS) products categorized as follows:

12.1.1 16-bit operating environments

The early versions of Windows were often thought of as just graphical user interfaces, because they ran on top of MS-DOS and used it for file system services. However even the earliest 16-bit Windows versions have many typical operating system functions, such as having their own executable file format and providing their own device drivers for applications. Unlike MS-DOS, Windows allowed users to execute multiple graphical applications at the same time, through cooperative multitasking. Finally, Windows implemented an elaborate, segment-based, software virtual memory scheme which allowed it to run applications larger than available memory: code segments and resources were swapped in and thrown away when memory became scarce, and data segments moved in memory when a given application had left processor control, typically waiting for user input. 16-bit Windows versions include Windows 1.0, Windows 2.0 and Windows/286.

12.1.2 Hybrid 16/32-bit operating environments

Windows/386 introduced a 32-bit protected mode kernel and virtual machine monitor. For the duration of a Windows session, it created one or more virtual 8086 environments and provided device virtualization for the video card, keyboard, mouse, timer and interrupt controller inside each of them. The user-visible consequence was that it became possible to preemptively multitask multiple MS-DOS environments in separate Windows. Windows applications were still multi-tasked cooperatively inside one of such real-mode environments.

Windows 3.0 and Windows 3.1 improved the design, because of virtual memory and loadable virtual device drivers which allowed them to share arbitrary devices between multitasked DOS windows. Because of this, Windows applications could now run in 16-bit protected mode, which gave them access to several megabytes of memory and removed the obligation to participate in the software virtual memory scheme. They still ran inside the same address space, where the segmented memory provided a degree of protection, and multi-tasked cooperatively

12.1.3 Hybrid 16/32-bit operating systems

With the introduction of 32-bit Windows, Windows finally stop relying on DOS for file management. Windows 95 introduced Long File Names. The most important novelty was the possibility of running 32-bit multi-threaded preemptively multitasked graphical programs. However, the necessity of keeping compatibility with 16-bit programs meant the GUI components were still 16-bit only and not fully reentrant, which resulted in reduced performance and stability.

Microsoft's next OS was Windows 98; there were two versions of this. In 2000, Microsoft released Windows ME, which used the same core as Windows 98 but adopted the visual appearance of Windows 2000, as well as a new feature called System Restore, allowing the user to set the computer's settings back to an earlier date.

12.1.4 32-bit operating systems

This family of Windows systems was designed for higher-reliability business use. The first release was Windows NT 3.1, followed by NT 3.5, NT 3.51, and NT 4.0.

Microsoft then moved to combine their consumer and business operating systems. Their first attempt, Windows 2000, failed to meet their goals, and was released as a business system. The home consumer edition of Windows 2000, codenamed "Windows Neptune," ceased development and Microsoft released Windows Me in its place. Eventually "Neptune" was merged into their new project, Whistler, which later became Windows XP.

12.1.5 64-bit operating systems

Windows NT included support for several different platforms before the x86-based personal computer became dominant in the professional world. Versions of NT from 3.1 to 4.0 supported DEC Alpha and MIPS R4000, which were 64-bit processors, although the operating system treated them as 32-bit processors.

With the introduction of the Intel Itanium architecture, Microsoft released new versions of Windows 2000 to support it. Itanium versions of Windows XP and Windows Server 2003 were released at the same time as their mainstream x86 (32-bit) counterparts. On April 25, 2005, Microsoft released Windows XP Professional x64 Edition and x64 versions of Windows Server 2003 to support the AMD64/Intel64 architecture. Microsoft dropped support for the Itanium version of Windows XP in 2005. Windows Vista is the first end-user version of Windows that Microsoft has released simultaneously in 32-bit and x64 editions.

12.2 Presentation of contents

12.2.1 General Architecture

12.2.1.1 Extensibility

12.2.1.2 Portability

12.2.1.3 Reliability and Security

12.2.2 The Hardware Abstraction Layer (HAL)

12.2.2.1 Objects

12.2.2.2 Threads

12.2.2.3 Traps, Interrupts, and Exceptions

12.2.2.4 Thread Scheduling

12.2.3 Multiprocess Synchronization

12.2.3.1 The NT Executive
12.2.3.1.1 **Object Manager**
12.2.3.1.2 Process and Thread Manager
12.2.3.1.3 Virtual Memory Manager
12.2.3.1.4 Security Reference Manager
12.2.3.1.5 I/O Manager
12.2.3.1.6 Cache Manager
12.2.3.1.7 The Native API
12.2.4 NT Subsystems
12.2.4.1 Win 32 API: The Programmer's View of NT

12.2.1 General Architecture

The product goals for Windows NT were that it should be an extensible, portable, reliable, and secure OS for contemporary computers.

12.2.1.1 Extensibility

There are two dimensions to the extensibility aspect. The first relates to OS configurations. A Windows NT machine can be configured for a workstation or a server. In either configuration, the OS uses the same fundamental source code, but different components are incorporated into each at compile time. This allows Windrows NT to be optimised to perform best according to the way the machine will be used-as a workstation or as a server-without building two different OSs.

The second, aspect of extensibility is in the way the OS software is structured. Windows NT is designed using an extensible nucleus software model. In this approach, only the most essential OS functions are implemented in a small nucleus of code (microkernel). Additional mechanisms are then implemented on top of the nucleus to define policy as needed. This approach has the advantage that key mechanisms can be carefully designed and tested as one trusted subassembly that can then be used to implement many different policies. This is a basic approach to support the goals of good security and reliable operation. The NT Kernel provides these essential low-level mechanisms as a layer of abstraction from the hardware (see Figure 12.1). The NT Executive is designed as a layer of abstraction of the NT Kernel. It provides specific mechanisms for general object and memory management, process management, file management, and device management. Together, the NT Kernel and the NT Executive provide the essential elements of the OS.

The NT Kernel and the NT Executive are combined into a single executable, NTOSKRNL.EXE, before they are actually executed. NTOSKRNL.EXE also invokes additional dynamically linked libraries (DLLs) whenever they are needed. Thus the logical view of Windows NT is quite different from the way the executable actually appears in memory. It is best to use the logical view for considering different aspects of the OS, since that is the model under which it is designed.

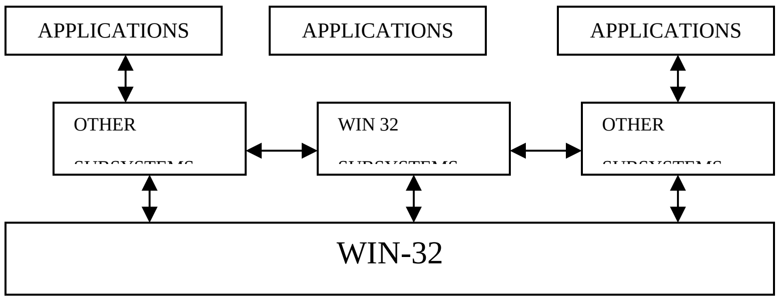


Figure 12.1 NT Organization

The next layer of abstraction of Windows NT is the subsystem layer. Subsystems provide application portability for windows software. An NT Subsystem is a software module that uses the services implemented in the Kernel and the Executive to implement more abstract services, especially the services offered by some target OS. For example, Version 4.0 has a POSIX subsystem that executes on top of the Kernel and the Executive that makes Windows NT look like POSIX; such subsystems are called environment subsystems. Other subsystems implement specialized services such as the security subsystem. All subsystems (and all application programs that use the subsystems) execute when the processor is in user mode. Subsystems are the key component in allowing Microsoft to support various computational models, such as the MS-DOS and Win 16 program models. Application programs written to run on MS-DOS use the MS-DOS subsystem interface. This subsystem provides the same API to the application, as does MS-DOS, thereby allowing old MS-DOS programs to run on a Windows NT system.

12.2.1.2 Portability

The portability aspect of Windows NT overlaps its extensibility. Subsystems allow Windows NT to be extended to meet various application support requirement. Microsoft has built various subsystems to implement OS personalities of interest to their customers. Besides the MS-DOS subsystem, there are subsystems to support Win 16 applications and POSIX programs, as well as a new Win32 subsystem. In general, it is possible for software developers to implement any subsystem to satisfy their general requirements for OS service; such a subsystem uses the Executive/Kernel interface. Even so, the Win32 subsystem takes a special role in Windows NT because it implements various extensions of the NT Executive that are needed by all other subsystems; every subsystem relies on the presence of the Win32 subsystem. While it is possible to add new environment subsystems to Windows NT, and to omit most of them, the Win32 subsystem must always be present.

Another aspect of portability is the ability to port Windows NT across different hardware platforms. Microsoft's goal was to be able to reuse the Kernel, Executive, and subsystems on new microprocessors as they became available without having to rewrite the Kernel. Windows NT's designers carefully identified the things that were common across a wide set of microprocessors and the things that were different. This allowed them to create a hardware abstraction layer (HAL) software module to isolate the Kernel from hardware differences. The HAL is responsible for mapping various low-level, processor-specific operations into a fixed interface that is used by the Windows NT Kernel and Executive. The HAL also executes with the processor in supervisor mode.

The HAL, Kernel, and Executive are supervisor-mode software that collectively export an API that is used by subsystem designers (but not by application programmers). Environment subsystem designers choose a target API (such as the Win 16) and then build a subsystem to implement the API using the supervisor portion of Windows NT. Microsoft has even chosen its own preferred API-the Win32 API-which is also the API for the Win32 subsystem. Windows NT application programs are written to work on the Win32 API rather than on the interface to NTOSKRNL.EXE.

12.2.1.3 Reliability and Security

Separating the HAL, Kernel, Executive, and subsystem functionality from one another, thus eliminating unnecessary interactions, supports reliability. Windows NT is designed to meet standard requirement for trusted OSs. According to Solomon, in 1995 United Kingdom Information Technology Security Evaluation and Certification Board certified Windows NT at the C2 level by the United States National Computer Security Centre and in 1996 at F-C2/E3 level. Much of the security mechanism is implemented in a Security Subsystem that depends upon the Security Reference Manager in the Executive.

12.2.2 The Hardware Abstraction Layer (HAL)

The HAL is a low-level software module that translates critical hardware behaviours into a standardized set of behaviours. The HAL functions are exported through a kernel-mode DLL, HAL.DLL. The OS calls functions in HAL.DLL when it needs to determine the way the host hardware behaves. This allows the Windows NT code to call a HAL function everywhere a hardware specific address is needed. For example, device interrupts usually have addresses determined by the microprocessor architecture, and they differ from one microprocessor to another. The HAL interface allows Windows NT to reference the interrupt addresses via functions rather than by using the hardware addresses directly.

The HAL implementation for any specific microprocessor provides the appropriate hardware-specific information via the corresponding function on the HAL API. This means that it is possible to use the same source code on a Digital Equipment Alpha processor as is used on an Intel Pentium processor. It also means that it is possible to create device drivers for Windows NT that will also work without change in Windows 9x.

The use of the HAL is transparent above the Executive/Kernel interface. Subsystem and application programmers are generally unconcerned with the type of processor chip in the computer. Windows NT provides a fixed set of services independent of the hardware platform type.

The NT Kernel creates the basic unit of computation and provides the foundation for multitasking support. It does so without committing to any particular policy/strategy for process management, memory management, file management, or device management. To appreciate the level of support the Kernel provides, think of the Kernel as offering a collection of building components such as wheels, pistons, lights, and so on, that could be used to build a sports car, a sedan, a sports utility vehicle, or a truck. Similarly, the Kernel's clients can combine the components to build a compound component that defines a policy for how the low-level components are used.

The Kernel provides objects and threads on top of the HAL and the hardware. Software that uses the Kernel can be defined using objects and threads as primitives, that is, these abstractions appear to Kernel client software as natural parts of the hardware. To implement objects and threads, the Kernel must manage the hardware interrupts and exceptions, perform processor scheduling, and handle multiprocessor synchronization.

12.2.2.1 Objects

The NT Kernel defines a set of built-in object types. Some kernel object types are instantiated by the Kernel itself to form other parts of the overall OS execution image. These objects collectively save and manipulate the Kernel's state. Other objects are instantiated and used by the Executive, subsystems, and application code as the foundation of their computational model. That is, Windows NT and all of its applications are managed at the Kernel level as objects.

Kernel objects are intended to be fast. They run in supervisor mode in a trusted context, so there is no security and only limited error checking for Kernel objects, in contrast to normal objects, which incorporate these features. However, Kernel objects cannot be manipulated directly by user-mode programs, only through function calls. Kernel objects are characterized as being either control objects or dispatcher objects.

A) Control Objects

Control objects implement mechanisms to control the hardware and other Kernel resources. When an application program creates a new process, it requests that the Kernel create a type of control object, a new process object; the OS returns a handle to the object after it has been created. The application refers to the object by using the handle. When the application program manipulates the process, it is manipulating the underlying kernel process object. There are other control objects. The asynchronous procedure call (APC) object is an object that can be used to interrupt another thread and cause it to execute a specific procedure. An interrupt object is an object created to match up to each interrupt source so that when a designated interrupt occurs, the corresponding object will receive a message. A profile object is an object that can be used to monitor the amount of time a thread spends executing different parts of the code. Other control objects exist to handle power failure when it occurs, to check to see if power has failed, to do power management, and so on.

B) Dispatcher Objects

Dispatcher objects are used to implement threads along with their scheduling and synchronization operations. Each dispatcher object has built-in characteristics that are used to support user-level synchronization. A process object creates a computational abstraction that can have an address space and a set of resources. However, in Windows NT the process object cannot execute. A dispatcher object called a thread object is the active element of the computation abstraction; it has its own stack and can execute within a process. Whenever any application program is to be executed, it must have an associated process object and a thread object. Other dispatcher objects are used primarily to implement one form or another of synchronization. While a single synchronization primitive as this would be sufficient (ultimately, all synchronization is based on the Kernel spin-lock described shortly), the Executive will implement a set of variants to simplify the use of dispatcher objects for synchronization.

12.2.2.2 Threads

A thread is an abstraction of computation. A Windows NT process object defines an address space in which one or more threads can execute, and each thread object represents one execution within the process. In a UNIX environment, there can be only one thread executing in each address space. A UNIX process does not differentiate between the address space concepts of a process object and the execution aspects of a thread object. In the Windows NT environment it is common to have more than one thread - a logical path traversal through the code in an address space - executing in a process. The separation of the thread concept from the rest of the process concept has been done so that it is natural to think of several different "threads of execution" within a single address space, all sharing the same resources.

12.2.2.3 Traps, Interrupts, and Exceptions

In Windows NT terminology, the Kernel trap handler is responsible for reacting to hardware interrupts and processor exceptions (such as system service call, execution errors, and virtual memory faults). Whenever an interrupt or processor exception is recognized by the hardware, the trap handler (see Figure 12.2) moves into action. It is responsible for doing the following:

- Disabling interrupts
- Determining the cause of the interrupt or exception.
- Saving processor state in a trap frame
- Re-enabling interrupts
- Changing the processor to supervisor mode if required
- Dispatching specialized code, for example, an Interrupt Service Routine (ISR), a DLL, an exception dispatcher, or the virtual memory handler, to handle the trap.

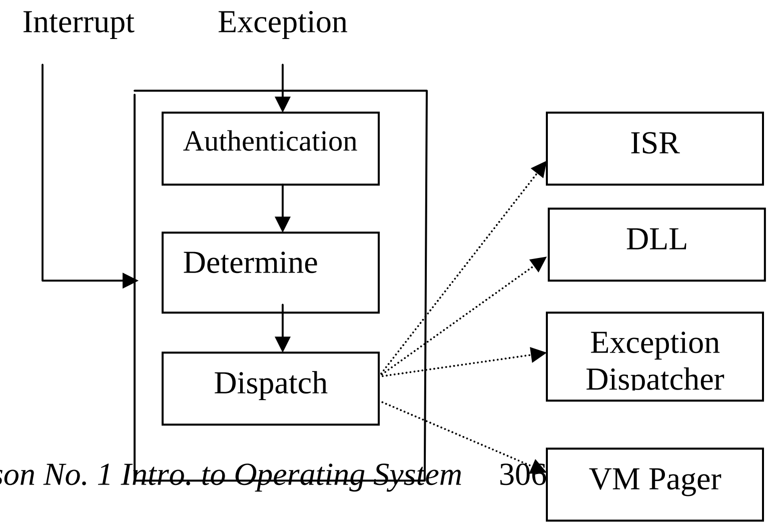


Figure 12.2 Trap Handler

In the case of an interrupt, the trap handler will normally run an ISR for the specific interrupt. For exceptions, the trap handler might address the cause itself or invoke the appropriate OS code to react to the exception.

As in all "system call interface" OS designs, supervisor-mode functions are invoked by an application program when it executes an instruction that causes exception; in many hardware sets, this is the trap instruction. The trap handler must be used to call system functions, since the processor mode needs to be switched from user to supervisor. Before the mode can be switched, the OS must be assured that the code to be executed (while the hardware is in supervisor mode) is trusted code. Therefore user programs are not allowed to link and call these functions directly. Instead, they can be invoked only through the trap handle in Windows NT, the trap handler uses a DLL, NTDLL.DLL, to authenticate the call and start the OS code; the application links NTDLL.DLL into its address space and then calls entry points in the DLL. These points are translated into traps (using the host hardware mechanism for raising an exception) that cause the processor mode to be switched to supervisor mode and a secure call to be made on the OS code.

Interrupts are used to allow a device to notify the OS when the device completes an operation. Windows NT's Interrupt management generally follows the same design that other OSs have used for a number of years. The device’s driver initiates each device operation. The thread initiating the operation might wait for the I/O call to complete (said to be a synchronous I/O call) or continue running concurrently with the I/O operation (said to be an asynchronous I/O call). Traditionally, the API does allow the application thread to use asynchronous I/O, though asynchronous I/O is fully supported in Windows NT. The API used with Windows NT extends the normal C routines so that application program can use asynchronous I/O operations.

In either the synchronous or asynchronous case, the processor continues to execute software concurrent with the device operation - the calling thread's code, in the asynchronous case, or another thread's code, in the synchronous case. The device will eventually signal the processor that it has completed the I/O operation by raising an interrupt. This causes the trap handler to run and to determine which device has completed and then to run an ISR that will finish the house-keeping related to completing the I/O operation. Each time the user moves the mouse or types a key or information arrives on a connected network, an interrupt is raised, the trap handler runs, and an ISR is called to manage the incoming information.

12.2.2.4 Thread Scheduling

The Windows NT thread scheduler is a time-sliced, priority-based, pre-emptive scheduler. The basic unit of processor allocation is a time quantum computed as a multiple of the number of clock interrupts. On most Windows NT machines, the time quantum ranges from about 20 to 200 milliseconds. Servers are configured to have time quanta that are six times longer than for a workstation with the same processor type. The scheduler supports 32 different scheduling queues. As in all multiple-level queue schedulers, as long as there are threads in the highest-priority queue, then only those threads will be allocated the processor. If there are no threads in that queue, then the scheduler will service the threads in the second highest-priority queue. If there are no threads ready to run in the second highest-priority queue, the scheduler will service the third highest-priority queue, and so on.

There are three levels of queues:

- Real-time level, consisting of the 16 highest-priority queues
- Variable-level, consisting of the next 15 higher-priority queues
- System-level, consisting of the lowest-priority queue

The scheduler attempts to limit the number of threads that are entered into the real-time queues, thereby increasing the probability that there will be little competition among threads that execute at these high-priority levels. However, Windows NT is not a real-time system and cannot guarantee that threads running at high priority will receive the processor before any fixed deadline. The highest-level queue processing continues through the variable-level queues, down to the system-level queue. The system-level queue contains a single "zero page thread" to represent an idle system. That is, when there are no runnable threads in the entire system, it executes the zero page thread until an interrupt occurs and another thread becomes runnable. The zero page thread is the single lowest-priority thread in the system, so it runs whenever there are no other runnable threads.

A thread's base priority is normally inherited from its process. The priority can also be set with various function calls, provided the caller has the authority to set the priority. The Win32 API model defines four priority classes:

- REAL TIME
- HIGH
- NORMAL
- IDLE

Each thread also has a relative thread priority within the class, any of the following:

- TIME CRITICAL
- HIGHEST
- ABOVE NORMAL
- NORMAL
- BELOW NORMAL
- LOWEST
- IDLE

Thus a thread could be in the HIGH class and operating at the ABOVE NORMAL relative priority at one moment, but then be in the HIGH class and operating at the BELOW NORMAL relative priority a little later. The thread's class and the class's NORMAL relative priority define base priority. If the priority class is not REAL TIME, then the thread's priority will be for one of the variable-level queues. In this case, Windows NT might adjust priorities of threads in the variable level according to system conditions. Windows NT does not change the priority of a thread that has been placed in the real-time levels.

The thread scheduler is also preemptive. This means that whenever a thread becomes ready to run, it is placed in a run queue at a level corresponding to its current priority. If there is another thread in execution at that time and that thread has a lower priority, then the lower-priority thread is interrupted and the new, higher-priority thread is assigned the processor. In a single-processor system, this would mean that a thread could cause itself to be removed from the processor by enabling a higher-priority thread. In a multiprocessor system, the situation can be subtler. Suppose that in a two-processor system, one processor is running a thread at level 10 and the other is running a thread at level 4. If the level 10 thread performs some action that causes a previously blocked thread to suddenly become runnable at level 6, then the level 4 thread will be halted and the new level 6 thread will begin to use the processor that the level 4 thread was using.

12.2.3 Multiprocess Synchronization

Single-processor systems can support synchronization by disabling interrupts. However, Windows NT is designed to also support multiprocessors, so the Kernel must provide an alternative mechanism to ensure that a thread executing on one processor does not violate a critical section of a thread on another processor. The Kernel employs spinlocks by which a thread on one process can wait for a critical section by actively testing a Kernel lock variable to determine when it can enter the critical section, if the hardware supports the test-and-set instruction. Spinlocks are implemented using the hardware. Spinlock synchronization is used only within the Kernel and Executive. User-mode programs use abstractions that are implemented by the Executive.

12.2.3.1 The NT Executive

The NT Executive builds on the Kernel to implement the full set of Windows NT policies and services, including process management, memory management, file management, and device management. Windows NT uses object-oriented technology but the NT Executive is designed and implemented at the source code level as a modularised set of elements.

- Object Manager
- Process and Thread Manager
- Virtual Memory Manager
- Security Reference Manager
- I/O Manager
- Cache Manager
- LPC facility
- Runtime functions
- Executive support functions

12.2.3.1.1 Object Manager

The Executive Object Manager implements another object model on top of the Kernel Object Manager, Whereas Kernel objects operate in a trusted environment, Executive objects are used by other parts of the Executive and user-mode software and must take extra measures to assure secure and reliable operation.

An Executive object exists in supervisor space, though user threads can reference it. This is accomplished by having the Object Manager provide a handle for each Executive object. Whenever a thread needs a new Executive object, it calls an Object Manager function to create the object (in supervisor space), to create a handle to the object (in the process's address space), and then to return the handle to the calling thread.

Sometimes a second thread will want to use an Executive object that has already been created. When the second thread attempts to create the existing object, the Object Manager notes that the object already exists, so it creates a second handle for the second thread to use to reference the existing Executive object. The two threads share the single object. The Object Manager keeps a reference count of all handles to an Executive object. When all outstanding handles have been closed, the Executive object is deal located. Thus it is important for each thread to close each handle it opens, preferably as soon as it no longer needs the handle.

There is a predefined set of about is object types for the Object Manager. When an object is created, it includes an object header (used by the Object Manager to manage the object) and a body to contain type-specific information. The header includes the following:

- Object name: Allows the object to be referenced by different processes.
- Security descriptor: Contains access permissions.
- Open handle information: Contains details of which processes are using the object
- Object type: Contains details of the object's class definition.
- Reference count: Holds the count of the number of outstanding handles that reference the object.

The Object Manager manages the information in the header. For example, when a new handle is created to an object, the Object Manager updates the open handle information and reference count. The object type information defines a standard set of methods that the object implements such as open, close, and delete. Some of these methods are supplied by the Object Manager, and some must be tailored to the object type; however, the interface is determined as part of the object header.

The object body format is determined by the Executive component that uses the object. For example, if the Executive object is a file object, the body format and contents are managed by the File Manager part of the I/O Manager in the Executive.

12.2.3.1.2 Process and Thread Manager

Manager provides an Executive Process and Thread Manager serves the same purpose in Windows NT that a process manager serves in any OS. It is the part of the OS responsible for the following:

- Creating and destroying processes and threads
- Overseeing resource allocation
- Providing synchronization primitives
- Controlling process and thread state changes
- Keeping track of most of the information that the OS knows about each thread

The Process Manager implements the process abstraction that will be used at the subsystem and application levels. Implementing the abstraction means that the Process Manager defines a number of data structures for keeping track of state of each process and thread. The base process descriptor is called an executive process control (EPROCESS block). The EPROCESS block contains information such as identifications, resource lists, and address space descriptions. The EPROCESS block also references a Kernel-level process control block (PCB, or the KPROCESS block), which contains the Kernel's view of the process. The NT Kernel manipulates its portion of the EPROCESS block, and the NT Executive is responsible for maintaining the remaining fields.

There is also a close relationship between an Executive process and a thread. Just as there is an EPROCESS block, there is also an executive thread control (ETHREAD) block for each thread in a process. Since the thread exists within a process, the EPROCESS block references the list of ETHREAD blocks. Information about the thread that is managed by the Process Manager is stored in the ETHREAD block. And because the thread is built on a Kernel-level thread object, there is also a Kernel thread control (KTHREAD) block containing the information about the Kernel thread object that is managed by the Kernel-level management. The EPROCESS block references a KPROCESS block, which references a set of KTHREAD blocks. The EPROCESS block also references a set of ETHREAD blocks, each of which references the same KTHREAD blocks.

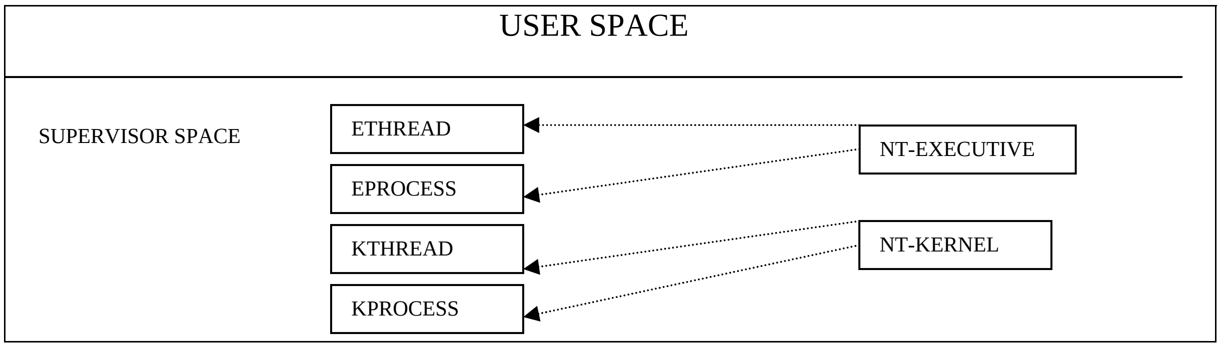


Figure 12.3 Process and Thread Descriptors

The NTOSKRNL function NtCreateProcess is called to create a process; that is, the Win32 API CreateProcess function calls NtCreateProcess. When NtCreateProcess is called (ordinarily by CreateProcess), it performs the following work in setting up the process.

- Calls the Kernel to have it create a Kernel process object.
- Creates and initializes an EPROCESS block.
- Creates an address space for the process.

A process has no ability to execute code in its address space; it must have at least one thread, called the base thread, to execute the code.

The NtCreateThread Executive function creates a thread that can execute within the process. (The Win32 API CreateProcess function calls both NtCreateProcess and NtCreateThread; the CreateThread function calls NtCreateThread to create additional threads within a process.) NtCreateThread performs the following work.

- Calls the Kernel to have it create a Kernel thread object.
- Creates and initializes an ETHREAD block.
- Initializes the thread for execution (sets up its stack, provides it with an executable start address, and so on).
- Places the thread in a scheduling queue.

12.2.3.1.3 Virtual Memory Manager

Windows NT is a paging virtual memory system, which saves a process's address space contents in secondary storage, loading portions of the image from the secondary storage into the primary storage on a page-by-page basis whenever it is needed.

When a process is created, it has 4 GB of virtual addresses available to it, though none of the addresses are actually allocated at that time. When the process needs space, it first reserves as much of the address space as it needs at that moment; reserved addresses do not cause any actual space to be allocated; rather, virtual addresses are reserved for later use. When the process needs to use the virtual addresses to store information, it commits the address space, meaning that some system storage space is then allocated to the process to hold information. A commit operation causes space on the disk (in the process's page file) to be allocated to the process; the information is stored on the disk until it is actually referenced by a thread

When an executing thread references a virtual address, the Virtual Memory Manager ensures that the page containing that virtual address is read from the page file and placed at some system-defined location in the physical executable memory. The Virtual Memory Manager maps the virtual address referenced by the thread into the physical executable memory.

The Virtual Memory Manager has been designed so that a large portion of each process's address space (usually half of it, though different configurations of Windows NT use different fractions) is mapped to the information used by the system when it is in supervisor mode (Figure 12.4). There are a few important implications of this decision, as follows.

A process can directly reference every location in the system.

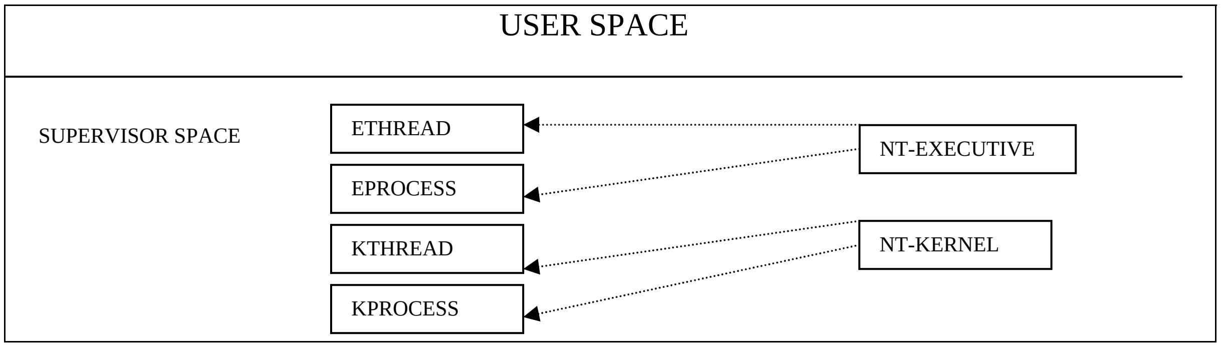


Figure 12.4. Virtual Memory

Every process shares the same view of the system's space. Such a large, shared virtual address space makes memory-mapped files feasible. In Figure 12.4, when a thread references an address in the user space, the virtual memory system loads the target location into the physical memory prior to its use so that the thread can read or write the virtual memory address by referencing a physical memory address. The same mapping takes place for OS memory references, though these references are protected, and every process's OS addresses map to the OS memory rather than to the application-specific part of the address space.

12.2.3.1.4 Security Reference Manager

The NT Kernel supports secure operation by including low-level mechanisms for authentication. The Security Reference Manager is the Executive-level mechanism to implement the critical parts of certifiable security policies. It is constructed to check object access according to any give protection policy (specified within subsystem components that manage the specific access that a process is trying to perform).

The Security Reference Monitor is a protection mechanism used by the Security Reference Manager in conjunction with a security policy module executing in user space. Windows NT includes a user space subsystem component, the Local Security Authority (LSA) server, to represent the desired security policy. The LSA uses its own policy database, stored in the machine’s Registry, to hold the details of the particular machine's policy. The authentication mechanism the LSA server uses to compare access requests with the database contents can also be provided on an installation-by-installation basis, though a default mechanism is provided with NT.

The Security Reference Manager authenticates access to Executive objects. Whenever any thread makes a system call to access an Executive object, the part of the Executive that handles the access passes a description of the attempted access to the Security Reference Monitor. The object contains a security descriptor identifying the object's owner and an access control list (ACL) of processes that are permitted access to the object. The Security Reference Monitor determines the thread's identity and access type and then verifies that the thread is allowed to access the object (according to the information in the (ACL).

12.2.3.1.5 I/O Manager

I/O Manager is responsible for handling all the input/output operations to every device in the system. The I/O Manager creates an abstraction of all device I/O operations on the system so that the system's clients can perform operations on a common place data structure.

The client can perform synchronous and asynchronous I/O.

The client can invoke the Security Reference Monitor whenever security is an issue.

The I/O Manager must accommodate device drivers written in high-level language by third parties. Those drivers must be able to execute in supervisor mode. Installation and removal of a device driver must be dynamic.

The I/O Manager can accommodate alternative files systems on the system's disks. This means that some files systems might use the MS-DOS format others might use an industry standard CD-ROM format, and yet others might use NT's own file system (NTFS).

I/O Manager extensions-device drivers and/or file systems-must be consistent with the memory-mapped file mechanism implemented in the Virtual Memory Manager, so extension designs are constrained by the facilities provided by the manager.

The I/O Manager is made up of the following components, as shown in Figure 12.5. Device drivers are at the lowest level. They manipulate the physical I/O devices. These drivers are described generically in most textbooks.

Intermediate drivers are software modules that work with a low-level device driver to provide enhanced service. For example, a low-level device driver might simply pass an error condition "upward" when it detects it, while an intermediate driver might receive the error and decide to issue a retry operation to the lower-level driver.

File system drivers extend the functionality of the lower-level drivers to implement the target file system.

A filter driver can be inserted between a device driver and an intermediate, driver, between an intermediate driver and a file system driver, or between the file system driver and the I/O Manager API to perform any kind of function that might be desired. For example, a network redirector filter can intercept file commands intended for remote files and redirect them to remote file servers.

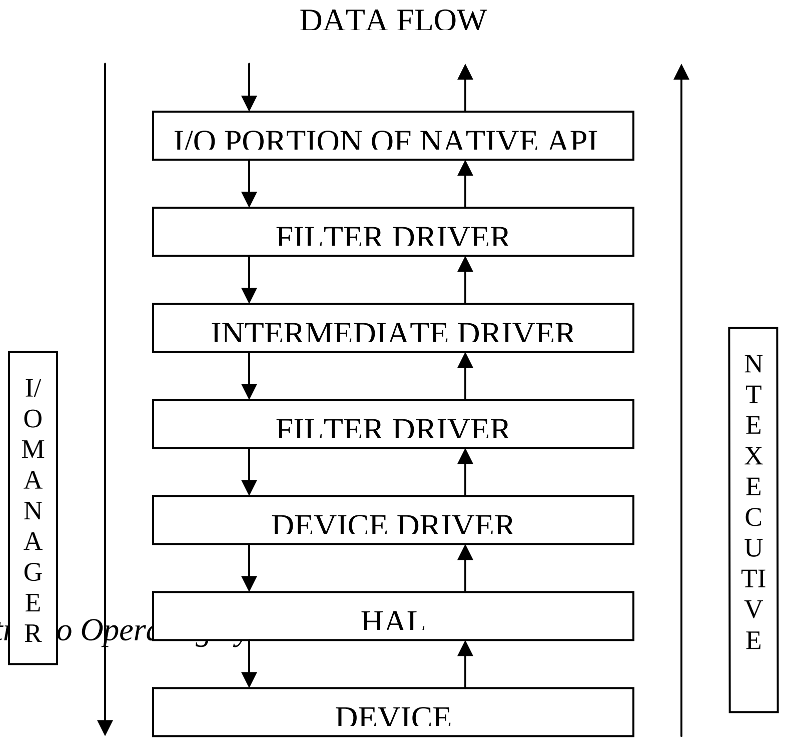


Figure 12.5 The I/O Manager

Drivers are the single component that can be added to the NT Executive to run a low-level device in supervisor mode. The OS has not been designed to support third party software, other than drivers, that want to add supervisor mode functionality. In today's commercial computer marketplace, a consumer can buy a computer from one vendor and then buy disk drives, graphic adapters, sound boards, and so on, from other vendors. The OS must be able to accommodate this spectrum of equipment built by different vendors. Therefore it is mandatory that the OS allow third parties to add software drivers for each of these hardware components that can be added to the computer.

The NT I/O Manager defines the framework in which device drivers, intermediate drivers, file system drivers, and filter drivers are dynamically added to and removed from the system and are made to work together. The dynamic Stream design allows one to easily configure complex I/O systems.

The I/O Manager directs modules by issuing I/O request packets (IRPs) into a stream. If the IRP is intended for a particular module, that module responds to the IRP; otherwise, it passes the IRP to the next module in the stream. Each driver in the stream has the responsibility of accepting IRPs, either reacting to the IRP if it is directed at the driver or passing it on to the next module if it is not.

All information read from or written to the device is managed as a stream of bytes, called a virtual file. Every driver is written to read and/or write a virtual file. Low-level device drivers transform information read from the device into a stream & transform stream information into a device-dependent format before writing it.

As a result of the design of the I/O system architecture, the API to the I/O subsystem is not complex. For example, subsystems can use NtCreateFile or NtOpen to create a handle to an Executive file object, NtReadFile and NtWriteFile to read and write an open file, and Ntlock and NtUnlock to lock a portion of a file.

12.2.3.1.6 Cache Manager

A bottleneck to an application's performance is the time the application must wait for a physical device to process an I/O command. As processors become faster, that fraction of the total runtime spent waiting for devices to complete their I/O operations increasingly dominates the total runtime. The solution to the problem is to devise ways for the thread to execute concurrently with its own device I/O operations. This means that the thread is able to predict information that it will need before it actually needs it and issue an I/O request in anticipation of using data, while concurrently processing data it has already read.

The Cache Manager is designed to work with the Virtual Memory Manager and the I/O Manager to perform read-ahead and write-behind on virtual files. The idea is a classic OS idea. That is, since files are usually accessed sequentially, whenever a thread reads byte i, it is likely to read byte i+1 soon thereafter. Therefore, on a read-ahead strategy, when the thread requests that byte i be read from the device, the Cache Manager asks the Virtual Memory Manager to prepare a buffer to hold K+1 bytes of information from the virtual file and instructs the I/O Manager to read byte i and the next K bytes into the buffer. Then when the thread requests byte i+1, i+2, ..., i+K, those bytes will have already been read, so the thread need not wait for a device operation to complete. The write-behind strategy works similarly.

Most of the Cache Manager's operation is transparent above the NTOSKRNL API. The Win32 API has only four attributes that it can set when CreateFile is called to influence the Cache Manager's operation. These attributes are essentially information to assure the Cache Manager that the thread will access the information in the file sequentially. The main clients for the Cache Manager are drivers that are added to the I/O Manager. It is these modules that customize the file system and use the file-caching facilities provided by this manager.

12.2.3.1.7 The Native API

Executive and Kernel are combined into the NTOSKRNL.EXE executable when NT is built. The combined Executive and Kernel module (with the underlying HAL) implements the full NT OS. In Version 5.0, NTOSKRNL exports about 240 functions, most of which are undocumented, meaning those only subsystem developers should base their software on the functions. Developers call this interface the NT Native API or the Executive API. In this lesson, it is called the Native API. Microsoft provides Windows NT with a set of complementary subsystems, some of which provide more abstract APIs that application programmers are expected to use.

12.2.4 NT Subsystems

Software systems are often constructed as a layered architecture. Layer i is constructed using the services provided by layer i -1, creating its own services and exporting them through its own (layer i) interface. There are several reasons for the popularity of layered architectures.

- It is a simple strategy for dividing and conquering a large problem.
- Each layer implements a well-defined subset of the total system functionality.
- The functionality at layer i can be designed and tested as a manageable unit.
- Layer i+1 services can simplify the way work is done using layer i or lower.

Layer i+1 services can be ported across different implementations of layer i. In the Windows NT architecture, subsystems provide a layer of service above the Native API. There can be many different subsystems, some related, but others independent of one another, as functionality is added to the computer system. For example, a typical Windows NT system includes the Win32 subsystem, the WinLogon service, a remote procedure call service, and perhaps a Win 16 subsystem.

If the Windows NT machine were required to support POSIX application programs, a POSIX subsystem could be added as a component in the subsystem layer. Each subsystem uses the Native API to provide the services it implements. The environment subsystems behave as a traditional interior layer. In the layered architecture approach, they use the Native API, add functionality and services, and then export their own API. In the Microsoft strategy, subsystem APIs are documented APIs, meaning that a programmer can write new software at the next higher-level layer and be assured that the API will be unchanged when implementations at a lower-level layer in the architecture are changed.

Figure 12.6 shows how this layering works in NT. The Win32 Subsystem exports a documented interface, the Win32 API, as a set of about 1,000 functions {f0, f1, .., fn}. The Win32 API is a documented interface. An application programmer can write software above the Win32 subsystem that calls the functions, {f0, f1, ..., fn} to accomplish an application-specific task.

The Win32 Subsystem also provides a user interface management system, since the Executive/Kernel does not have one of these. This is primarily a matter of practicality-when the system begins to run, some part of the system software needs to read the keyboard and mouse and manage the display. Rather than have each environment subsystem provide its own user interface, the Win32 Subsystem implements the common window manager for all subsystems. This means that there is a single human computer interaction model implemented in a single subsystem, but used by all other subsystems.

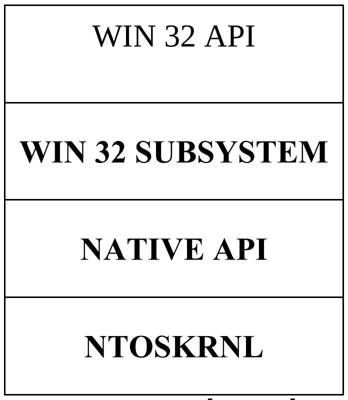


Figure 12.6 The Win32 API

A subsystem's design can be simple or complex. In the simplest case, each function or service that the subsystem exports is implemented wholly within the subsystem itself. For example, the subsystem might keep a data structure filled with information it extracts from information obtained through the Native API. When a program queries the subsystem, it simply reads the data structure and returns a result without ever interacting with the OS.

A slightly more complex case occurs when a subsystem function requires that the subsystem implementation interact with the OS via the Native API. For example, the WIn32 API function CreateProcess causes the WIn32 subsystem to call the Native API functions NtCreateProcess and NtCreateThread.

The most complex design requires that the interaction between the subsystem and the OS be more complicated than a function call or two. The Executive provides a special interprocess communication facility called the Local Procedure Call (LPC) facility. The LPC facility allows one process to call a function that is implemented in another process. This requires special OS activity, since the target procedure is not known to the compiler and link editor and is not determined until the processes are running. When the calling process starts an LPC, the OS takes the call request, finds the procedure in the target process's address space, and calls the target procedure. Two processes can communicate with one another by making LPCs back and forth. The most complex subsystem designs use LPCs to invoke Executive functions.

12.2.4.1 Win 32 API: The Programmer's View of NT

The Win32 API is the "official OS interface" to all Microsoft OSs. The rationale for having a single OS API relates to portability. That is, if all Microsoft OSs can export the same API, then an application writer can produce application software that will work on all OS versions. Further, enhancements to any of the OS products still provide the same services via the same, fixed interface. The cost of adopting this strategy is the need for a subsystem between the OS's native API and the API used by the application programmers.

MS-DOS created a set of fundamental OS services on which application programmers came to depend. Unfortunately, the original MS-DOS API is very old. As a result, it had many built-in dependencies on 16-bit address spaces, single thread of execution, and so on. The MS-DOS API was upgraded to a Windows interface, now generally regarded as the Win 16 API. Yet that still was not adequate to allow programmers to use the full power of Windows NT, Windows 9x, and CE. All of Microsoft's current OS family implement some variant of the Win32 API. Whenever an application programmer writes code for a Microsoft OS, the only documented interfaces available are the Win32 API versions for each OS. There are few differences between the Windows 9x and Windows NT implementations of Win32 API. Since CE is aimed at such hardware as palmtop computers and television set-top boxes, its variant of the Win32 API is distinctly different from the mainstream API. The Win32 API has about 1,000 function calls. The Win32 API uses most of the same abstractions that appear at the Native API, including processes, threads, objects, handles, and files. One reason the Win32 API is so much larger than the Native API is that the Win32 API also includes the interface to all of the graphics and user interfaces components, code that is not part of NT.

12.3 Keywords

Microkernel: It is a small nucleus of code comprising of the most essential OS functions.

NT Kernel: It provides specific mechanisms for general object and memory management, process management, file management, and device management.

HAL: Hardware Abstraction Layer (HAL) is responsible for mapping various low-level, processor-specific operations into a fixed interface that is used by the Windows NT Kernel and Executive.

Virtual Memory Manager: It maps the virtual address referenced by the thread into the physical executable memory.

I/O Manager: It is responsible for handling all the input/output operations to every device in the system.

Cache Manager: It is designed to work with the Virtual Memory Manager and the I/O Manager to perform read-ahead and write-behind on virtual files.

12.4 SUMMARY

Microsoft designed NT to be an extensible, portable operating system, able to take advantage of new techniques and hardware. NT supports multiple operating environments and symmetric multiprocessing. The use of kernel objects to provide basic services, and the support for client-server computing, enable NT to support a wide variety of application environments. For instance, NT can run programs compiled for MS-DOS, Win16, Windows 95, NT, and POSIX. It provides virtual memory, integrated caching, and preemptive scheduling. NT supports a security model stronger than those of previous Microsoft operating systems, and includes internationalization features. NT 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.

12.5 SUGGESTED READINGS / REFERENCE MATERIAL

7. The Design of the UNIX Operating System, Bach M.J., PHI, New Delhi, 2000.
8. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
9. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
10. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
11. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
12. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

12.6 SELF-ASSESSMENT QUESTIONS (SAQ)

1. What is meant by a multi-user and multi-tasking operating system?
2. How is Unix different from other Operating systems like DOS, Windows, etc.?
3. What are the various services provided by an Operating System?
4. What is function of a Kernel?
5. What is the general structure of Windows-NT operating system?
6. Windows-NT is a paging virtual memory system. Comment.
7. Discuss Win 32 API.