DIPLOMA IN COMPUTER APPLICATION

DCA-14-T

OPERATING SYSTEM

Centre for Distance and Online Education
Guru Jambheshwar University of Science & Technology, HISAR-125001

CONTENTS

3.3 Process Concept

3.3.1 The Process

3.3.2 Process States

3.3.3 Process control block

3.3.4 Threads

3.4 Process Scheduling

3.4.1 Scheduling Queue

3.4.2 Types of Schedulers

3.4.2.1 Long term Scheduler

3.4.2.2 Medium term Scheduler

3.4.2.3 Short term Scheduler

3.5 Operations on Process

3.5.1 Process Creation

3.5.2 Process Termination

3.6 Scheduling & Performance Criteria

3.6.1User-Oriented Scheduling Criteria

3.6.2 System-Oriented Scheduling Criteria

3.7 Scheduling Algorithms

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

3.7.2 Shortest Job First (SJF)

3.7.3 Shortest Remaining Time Next (SRTN) Scheduling

3.7.4 Round Robin Scheduling

3.7.5 Priority-Based Preemptive Scheduling

Lesson Number: 1

Introduction to Operating System

1.1 Learning Objectives

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. 2 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.

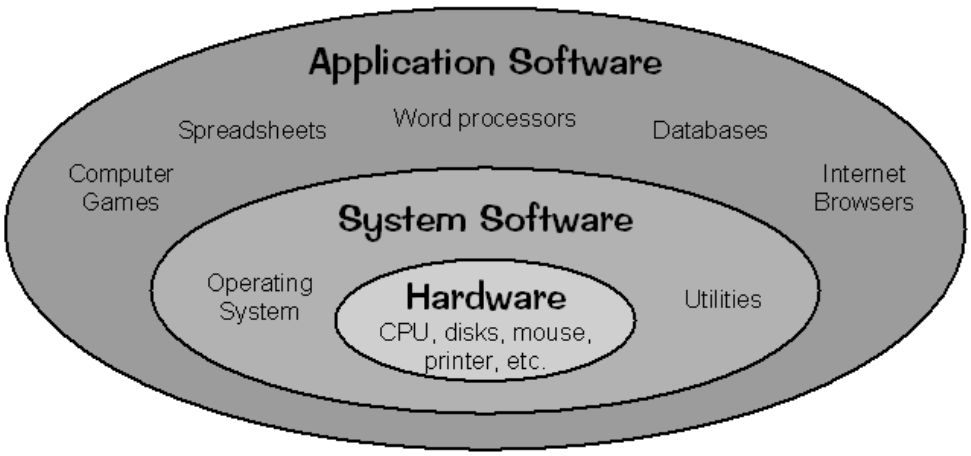


Figure 1: The relationship between application & system software

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.

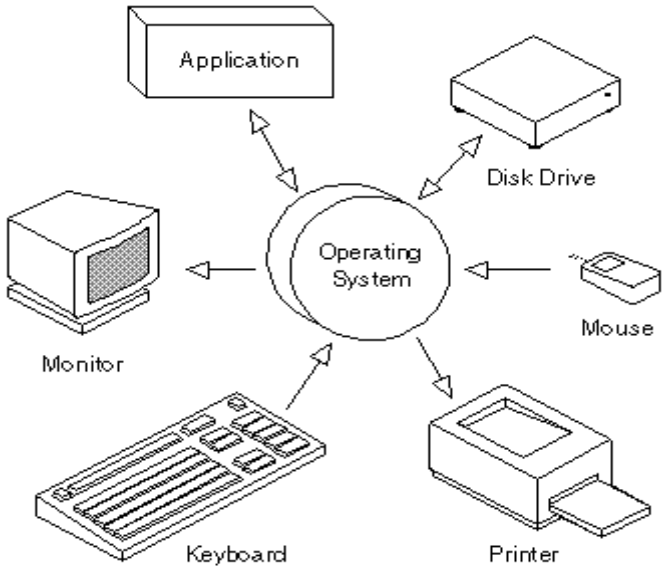


Figure 2: The interface of various devices to an operating system

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.

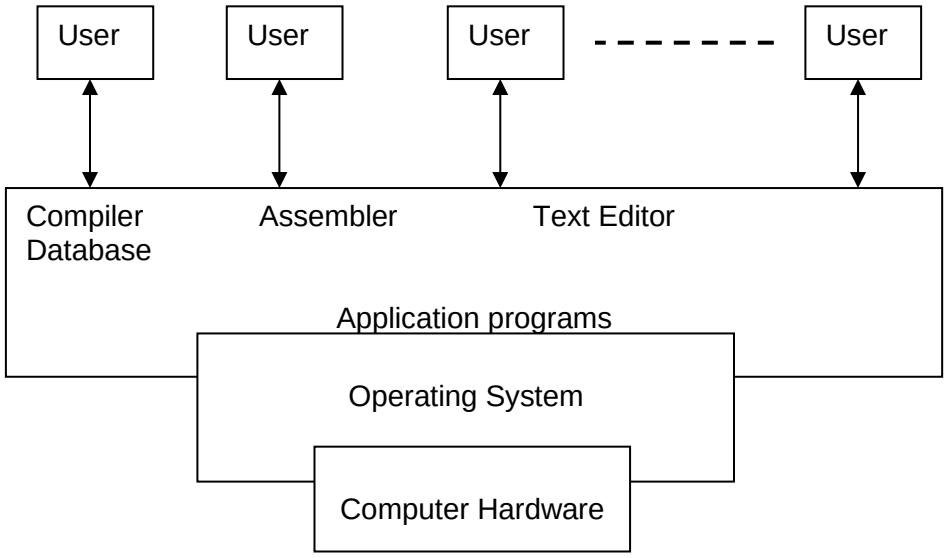


Figure 3. Basic components of a computer system

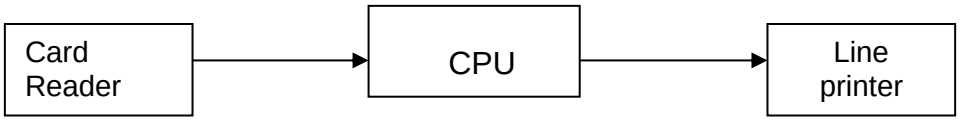
We can view an OS as a resource allocator. A computer system has many resources, which are to be required to solve a computing problem. These resources are the CPU time, memory space, files storage space, input/output devices & so on. The OS acts as a manager of all of these resources & allocates them 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.

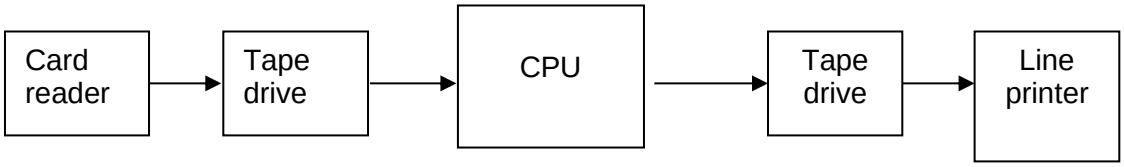
An OS can also be viewed as a control program, used to control the various I/O devices & the users programs. A control program controls the execution of the user programs to prevent errors & improper use of the computer resources. It is especially concerned with the operation & control of I/O devices. 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

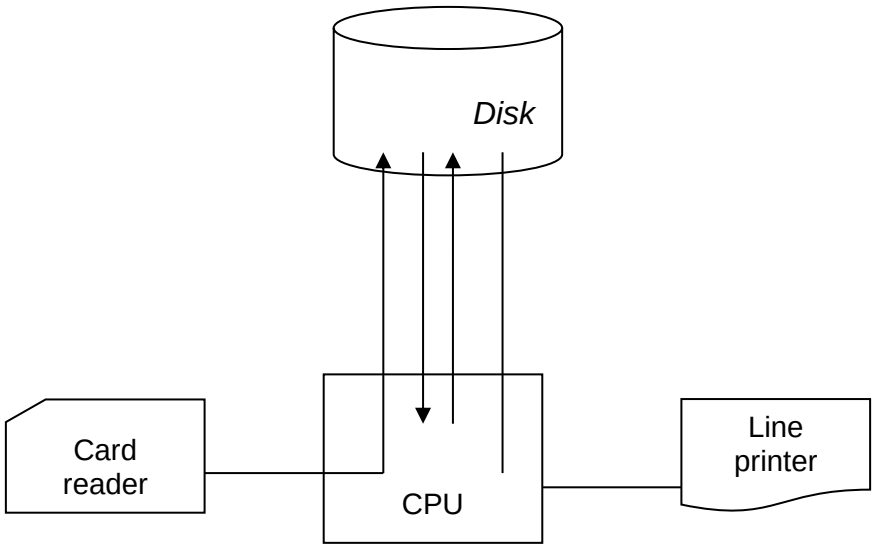
were retrieved. Third, to reduce the gap in speed between the I/O devices & the CPU, an interface called the control unit was placed between them to perform the function of buffering. A buffer is an interim storage area that works like this: as the slow input device reads a record, the control unit places 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 Operation



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.

1.3 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**.**

1.3.1 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.

1.3.2 Multi Programming 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.

1.3.3 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.

1.3.4 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.

1.3.5 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.

1.3.6 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.

1.3.7 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**.**

1.3.8 Network Operating System

It is one of the important type of operating system. Network Operating System runs on a server and gives the server the capability to manage data, users, groups, security, applications, and other networking functions. The basic purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Some examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

1.3.9 Real Time System

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.

1.4 Check Your Progress

1. What is operating system?

A. collection of programs that manages hardware resources

B. system service provider to the application programs

C. link to interface the hardware and application programs

D. all of the mentioned

2. To access the services of operating system, the interface is provided by the:

A. system calls

B. API

C. library

D. assembly instructions

3. Which one of the following is not true?

A. kernel is the program that constitutes the central core of the operating system

B. kernel is the first part of operating system to load into memory during booting

C. kernel is made of various modules which can not be loaded in running operating system

D. kernel remains in the memory during the entire computer session

4. Which one of the following error will be handle by the operating system?

A. power failure

B. lack of paper in printer

C. connection failure in the network

D. all of the mentioned

5. The main function of the command interpreter is:

A. to get and execute the next user-specified command

B. to provide the interface between the API and application program

C. to handle the files in operating system

D. none of the mentioned

6. By operating system, the resource management can be done via:

A. time division multiplexing

B. space division multiplexing

C. both (a) and (b)

D. none of the mentioned

1.5 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.

1.6 Keywords

Operating System is the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.

1.7 Self Assessment Questions (SAQ)

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

1.8 Answers To Check Your Progress

1. all of the mentioned
2. system calls
3. kernel is made of various modules which can not be loaded in running operating system
4. all of the mentioned
5. to get and execute the next user-specified command
6. both (a) and (b)

1.9 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

Operating System Structures

2.1 Learning Objectives

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

1. What is an operating system structure?
2. Roles of system programs and operating services.
3. What are the system call and its types.

2.2 Introduction

The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating-system designers.

2.3 Operating System Services

An Operating System supplies different kinds of services to both the users and to the programs as well. It also provides application programs (that run within an Operating system) an environment to execute it freely. It provides users the services run various programs in a convenient manner. These operating-system services

are provided for the convenience of the programmer, to make the programming task easier. Figure shows one view of the various operating-system services

and how they interrelate.

User Interface

Usually Operating system comes in three forms or types. Depending on the interface their types have been further subdivided. These are:

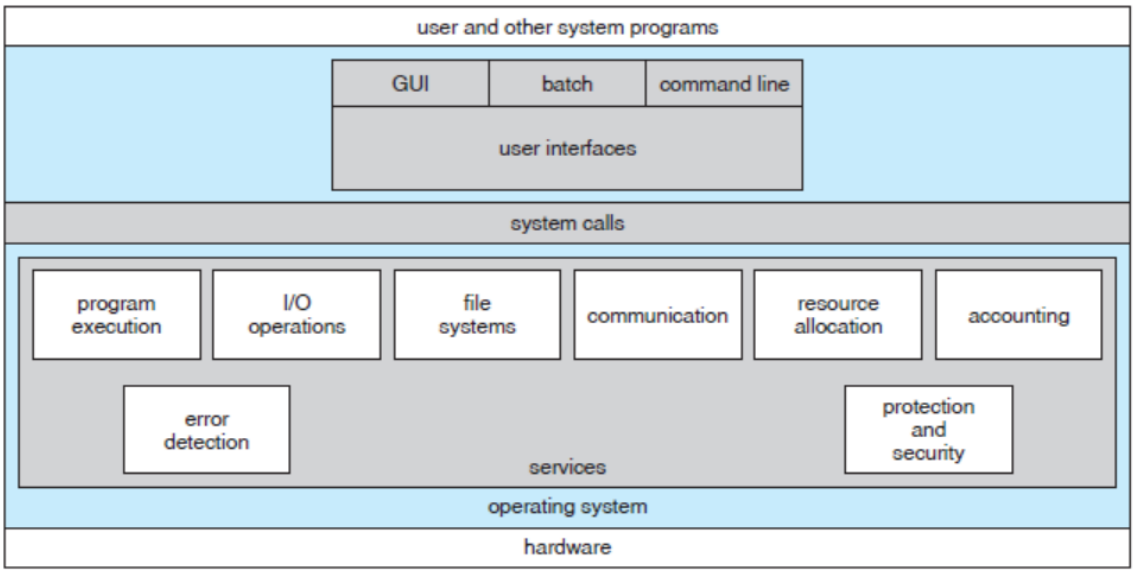


Figure 1: A view of operating system services

- Command line interface
- Batch based interface
- Graphical User Interface

Let's get to know in brief about each of them.

The command line interface (CLI) usually deals with using text commands and a technique for entering those commands. The batch interface (BI): commands and directives are used to manage those commands that are entered into files and those files get executed. Another type is the graphical user interface (GUI): which is a window system with a pointing device (like mouse or trackball) to point to the I/O, choose from menus driven interface and to make choices viewing from a number of lists and a keyboard to entry the texts.

Program execution

Operating system handles many kinds of activities from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated as a process. A process

includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use).Following are the major activities of an operating system with respect to program management.

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

I/O Operation

I/O subsystem comprised of I/O devices and their corresponding driver software. Drivers hides the peculiarities of specific hardware devices from the user as the device driver knows the peculiarities of the specific device. Operating System manages the communication between user and device drivers. Following are the major activities of an

- operating system with respect to I/O Operation.
- I/O operation means read or write operation with any file or any specific I/O device.
- Program may require any I/O device while running.
- Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computer can store files on the disk (secondary storage), for long term storage purpose. Few examples of storage media are magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management.

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.

- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, operating system manages communications between processes. Multiple processes with one another through communication lines in the network. OS handles routing and connection strategies, and the problems of contention and security. Following are the major

- activities of an operating system with respect to communication.
- Two processes often require data to be transferred between them.
- The both processes can be on the one computer or on different computer but are connected through computer network.
- Communication may be implemented by two methods either by Shared Memory or by Message Passing.

Error handling

Error can occur anytime and anywhere. Error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling.

- OS constantly remains aware of possible errors.
- OS takes the appropriate action to ensure correct and consistent computing.

Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management.

- OS manages all kind of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering computer systems having multiple users the concurrent execution of multiple processes, then the various processes must be protected from each another's activities.

Protection refers to mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection.

- OS ensures that all access to system resources is controlled.
- OS ensures that external I/O devices are protected from invalid access attempts.
- OS provides authentication feature for each user by means of a password.

2.4 System Program

System programs provide an environment where programs can be developed and executed. In the simplest sense, system programs also provide a bridge between the user interface and system calls. In reality, they are much more complex. For example, a compiler is a complex system program.

The system program serves as a part of the operating system. It traditionally lies between the user interface and the system calls. The user view of the system is actually defined by system programs and not system calls because that is what they interact with and system programs are closer to the user interface. The user view the operating system observed is actually the system programs and not the system calls. System programs can be divided into seven parts. These are given as follows

- **Status Information**: The status information system programs provide required data on the current or past status of the system. This may include the system date, system time, and available memory in system, disk space, logged in users etc.
- **Communications**: These system programs are needed for system communications such as web browsers. Web browsers allow systems to communicate and access information from the network as required

- **File Modification:** System programs that are used for file modification basically change the data in the file or modify it in some other way. Text editors are a big example of file modification system programs.
- **File Manipulation: These** system programs are used to manipulate system files. This can be done using various commands like create, delete, copy, rename, print etc. These commands can create files, delete files, copy the contents of one file into another, rename files, print them etc.
- **Application Programs:** Application programs can perform a wide range of services as per the needs of the users. These include programs for database systems, word processors, plotting tools, spreadsheets, games, scientific applications etc.
- **Programming Language Support:** These system programs provide additional support features for different programming languages. Some examples of these are compilers, debuggers etc. These compile a program and make sure it is error free respectively.

2.5 Operating System Structure

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations

2.5.1 Simple Structure: There are several commercial system that don't have a well- defined structure such operating systems begins as small, simple & limited systems and then grow beyond their original scope. MS-DOS is an example of such system. It was not divided into modules carefully. Another example of limited structuring is the UNIX operating system.

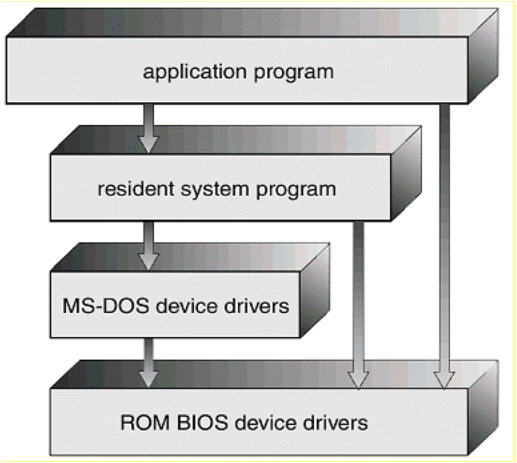


Figure 2: Layered approach

2.5.2 Layered Approach: Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer. This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise. Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

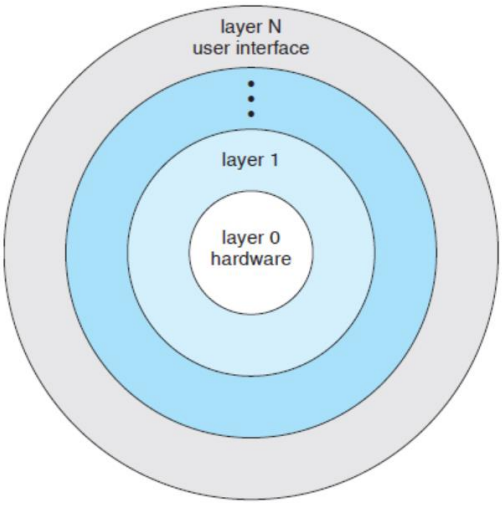


Figure 3: Layered operating system

2.5.3 Micro kernels: The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.

Most microkernels provide basic process and memory management, and message passing between other services, and not much more. Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.

System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel. Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.

Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic. Another microkernel example is QNX, a real-time OS for embedded systems.

> ***2.5.4 Modules:*** *Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems. The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.*

For example, the Solaris operating system structure, is organized around a core kernel with seven. Types of loadable kernel modules:

- Scheduling classes
- File systems
- Loadable system calls
- Executable formats
- STREAMS modules
- Miscellaneous
- Device and bus drivers

2.6 System Calls

A system call is programming interface for an application to request service from the operating system. A system call is a **method** used by application programs to **communicate with the system core**. In modern operating systems, this method is used if a user application or process needs to pass information onto the hardware, other processes or the kernel itself, or if it needs to read information from these sources. This makes these calls a **link between user mode and kernel mode**, the two key access and security modes for processing CPU commands in computer systems. As soon as the action required by a system call is carried out, the kernel gives control back again, and the program code is continued from the point it had reached before the system call was started. The need for system calls is closely tied to the modern operating system model with user mode and kernel mode, which was implemented as a response to the rising number of processes being carried out simultaneously in computers' **main memory**

The more privileged kernel mode is the pivotal control system here because – as mentioned already – not only are all services and processes in the system itself run there, but also **system-critical actions by application programs** that are blocked in user mode. One requirement is the right system call through the respective program, which in most cases is simply for access to processing power or memory structures. If an application needs more **computing power** or **storage space**, for example, or an **application-external file** is required, system calls are essential. After the entire file is copied, the program may close both files, write a message to the console or window, and finally terminate normally. As we simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. This system call sequence is shown in Figure

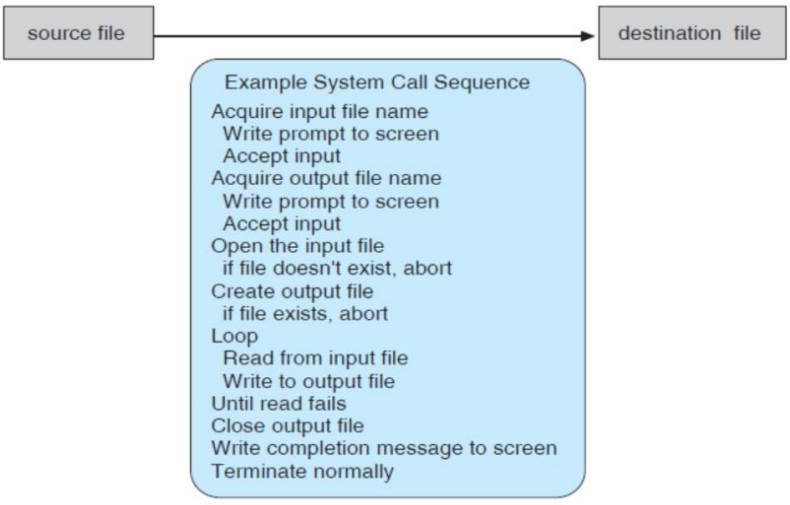


Figure 4: Example of how system calls are used

2.7 Types of System Calls

The system call provides an interface to the operating system services. Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained: Portability: as long a system supports an API, any program using that API can compile and run. Ease of Use: using the API can be significantly easier than using the actual system call.

Three general methods exist for passing parameters to the OS: Parameters can be passed in registers. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register. Parameters can also be pushed on or popped off the stack by the operating system. There are 5 different categories of system calls: process control, file manipulation, device manipulation, information maintenance, and communication.

2.7.1 Process Control

A process or job executing one program may want to load and execute another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program. A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

- **End, abort:** A running program needs to be able to has its execution either normally (end) or abnormally (abort).
- **Load, execute:** A process or job executing one program may want to load and executes

another program.

- **Create Process, terminate process:** There is a system call specifying for the purpose of creating a new process or job (create process or submit job). We may want to terminate a job or process that we created (terminates process, if we find that it is incorrect or no longer needed).

- **Get process attributes, set process attributes:** If we create a new job or process we should able to control its execution. This control requires the ability to determine & reset the attributes of a job or processes (get process attributes, set process attributes).
- **Wait time:** After creating new jobs or processes, we may need to wait for them to finish their execution (wait time).
- **Wait event, signal event:** We may wait for a specific event to occur (wait event). The jobs or processes then signal when that event has occurred (signal event).

2.7.2 File Management

We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it. There is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls. These are as follow

- **Create file, delete file:** We first need to be able to create & delete files. Both the system calls require the name of the file & some of its attributes.
- **Open file, close file:** Once the file is created, we need to open it & use it. We close the file when we are no longer using it.
- **Read, write, reposition file:** After opening, we may also read, write or reposition the file (rewind or skip to the end of the file).
- **Get file attributes, set file attributes:** For either files or directories, we need to be able to determine the values of various attributes & reset them if necessary. Two system calls get file attribute & set file attributes are required for their purpose.

2.7.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file. User programs *request* the device, and when finished they *release* the device. Once the device has been requested and allocated to us, we can read, write, and (possibly) reposition the

device, just as we can with files.

- **Request device, release device:** If there are multiple users of the system, we first request the device. After we finished with the device, we must release it.
- **Read, write, reposition:** Once the device has been requested & allocated to us, we can read, write & reposition the device.

2.7.4 Information Maintenance

The operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes). Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*. The OS also keeps information about all its processes and provides system calls to report this information.

- **Get time or date, set time or date:** Most systems have a system call to return the current date & time or set the current date & time.
- **Get system data, set system data:** Other system calls may return information about the system like number of current users, version number of OS, amount of free memory etc.
- **Get process attributes, set process attributes:** The OS keeps information about all its processes & there are system calls to access this information.

2.7.5 Communication

There are two modes of communication such as:

- **Message passing model:** Information is exchanged through an inter process

  Communication facility provided by operating system. Each computer in a network has a name by which it is known. Similarly, each process has a process name which is translated to an equivalent identifier by which the OS can refer to it. The get host id and get processed systems calls to do this translation. These identifiers are then passed to the general purpose open & close calls provided by the file system or to specific open connection system call. The recipient process must give its permission for communication to take place with an accept connection call. The source of the communication known as client & receiver known as server exchange

messages by read message & write message system calls. The close connection call terminates the connection.

- **Shared memory model:** processes use map memory system calls to access regions of memory owned by other processes. They exchange information by reading & writing data in the shared areas. The processes ensure that they are not writing to the same location simultaneously.

2.7.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multi programmed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to PDAs, must be concerned with protection. Typically, system calls providing protection include

- **Set permission and get permission**, which manipulate the permission settings of resources such as files and disks.
- **Allow user and deny user** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

2.8 Check Your Progress

1. The …….. is regarded as shell, is the layer that actually interacts with the user.
2. ……. is a program interface that enables a user to communicate with computer through graphics or symbols.
3. A…….. is programming interface for an application to request service from the operating system.
4. A running program needs to be able to has its execution either normally called as…….. or abnormally as……....
5. Application developers often do not have direct access to the system calls, but can access them through an …….

2.9 Summary

Operating system combines the functions of an extended machine and a resource manager. The extended machine separates applications from the low-level platform-dependent details by providing

high-level platform-independent abstractions such as windows, sockets, files. The resource manager separates applications from each other by providing mechanisms such as sharing and locking. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. The type of system desired is the foundation for choices among various algorithms and strategies that will be needed.

2.10 Keywords

Command interpreter, that allows users to directly enter commands to be performed by the operating system

Graphical user interface, or GUI allows users to interface with the operating system .

System calls provide an interface to the services made available by an operating

system.

2.11 Self Assessment

1. What is the purpose of system calls?
2. What are the five major activities of an operating system with regard to process management?
3. What are the three major activities of an operating system with regard to memory management?
4. What system calls have to be executed by a command interpreter or shell in order to start a new process?
5. What is the purpose of system programs?
6. What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

7. Describe three general methods for passing parameters to the operating system.
8. What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

2.12 Answer To Check Your Progress

1. user interface
2. GUI (Graphical User Interface)
3. system call
4. End ,Abort
5. application programming interface (API).

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

Lesson Number: 3

Process Management

3.1 Learning Objectives

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. To introduce the notion of a process—a program in execution, which forms the basis of all computation
2. To describe the various features of processes, including scheduling,

   Creation and termination, and communication.

3. To describe communication in client–server systems

3.2 Introduction

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs, and these needs resulted in the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, you will read about what processes are and how they work.

3.3 Process Concept

3.3.1 The Process

A process or task is an instance of a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data and a data section, which contains global variables .A process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure .

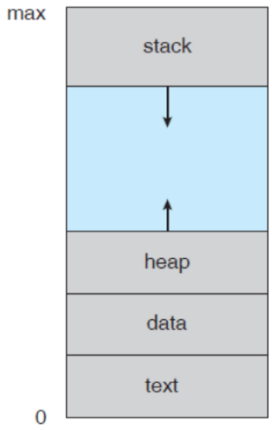


Figure 1: Process in memory

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an *active* entity, with a program counter Specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

3.3.2 Process States

As a process executes, it changes state. The state of a process is defined in

part by the current activity of that process. Each process may be in one of the

following states:

1. **New: –** The state in which a process is created. The New state is the newly created program which is stored in the secondary storage, and taken by the operating system at the time of process creation.
2. Ready:-The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3. Running -Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4. Waiting:- Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5. Terminated :-Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Many processes may be in ready and waiting state at the same time. But only one process can be running on any processor at any instant as shown in figure below

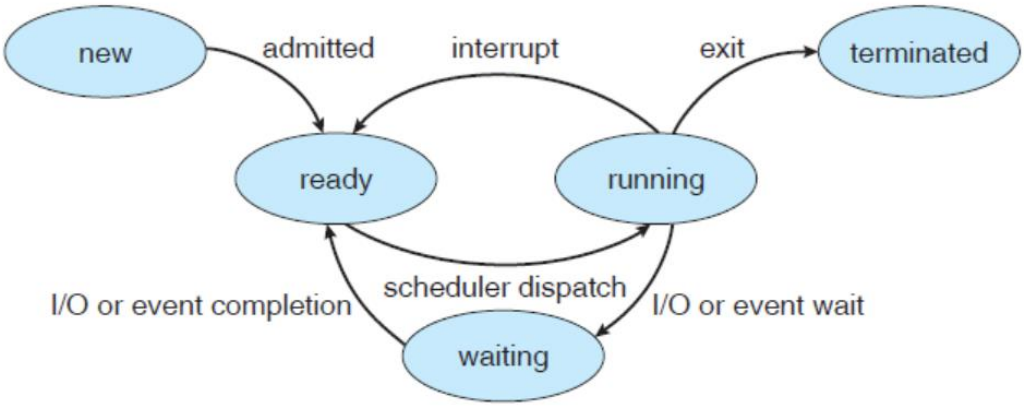


Figure 2 Diagram of process states

3.3.3 Process Control Block

Each process is represented in the operating system by a process control block. The Attributes of the process are used by the Operating System to create the process control block (PCB) for each of them.

This is also called context of the process. Attributes which are stored in the PCB are described below.

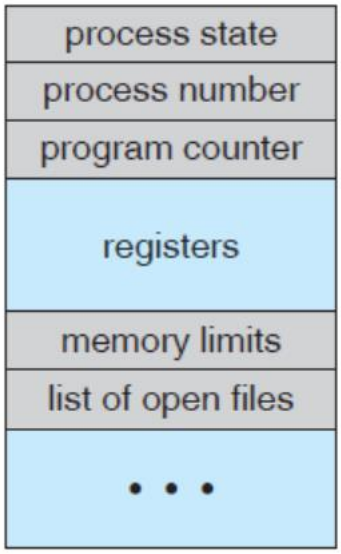


Figure 3 Process Control Block

1. Process state. The state may be new, ready, running, and waiting, halted, and so on.
2. Program counter. The counter indicates the address of the next instruction to be executed for this process.
3. CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
4. CPU-scheduling information. This information includes a process priority,

   Pointers to scheduling queues, and any other scheduling parameters.
5. Memory-management information. This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
6. Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
7. I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

3.3.4 Threads

A thread means a lightweight process. Thread is the basic unit of CPU execution, which consists of thread ID, Program counter, set of registers to hold the information of current working variable, and Stack, which includes the history of execution.A thread shares some information to its associated thread and the information consists of open files, the data segment, and code segments. If a thread changes a code segment memory item, then the other threads can see the changes in the thread.

In other words, the thread is defined as a flow of execution via the process code. Thread improves the performance of applications with the help of parallelism because threads are executed in a parallel manner. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

3.4 Process Scheduling

Scheduling is a fundamental function of OS. The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones. When a computer is multiprogrammed, it has multiple processes completing for the CPU at the same time. If only one CPU is available, then a choice has to be made regarding which process to execute next. This decision making process is known as scheduling and the part of the OS that makes this choice is called scheduler.

3.4.1 Scheduling Queue

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue. The Operating System maintains the following important process scheduling queues −

- Job queue − This queue keeps all the processes in the system.

- Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue.

This queue is generally stored as a linked list . a ready queue header contains pointers to the first & final PCB in the list. The PCB includes a pointer field that points to the next PCB in the ready queue. The lists of processes waiting for a particular I/O device are kept on a list called device queue. Each device has its own device queue. A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution & is given the CPU.

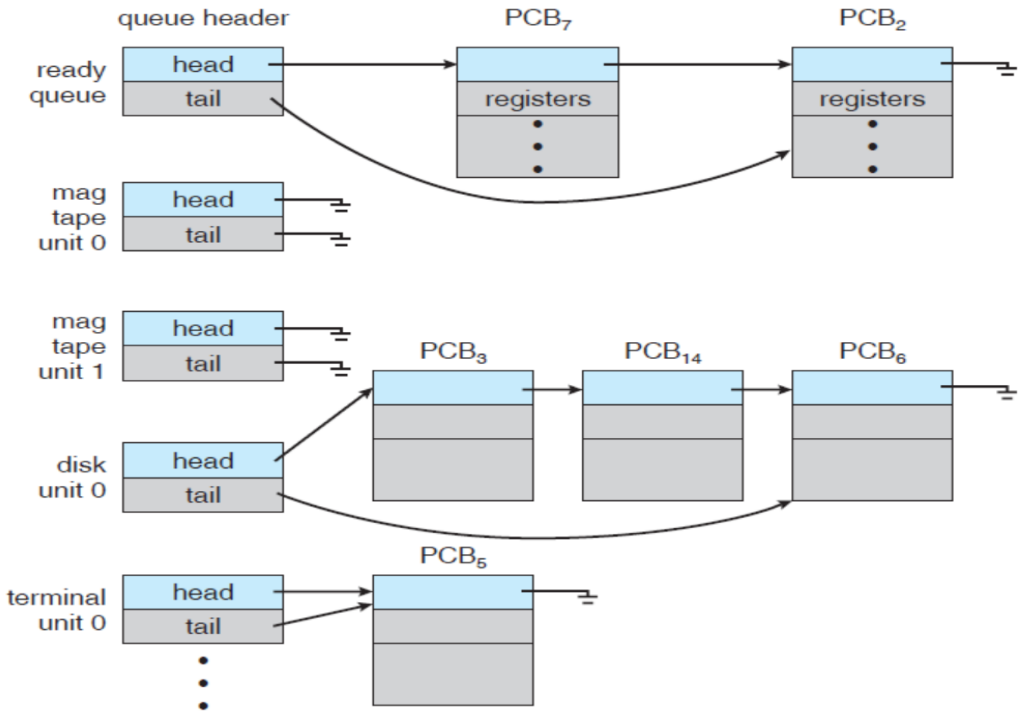


Figure 4 The ready queue and various I/O device queue

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

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

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

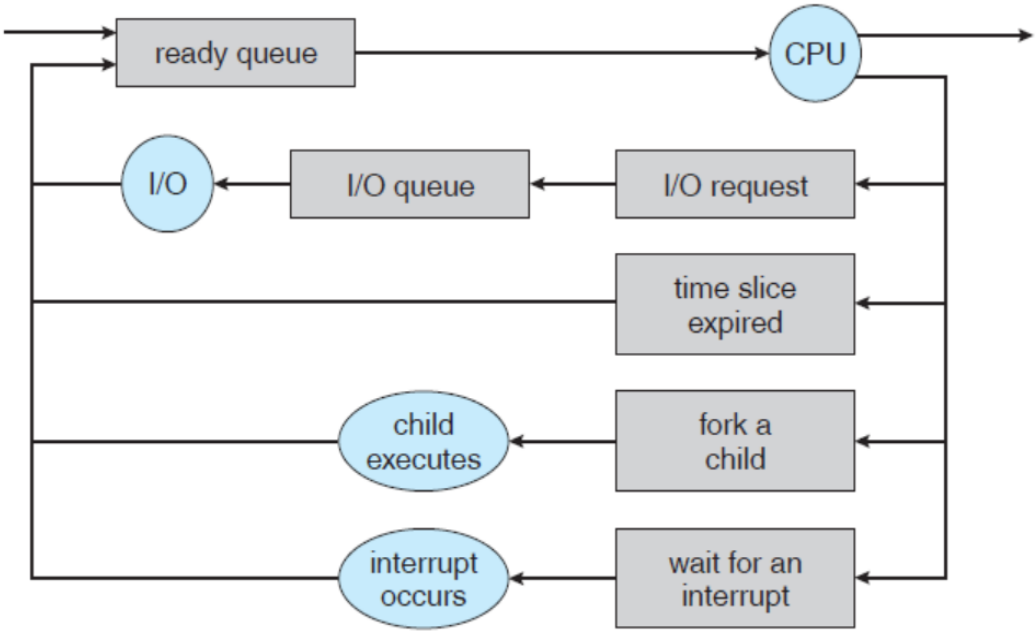


Figure 5 Queuing diagram representation of process scheduling

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

3.4.2 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.

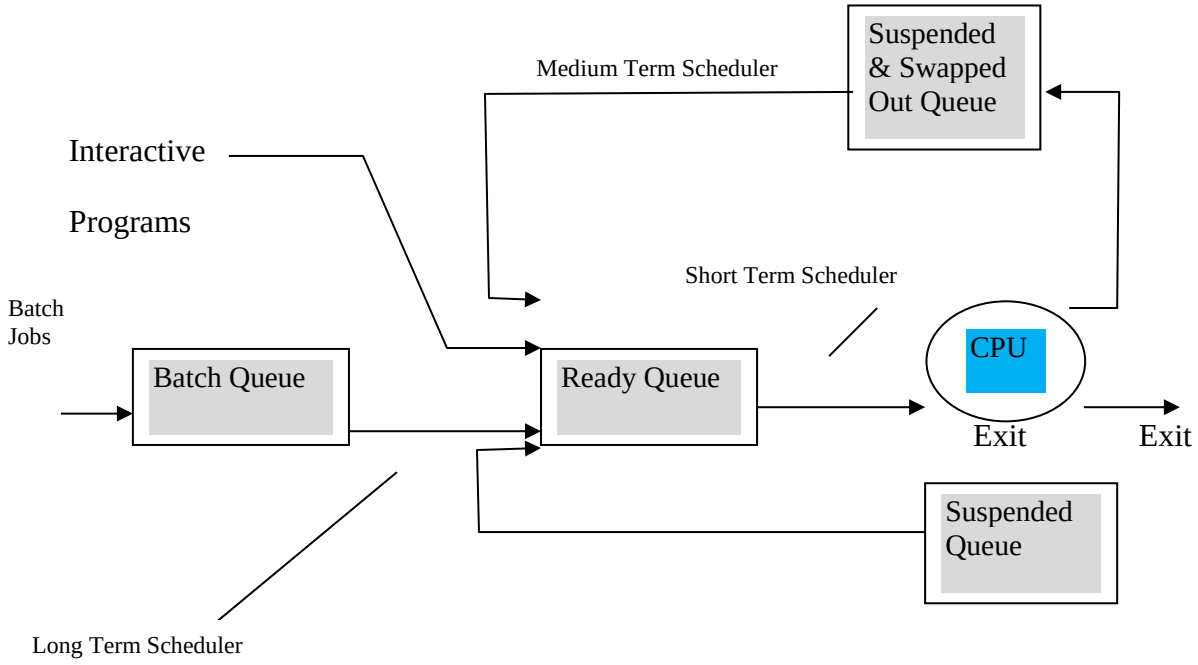


Figure 6 - Process Schedulers

3.4.2.1 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.

3.4.2.2 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:

3.4.2.3 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.

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.

3.5 Operations on Process

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

3.5.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, whereas the new processes are called the children of that process. Each of these new processes may in turn create other processes. In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

When a process is created it obtains, in addition to the various physical and logical resources, initialization data (or input) that may be passed along from the parent process to the child process. For example, consider a process whose function is to display the status of a file, say F1, on the screen of a terminal.

When it is created, it will get, as an input from its parent process, the name of the file F1, and it will execute using that datum to obtain the desired information.

It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, F1 and the terminal device, and may just need to transfer the datum between the two. When a process creates a new process, two possibilities exist in terms of execution: 1. The parent continues to execute concurrently with its children. 2. The parent waits until some or all of its children have terminated.

3.5.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call). All the resources of the process-including physical and virtual memory, open files, and I/O buffers-are deallocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another process via an appropriate system call (for example, abort). Usually, only the parent of the process that is to be terminated can invoke such a system call. Otherwise, users could arbitrarily kill each other's jobs. A parent therefore needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.

The task assigned to the child is no longer required. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.

3.6 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 toughput, 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.

3.6.1 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.

3.6.2 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.

3.7 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.

3.7.1 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).

3.7.2 Shortest Job Scheduling(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.

3.7.3 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.

3.7.4 Round Robin Scheduling

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.

3.7.5 Priority –Based Preemptive Scheduling

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.

3.7.6 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 STR

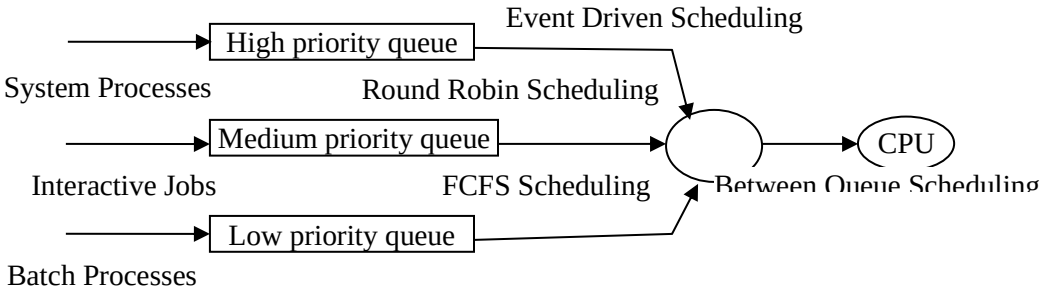


Figure 7: Multilevel Queue Scheduling

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.

3.8 Check Your Progress

1. The systems which allow only one process execution at a time, are called _____
   a) uniprogramming systems
   b) uniprocessing systems
   c) unitasking systems
   d) none of the mentioned
2. A process can be terminated due to _____
   a) normal exit
   b) fatal error
   c) killed by another process
   d) all of the mentioned
3. What is interprocess communication?
   a) communication within the process
   b) communication between two process
   c) communication between two threads of same process
   d) none of the mentioned
4. What is the ready state of a process?
   a) when process is scheduled to run after some execution
   b) when process is unable to run until some task has been completed
   c) when process is using the CPU
   d) none of the mentioned
5. A process stack does not contain _____
   a) Function parameters
   b) Local variables
   c) Return addresses
   d) PID of child process
6. The number of processes completed per unit time is known as _____
   a) Output
   b) Throughput
   c) Efficiency
   d) Capacity

7. The state of a process is defined by _____
   a) the final activity of the process
   b) the activity just executed by the process
   c) the activity to next be executed by the process
   d) the current activity of the process
8. What is a Process Control Block?
   a) Process type variable
   b) Data Structure
   c) A secondary storage section
   d) A Block in memory
9. A single thread of control allows the process to perform _____
   a) only one task at a time
   b) multiple tasks at a time
   c) only two tasks at a time
   d) all of the mentioned
10. The systems which allow only one process execution at a time, are called _____
    a) uniprogramming systems
    b) uniprocessing systems
    c) unitasking systems
    d) none of the mentioned
11. A process can be terminated due to _____
    a) normal exit
    b) fatal error
    c) killed by another process
    d) all of the mentioned
12. What is interprocess communication?
    a) communication within the process
    b) communication between two process
    c) communication between two threads of same process
    d) none of the mentioned

13. What is the ready state of a process?
    a) when process is scheduled to run after some execution
    b) when process is unable to run until some task has been completed
    c) when process is using the CPU
    d) none of the mentioned
14. A process stack does not contain _____
    a) Function parameters
    b) Local variables
    c) Return addresses
    d) PID of child process
15. The number of processes completed per unit time is known as _____
    a) Output
    b) Throughput
    c) Efficiency
    d) Capacity
16. The state of a process is defined by _____
    a) the final activity of the process
    b) the activity just executed by the process
    c) the activity to next be executed by the process
    d) the current activity of the process
17. What is a Process Control Block?
    a) Process type variable
    b) Data Structure
    c) A secondary storage section
    d) A Block in memory
18. A single thread of control allows the process to perform _____
    a) only one task at a time
    b) multiple tasks at a time
    c) only two tasks at a time
    d) all of the mentioned

3.9 Summary

State of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process control block (PCB). A process, when it is not executing, is placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue. Operating systems must provide a mechanism for parent processes to create new child processes. The parent may wait for its children to terminate before proceeding, or the parent and childrenmay execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.

3.10 Keywords

Long term scheduler is also known as job scheduler. It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory.

Short term scheduler is also known as CPU scheduler. It selects one of the Jobs from the ready queue and dispatch to the CPU for the execution.

Medium term scheduler takes care of the swapped out processes. If the running state processes needs some IO time for the completion then there is a need to change its state from running to waiting.

The **Attributes** of the process are used by the Operating System to create the process control block (PCB) for each of them

The total amount of time required by the CPU to execute the whole process is called the **Burst Time**.

3.11 Answers to check Your Progress

1. uniprocessing systems
2. all of the mentioned

3. communication between two process
4. when process is scheduled to run after some execution
5. PID of child process
6. Throughput
7. the current activity of the process
8. Data Structure
9. only one task at a time
10. uniprocessing systems
11. all of the mentioned
12. communication between two process
13. when process is scheduled to run after some execution
14. PID of child process
15. Throughput
16. the current activity of the process
17. Data Structure
18. only one task at a time

3.12 Reference/Suggested Readings

Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.

11. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
12. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
13. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
14. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson Number:4

Deadlock

4.1 Learning 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

4.2 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.

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.

4.2.1 Preemptable & Non-Preemptable 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.

4.2.2 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

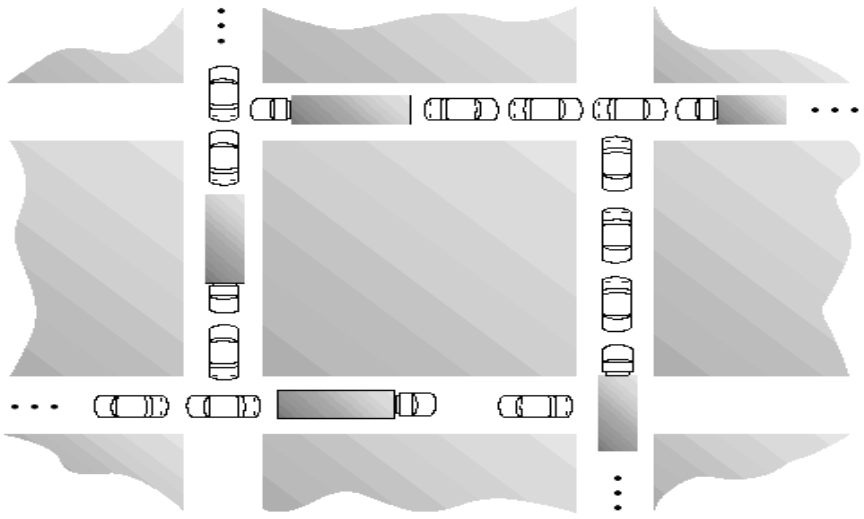


Figure 1: Consider each section of the street as a resource.

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.

4.3 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.

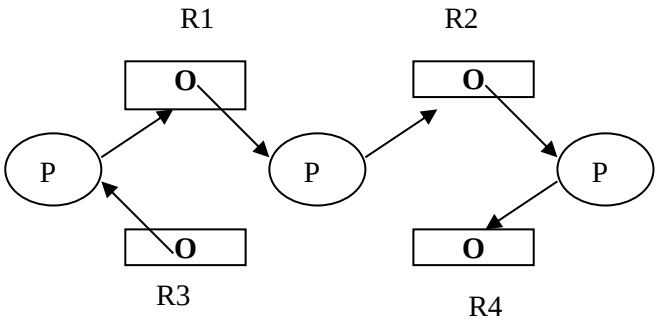
When a request is made, a request edge is added.

When request is fulfilled, the request edge is transformed into an assignment edge.

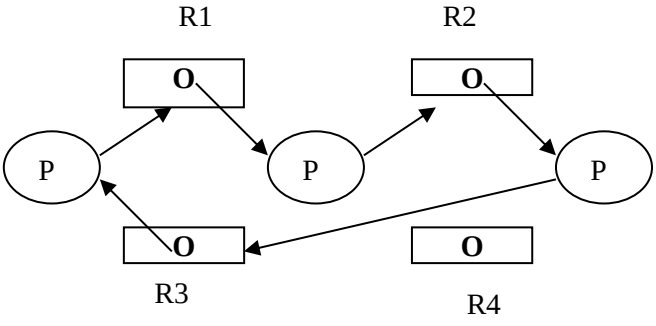
When process releases the resource, the assignment edge is deleted.

4.3.1 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

4.4 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.

4.5 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.

4.5.1 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.

4.5.2 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.

4.5.3 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"conditioneffectively.

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.

4.5.4 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.

4.6 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.

4.6.1 Banker's Algorithm

In this analogy

Customers	≡	processes
Units	≡	resources, say, tape drive
Banker	≡	Operating System

Customers	**Used**	**Max**	
A	0	6	Available Units

B	0	5	= 10
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	
B	1	5	Available Units
C	2	4	= 2
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 This is an unsafe state.

Customers	Used	Max	
A	1	6	
B	2	5	Available Units
C	2	4	= 1
D	4	7	

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.

4.6.2 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.

4.7 Check Your Progress

1. Which of the following condition is required for a deadlock to be possible?
   a) mutual exclusion
   b) a process may hold allocated resources while awaiting assignment of other resources
   c) no resource can be forcibly removed from a process holding it
   d) all of the mentioned

2. A system is in the safe state if _____
   a) the system can allocate resources to each process in some order and still avoid a deadlock
   b) there exist a safe sequence
   c) all of the mentioned
   d) none of the mentioned
3. A problem encountered in multitasking when a process is perpetually denied necessary resources is called _____
   a) deadlock
   b) starvation
   c) inversion
   d) aging
4. To avoid deadlock _____
   a) there must be a fixed number of resources to allocate
   b) resource allocation must be done only once
   c) all deadlocked processes must be aborted
   d) inversion technique can be used
5. The request and release of resources are _____
   a) command line statements
   b) interrupts
   c) system calls
   d) special programs
6. For a deadlock to arise, which of the following conditions must hold simultaneously?
   a) Mutual exclusion
   b) No preemption
   c) Hold and wait
   d) All of the mentioned
7. Deadlock prevention is a set of methods _____
   a) to ensure that at least one of the necessary conditions cannot hold
   b) to ensure that all of the necessary conditions do not hold

c) to decide if the requested resources for a process have to be given or not

d) to recover from a deadlock

8. A deadlock avoidance algorithm dynamically examines the _____ to ensure that a circular wait condition can never exist.

a) resource allocation state

b) system storage state

c) operating system

d) resources

9. If no cycle exists in the resource allocation graph _____

a) then the system will not be in a safe state

b) then the system will be in a safe state

c) all of the mentioned

d) none of the mentioned

10. If deadlocks occur frequently, the detection algorithm must be invoked _____

a) rarely

b) frequently

c) rarely & frequently

d) none of the mentioned

4.8 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.

4.9 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.

4.10 Self-Assessments Test

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?

4.11 Answers to check Your Progress

1. all of the mentioned
2. the system can allocate resources to each process in some order and still avoid a deadlock
3. starvation
4. there must be a fixed number of resources to allocate
5. system calls
6. All of the mentioned
7. to ensure that at least one of the necessary conditions cannot hold
8. resource allocation state
9. then the system will be in a safe state
10. frequently

4.12 Reference/Suggested Readings

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: 5

Memory Management

5.1 Learning Objectives

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.

5.2 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.

5.3 Logical & Physical Address Space

Physical address space in a system can be defined as the size of the main memory. It is really important to compare the process size with the physical address space. The process size must be less than the physical address space.

```
Physical Address Space = Size of the Main Memory

If, physical address space = 64 KB = 2 ^ 6 KB = 2 ^ 6 X 2 ^ 10 Bytes = 2 ^ 16 bytes

Let us consider,
word size = 8 Bytes = 2 ^ 3 Bytes

Hence,
Physical address space (in words) = (2 ^ 16) / (2 ^ 3) = 2 ^ 13 Words

Therefore,
Physical Address = 13 bits

In General,
If, Physical Address Space = N Words

then, Physical Address = log2 N
```

Logical address space can be defined as the size of the process. The size of the process should be less

enough so that it can reside in the main memory.

Logical Address Space = 128 MB = (2 ^ 7 X 2 ^ 20) Bytes = 2 ^ 27 Bytes
Word size = 4 Bytes = 2 ^ 2 Bytes

Logical Address Space (in words) = (2 ^ 27) / (2 ^ 2) = 2 ^ 25 Words
Logical Address = 25 Bits

In general,
If, logical address space = L words
Then, Logical Address = Log_2L bits

5.4 Contiguous Memory Management System

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.

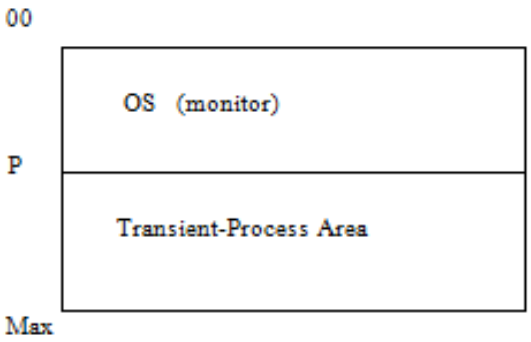


Figure 1: Single contiguous memory management

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.

5.4.1 Fixed Partitioned Memory Management 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.

5.4.1.1 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.

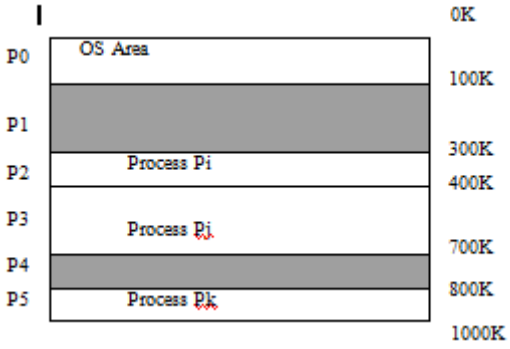


Figure 2 : Fixed Partitions

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.

5.4.1.2 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.

5.4.1.3 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.

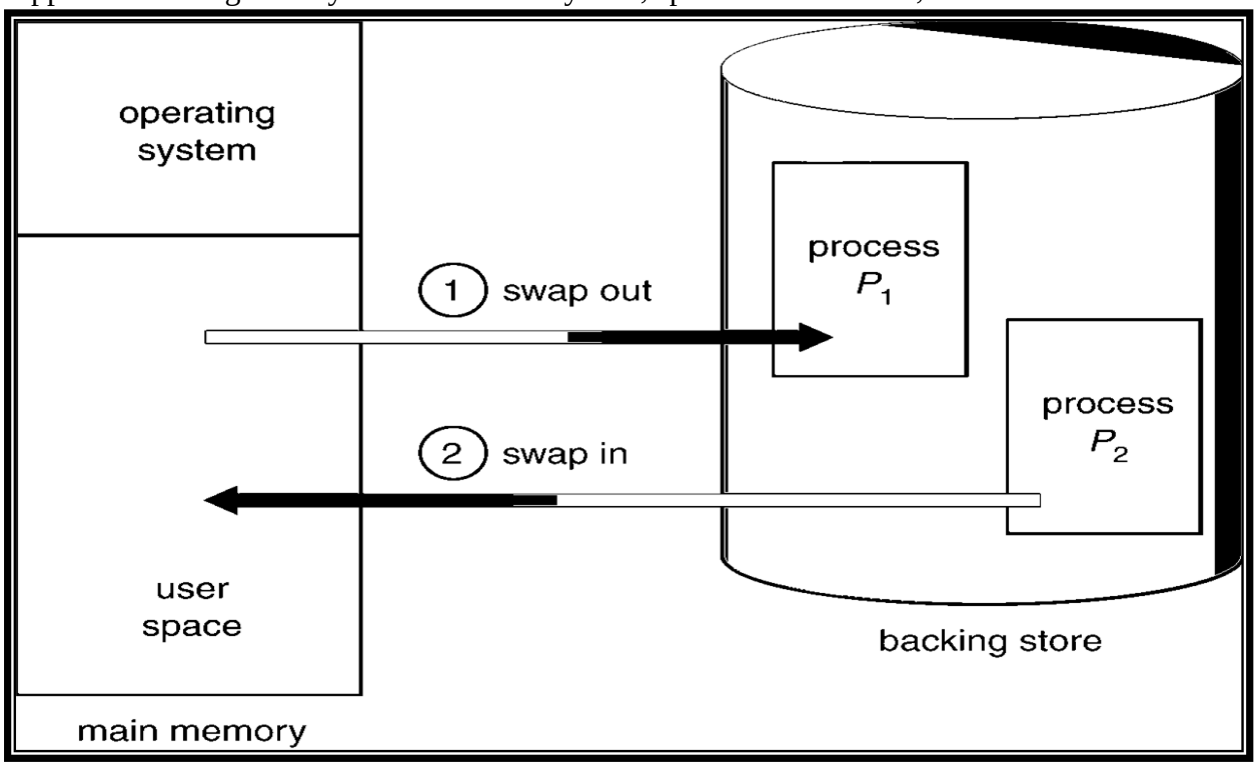


Figure 4: Showing process of Swapping

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.

5.4.1.4 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.

- **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.

- **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.

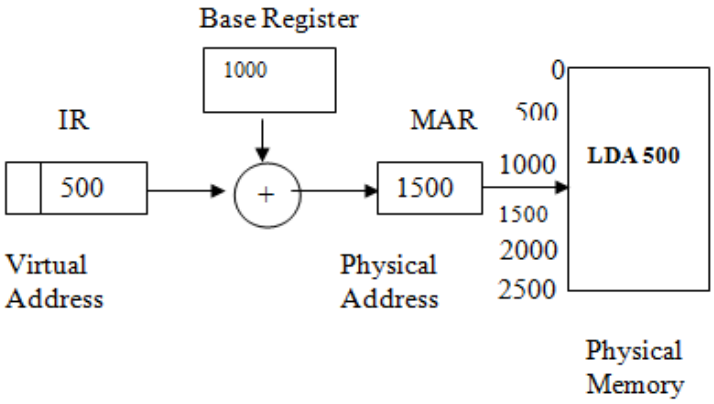


Figure 5 : Dynamic relocation

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 5, 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.

5.4.1.5 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.

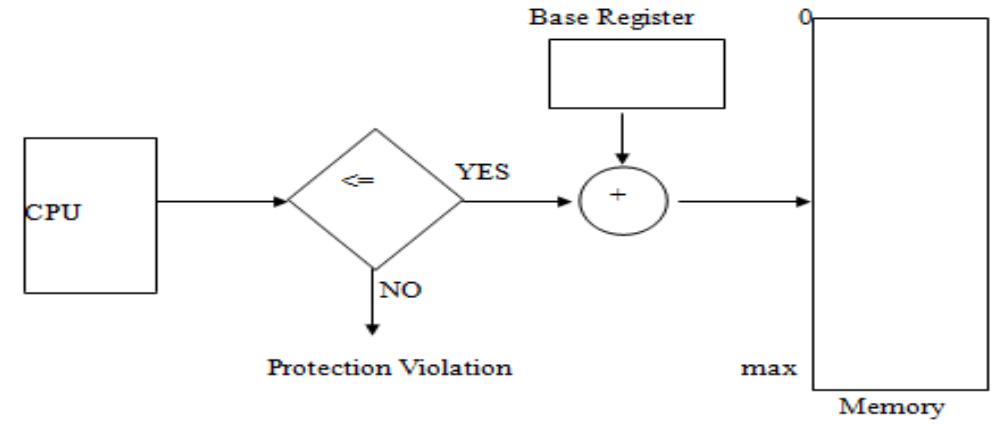


Figure 6 – Base-limit register

5.4.1.6 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 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.

5.4.1.7 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.

5.4.2 Variable Memory Management System

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.

5.4.2.1 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.

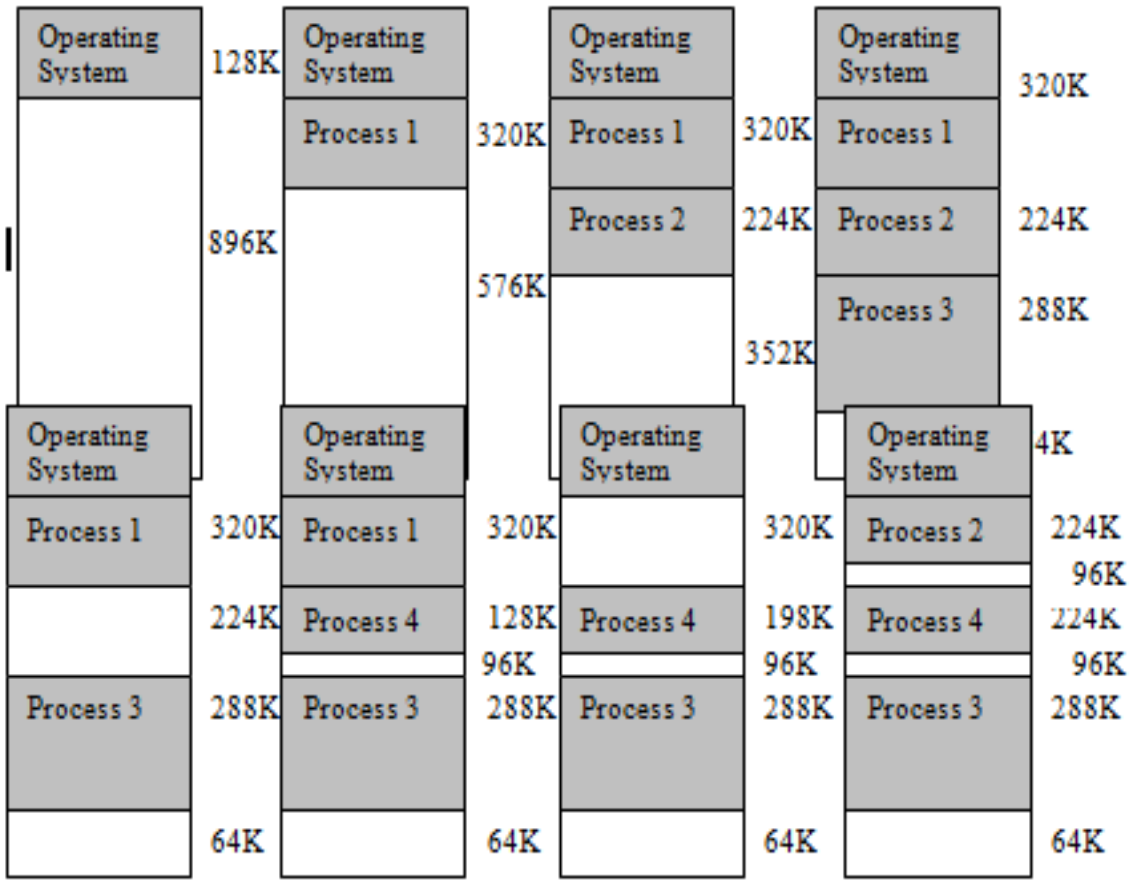


Figure 7: Partitions in dynamic memory partitioning

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 7 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 utilization.

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.

5.4.2.2 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.

5.4.2.3 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.

5.4.2.4 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.

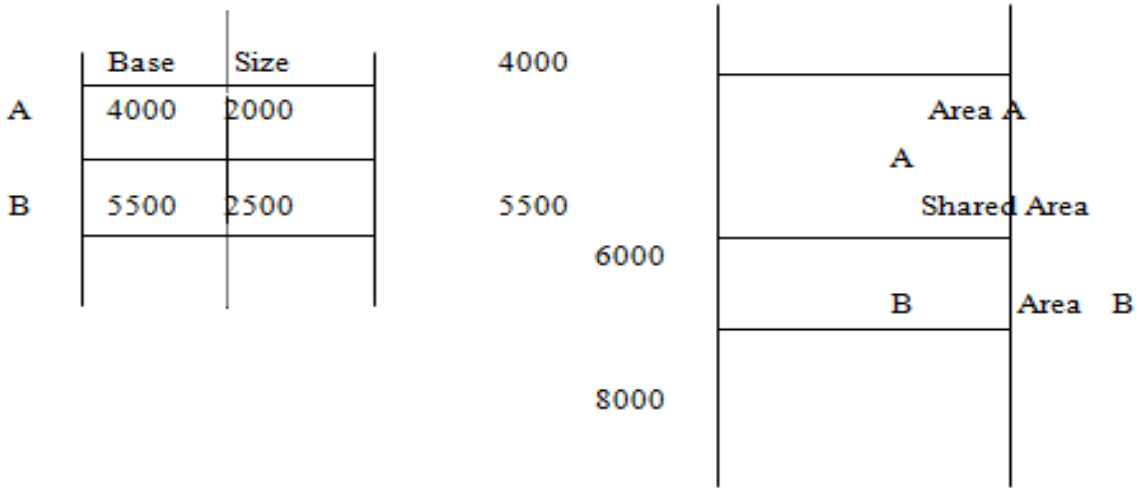


Figure 8: Overlapping partitions

Some aspects of the issues involved in self-referencing of shared code are illustrated in Figure 8, 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.

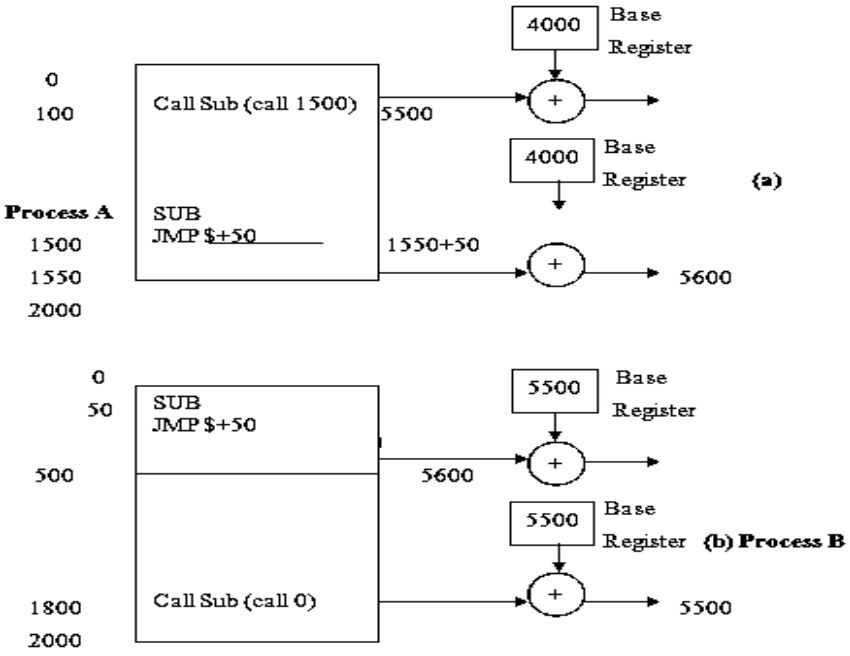


Figure 9: Accessing shared code (a) Process A (b) Process B

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.

5.4.2.5 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.

5.5 Non Contiguous Memory allocation

Non-Contiguous memory allocation is basically a method on the contrary to contiguous allocation method, allocates the memory space present in different locations to the process as per it's requirements. As all the available memory space is in a distributed pattern so the freely available memory space is also scattered here and there.

This technique of memory allocation helps to reduce the wastage of memory, which eventually gives rise to Internal and external fragmentation.

Figure 10: Non Contiguous Memory allocation

5.6 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.

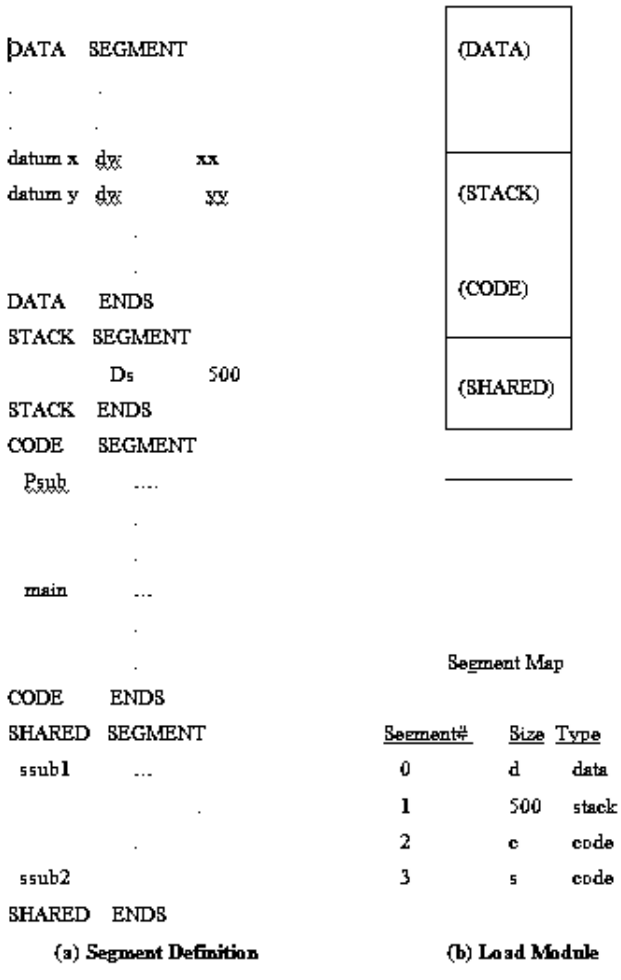


Figure 11 - Segments

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-11(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".

5.6.1 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.

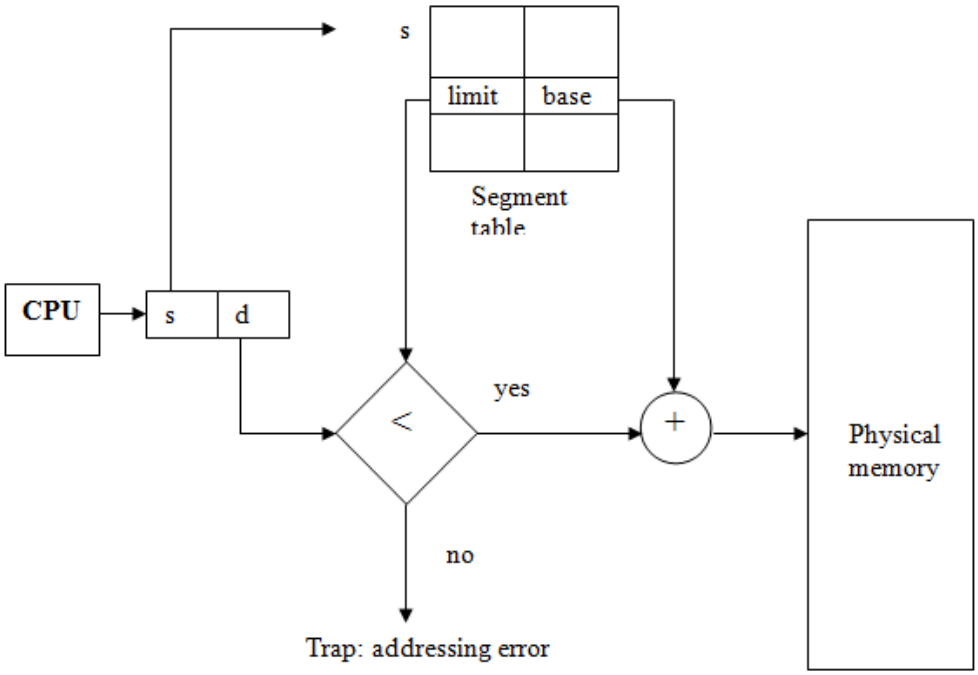
For example, the subroutine SSUB2 in segment SHARED is assumed to begin at offset 100. However, the unique designation of an item in a segmented address space requires the specification of both its segment & the relative offset therein. Offset 100 may fetch the first instruction of the subroutine SSUB2 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 11(a) is depicted in Figure 11(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 physicalmemory.



Hardware support for Segmentation

Figure 12 : 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 12 illustrates a sample placement of the segments defined in Figure 11 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 11(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.

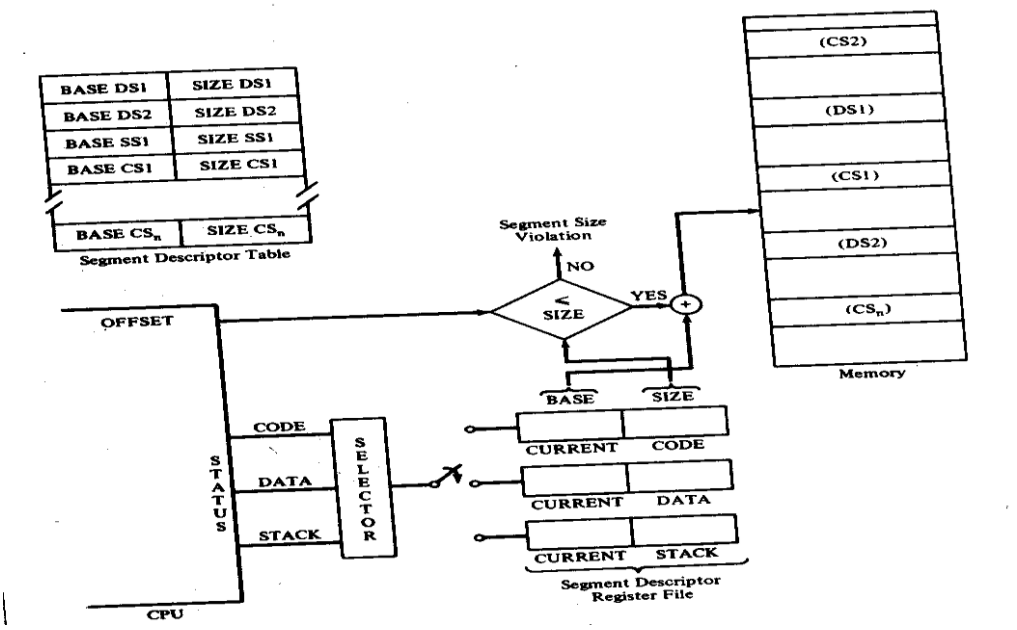


Figure 13 : Segment-descriptor cache registers

Register-assisted translation of virtual to physical addresses is illustrated in Figure 13. 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.

5.6.2 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.

5.6.3 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 14 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

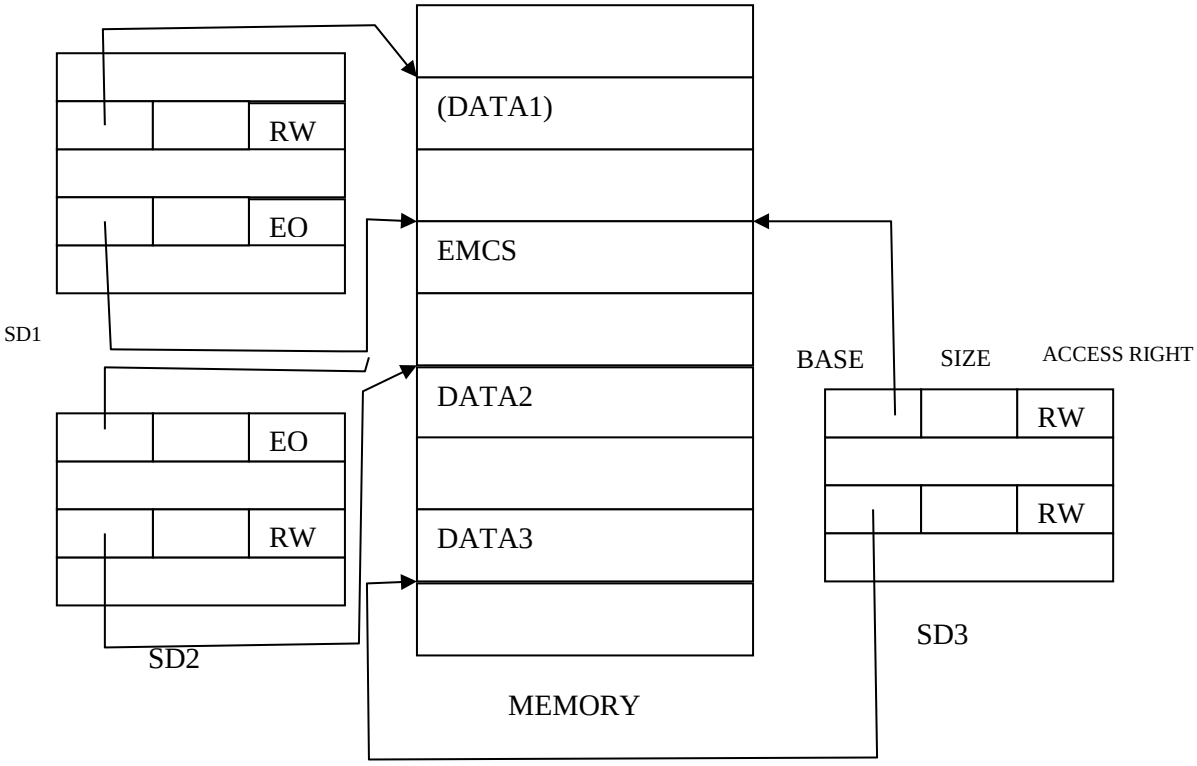


Figure 14: Sharing in segmented systems

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.

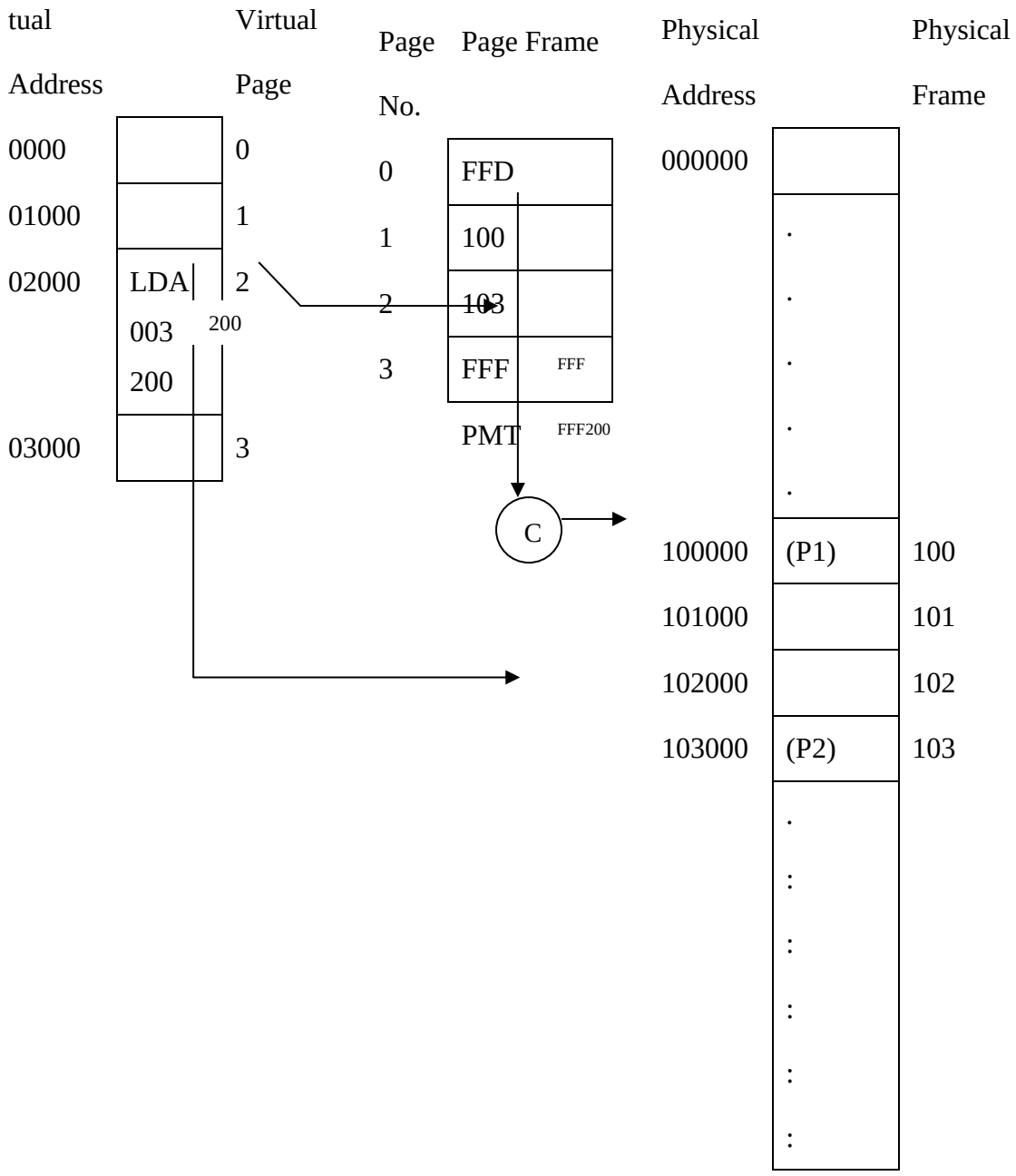
As described, sharing is encouraged in segmented systems. This presents some problems in systems that also support swapping, which is normally done to increase processor utilization. For example, a shared segment may need to maintain its memory residence while being actively used by any of the processes 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.

5.7 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.

5.7.1 Principles of Operation



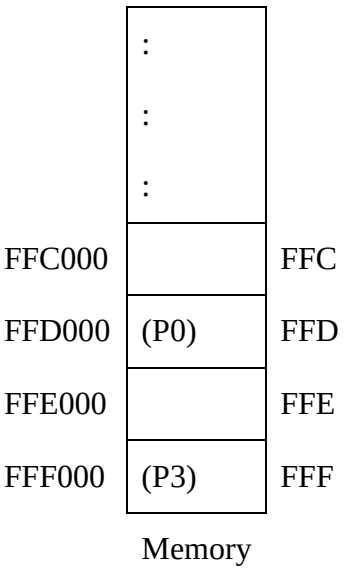


Figure 15: Paging

Figure 15 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.

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.

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.

Figure 16 : Memory-map table (MMT)

5.7.2 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.

5.7.3 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.

0Associative 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.

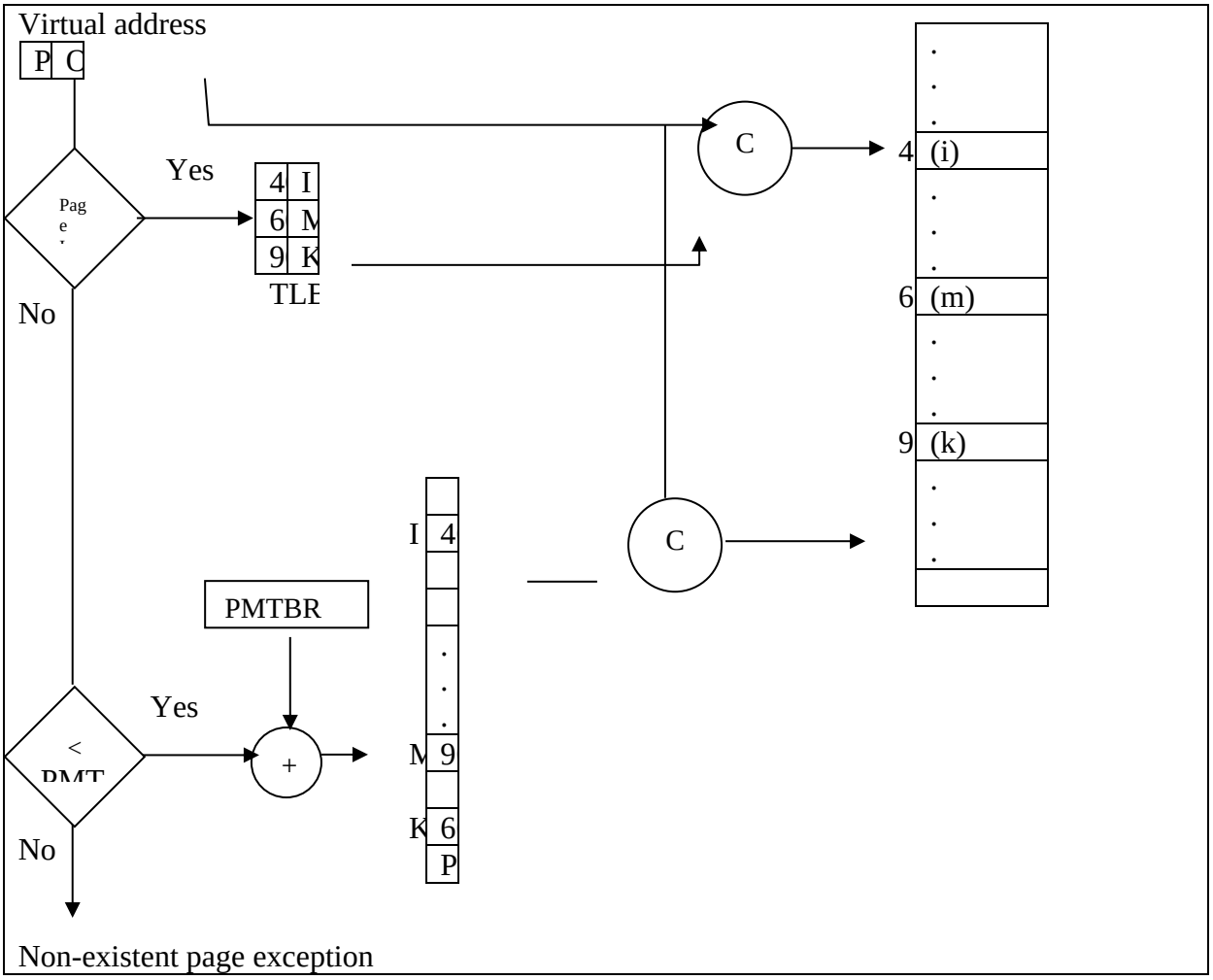


Figure 17 : Translation-lookaside buffer (TLB)

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 17 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 $0 \le h \le 1$; 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.

5.7.4 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.

5.8 Virtual Memory Management

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.

5.8.1 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, … , v-1}, & the physical memory space by M = {0, 1, ….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

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

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 18. 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 18.

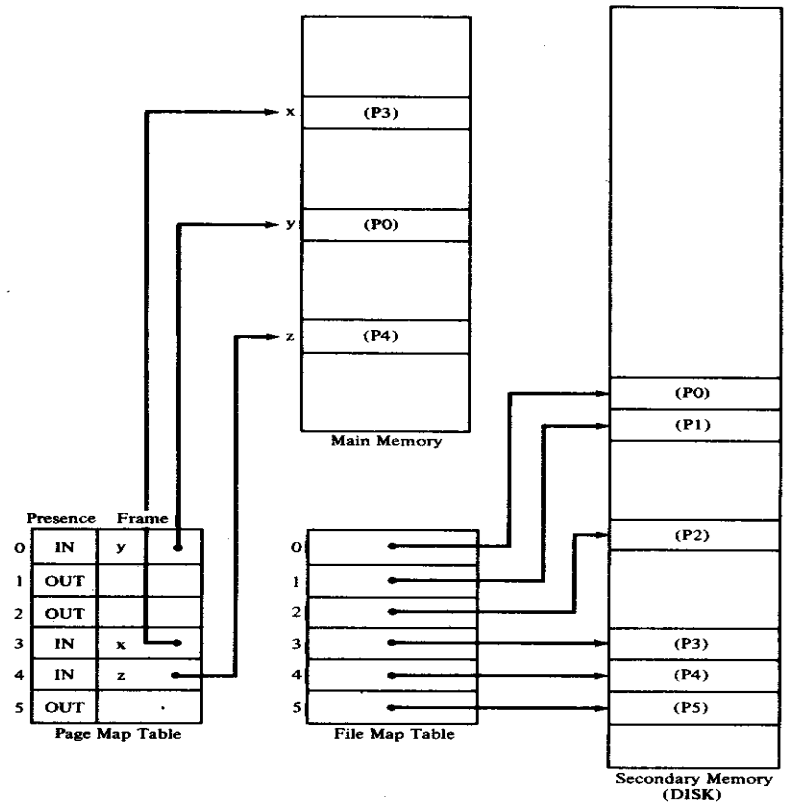


Figure 18: Virtual memory

5.8.2 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

5.9 Demand Paging

According to the concept of Virtual Memory, in order to execute some process, only a part of the process needs to be present in the main memory which means that only a few pages will only be present in the main memory at any time.

However, deciding, which pages need to be kept in the main memory and which need to be kept in the secondary memory, is going to be difficult because we cannot say in advance that a process will require a particular page at particular time.

Therefore, to overcome this problem, there is a concept called Demand Paging is introduced. It suggests keeping all pages of the frames in the secondary memory until they are required. In other words, it says that do not load any page in the main memory until it is required.

5.10 Page Replacement Algorithm

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.

5.11 Demand Segmentation

Operating system also uses **demand segmentation**, which is similar to demand paging. Operating system to uses demand segmentation where there is insufficient hardware available to implement 'Demand Paging'.

The segment table has a valid bit to specify if the segment is already in physical memory or not. If a segment is not in physical memory then segment fault results, which traps to the operating system and brings the needed segment into physical memory, much like a page fault.

Demand segmentation allows for pages that are often referenced with each other to be brought into memory together, this decreases the number of page faults.

Another space server would be to keep some of a segment's page tables on disk and swap them into memory when needed.

5.12 Check Your Progress

1. What is Address Binding?
   a) going to an address in memory
   b) locating an address with the help of another address
   c) binding two addresses together to form a new address in a different memory space
   d) a mapping from one address space to another
2. If the process can be moved during its execution from one memory segment to another, then binding must be _____
   a) delayed until run time
   b) preponed to compile time
   c) preponed to load time
   d) none of the mentioned
3. The _____ swaps processes in and out of the memory.
   a) Memory manager
   b) CPU
   c) CPU manager
   d) User
4. In a system that does not support swapping _____
   a) the compiler normally binds symbolic addresses (variables) to relocatable addresses
   b) the compiler normally binds symbolic addresses to physical addresses
   c) the loader binds relocatable addresses to physical addresses
   d) binding of symbolic addresses to physical addresses normally takes place during execution
5. CPU fetches the instruction from memory according to the value of _____
   a) program counter
   b) status register
   c) instruction register
   d) program status word

6. A memory buffer used to accommodate a speed differential is called _____
   a) stack pointer
   b) cache
   c) accumulator
   d) disk buffe

7. Which one of the following is the address generated by CPU?
   a) physical address
   b) absolute address
   c) logical address
   d) none of the mentioned

8. Run time mapping from virtual to physical address is done by _____
   a) Memory management unit
   b) CPU
   c) PCI
   d) None of the mentioned

5.13 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.

5.14 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.

5.15 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?

5.16 Answers to Check Your Progress

1. a mapping from one address space to another
2. delayed until run time

3. Memory manager
4. the compiler normally binds symbolic addresses (variables) to relocatable addresses
5. program counter
6. cache
7. logical address
8. Memory management unit

5.17 Reference/Suggested Readings

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

Lesson Number: 6

File System

6.1 Learning 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.

6.2 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."

6.3 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:

a) Minimal I/O operations.

b) Flexible file naming facilities.

c) Automatic allocation of file space.

d) Dynamic allocation of file space.

e) Unrestricted flexibility between logical record size and physical block size.

f) Protection of files against illegal forms of access.

g) Static and dynamic sharing of files.

h) 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.

6.3.1 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

6.3.2 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.

6.3.3 File Types

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an *extension,* usually separated by a period character figure --.In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension of additional characters. File name examples include *resume.doc, Server.java etc.* The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a *.com, .exe,* or *.bat* extension can be *executed,* for instance. The *.com* and *.exe* files are two forms of binary executable files, whereas a

.bat file is a **batch file** containing, in ASCII format, commands to the operating system

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Figure 1: File Types

The files under UNIX can be categorized as follows:

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

6.3.4 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.

6.3.5 File Sharing & 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 updating 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.

6.3.6 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.

6.3.7 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.

6.4 Access Method

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.

6.4.1 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.

6.4.2 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 arecord 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.

6.4.3 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.

6.5 Directory Structures

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.

6.5.1 The Logical Structure of a Directory

6.5.1.1 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.

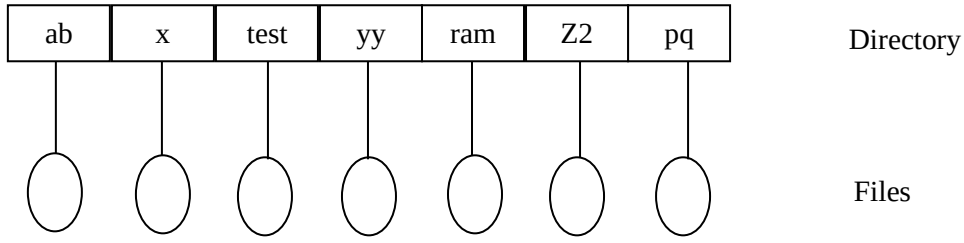


Figure 2: Single-level Directory

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.

6.5.1.2 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.

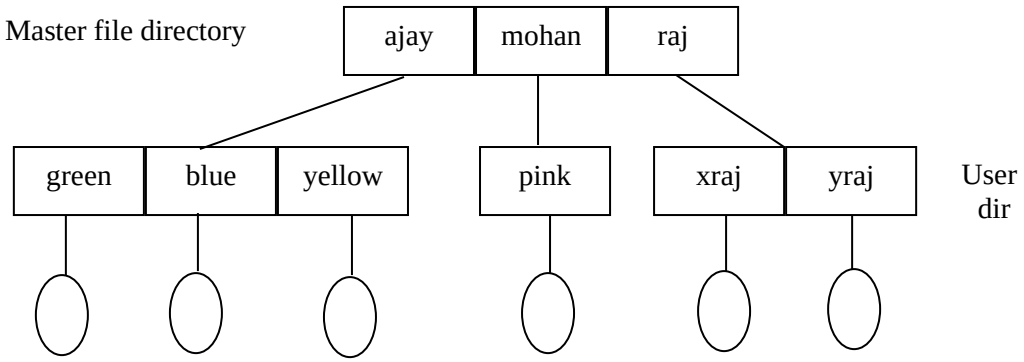


Figure 3: Two-Level Directory

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.

6.5.1.3 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.

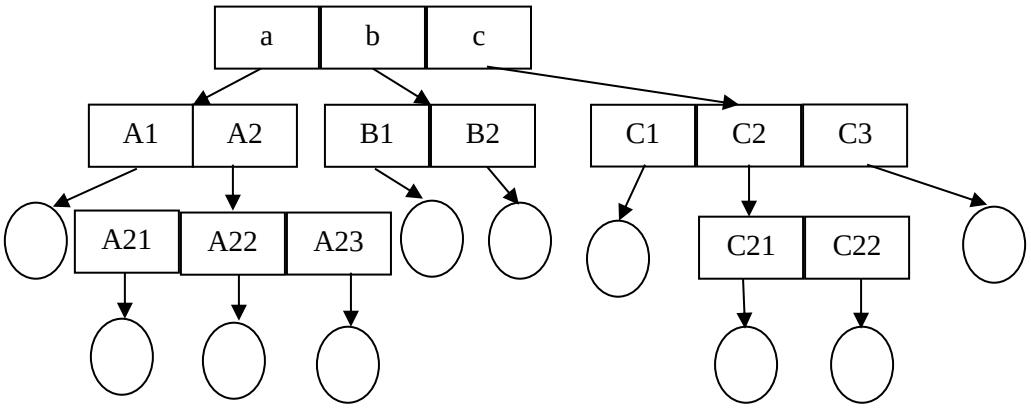


Figure 4 : Tree-Structured Directories

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.

6.5.1.4 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.

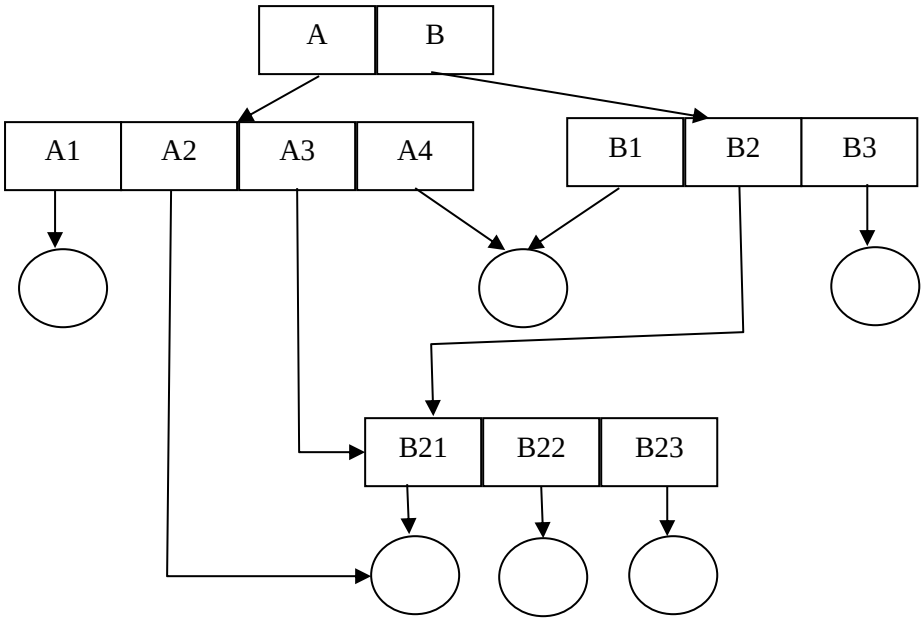


Figure 5: Acyclic-Graph Directories

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.

6.5.1.5 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.

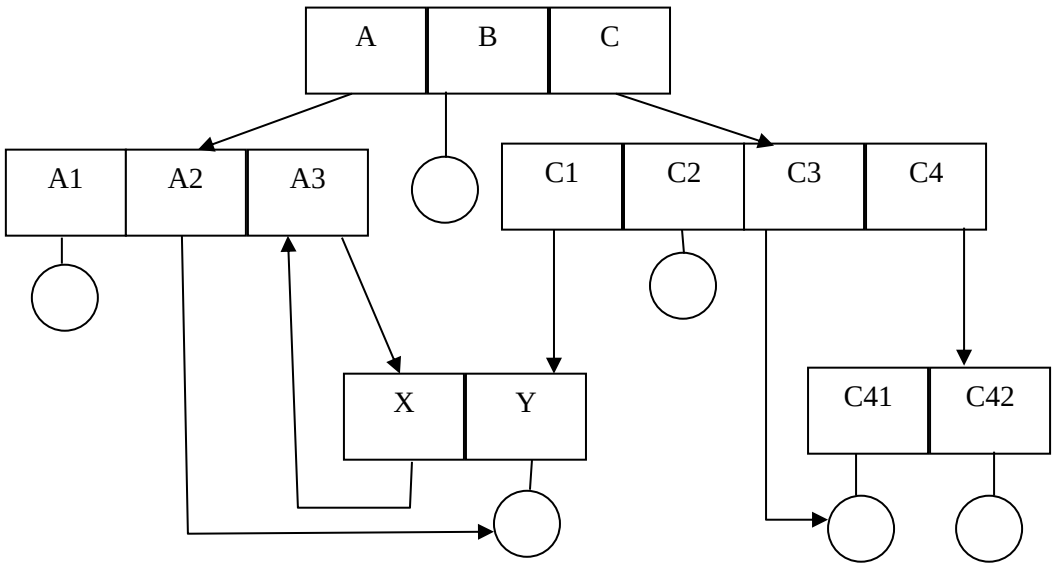


Figure 6: General Graph Directory

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.

6.6 Allocation Methods

6.6.1File Space allocation

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.

6.6.1.1 Contiguous Space Allocation

In contiguous allocation, each file occupies a set of contiguous blocks on the disk. If only one job is accessing the disk, accessing block b+1 after block b does not require any head movement. It is enough to move the next sector under the current head position. Therefore, disk seeks are minimal. This method is simple to implement. It is enough to remember only starting location (starting disk block number) and the length of the file (number of blocks in the file).

The directory entry has the name of the file and the length of the file. Figure 35.1 shows an example of how files are allocated contiguously. In the example shown, there is a file named count which starts from disk block number 0 and the length of the file is 2 blocks. That is, disk block number 0 and disk block number 1 are allocated for the file. Similarly, the file named mail starts from disk block number 19 and the length is 6 disk blocks.

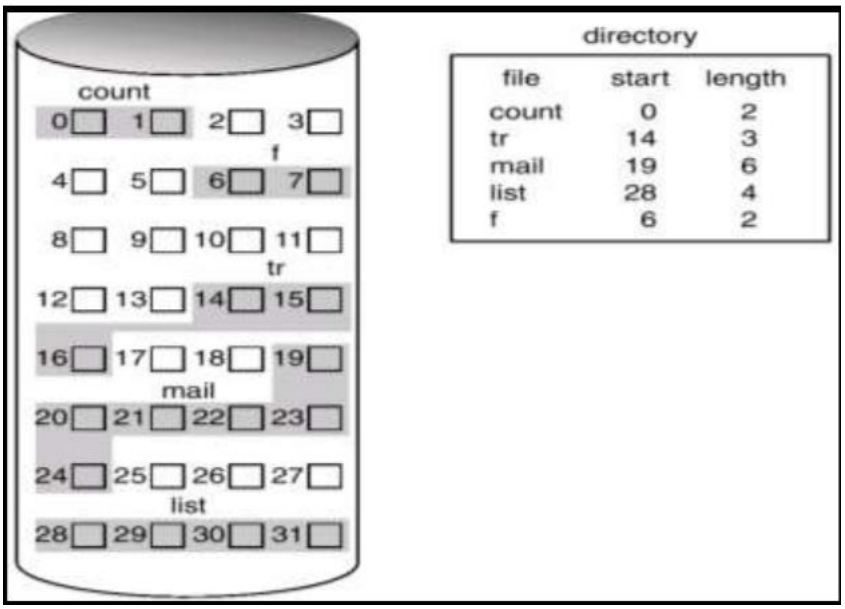


Figure 7: Directory Structure

Both sequential access and direct access are possible when files are contiguously allocated. In the

example shown in Figure 2, consider the file mail. To access the file sequentially, the starting disk block number (19) is taken from the directory entry and the subsequent disk blocks are accessed one after the other. For direct access, the first disk block number is taken from the directory entry and the nth disk block is calculated by adding n to the first disk block number. For example, it is easy to calculate that the third disk block of the file is 21.

In contiguous allocation, the dyamic storage-allocation problem has to be sorted out. Suppose there is a list of free holes and it is necessary to satisfy a request of size n from this list of free holes. It is possible to use first fit or best fit allocation strategy. In first fit allocation, the first hole that is large enough to accommodate n blocks is chosen for allocation. In best fit strategy, the smallest hole that is large enough to accommodate n blocks is chosen for allocation.

Another issue with contiguous allocation is external fragmentation. As files are created and deleted, in due course of time, free disk blocks may not be present contiguously. But there may be free disk blocks here and there. There may be n free disk blocks in the disk, but the n free blocks may not be present contiguously. Now, when a file of size n is to be brought in, it cannot be accommodated. To overcome this external fragmentation, compaction is needed.

The next issue is that it is difficult to determine how much space is needed for a file. Initially, when a file is created, the file may occupy only a few bytes. But, it may grow over time. It is not possible to know beforehand the size to which the file will grow. One way to overcome this is to overestimate the space needed and allocate a larger number of blocksfor the file. But, this results in wastage of space. Some of the allocated blocks may not be used by the file. The second way is to find a larger hole, copy the file contents to the new space and free the old space whenever the file grows. Even if the file size in known in advance, the file may grow over a long period of time. Till then, the extra space allocated remains free and unused. This results in internal fragmentation. To overcome internal fragmentation, a modified contiguous allocation is used. Initially, a contiguous chunk of space is allocated for the file. If this is not large enough, another chunk of space (extent) is allocated. The Veritas file system uses this strategy.

6.6.1.2 Linked Allocation

The linked allocation solves all problems of contiguous allocation. In linked allocation, each file is a

linked list of disk blocks. The allocated disk blocks may be scattered anywhere on the disk. The directory entry has a pointer to the first and the last blocks of the file. Figure 35.2 shows an example of linked allocation. There is a file named jeep. The starting disk block number 9 is given in the directory entry. Disk block 9 has the address of the next disk block, 16. Disk block 16 points to disk block 1 and so on. The last disk block of the file, disk block 25 has a pointer value of -1 indicating that it is the last disk block of the file.

This allocation method does not suffer from external fragmentation. The size of a file need not be declared when the file is created. The file can continue to grow as long as free blocks are available. Since there is no external fragmentation, compaction is not needed. The linked allocation also has some limitations. There is no possibility for random access because it is necessary to follow the pointers from one disk block to another to access the disk blocks. Only sequential-access is allowed.

The second limitation is the space required for pointers. Each disk block stores a pointer and this occupies space. To avoid this, blocks are collected into groups called clusters (say, 4 blocks). Rather than having a pointer from each disk block to the next block, there is a pointer from one cluster (group of blocks) to another cluster. This results in having less number of pointers. This also results in fewer disk-head seeks, because, within a cluster there is no need for the movement of the head. But this may lead to increased internal fragmentation. If a cluster is allocated for a file and if all the blocks in the cluster are not used, then that space is wasted and results in internal fragmentation. The third limitation is reliability. Many pointers are in use. If the pointers are lost or damaged, it results in loss of information. To improve reliability, a doubly linked list can be used rather than using a singly linked list so that even if one of the pointers between two blocks is lost, the other will still be available. Another improvement is to store the file name and the relative block number in each block. The pointers are still there, but after moving to a disk block using a pointer, the file name and the relative block number can be checked to confirm if the block really belongs to the file. A variation of linked allocation scheme is used in the File-allocation table (FAT). This is the disk-space allocation scheme used by MS-DOS. A section of the disk at the beginning of each volume contains this table (FAT). Figure 3 shows an example file allocation table. The directory entry has the address of the first disk block number (217). This 217 is used as an index into the FAT to get the entry where 618 is stored. This means that the next disk block number is 618. While using 618 as an index into the FAT, 339 is

got, which means that, 339 is the next disk block. When 339 is used as index, the entry shows that end of file is reached.

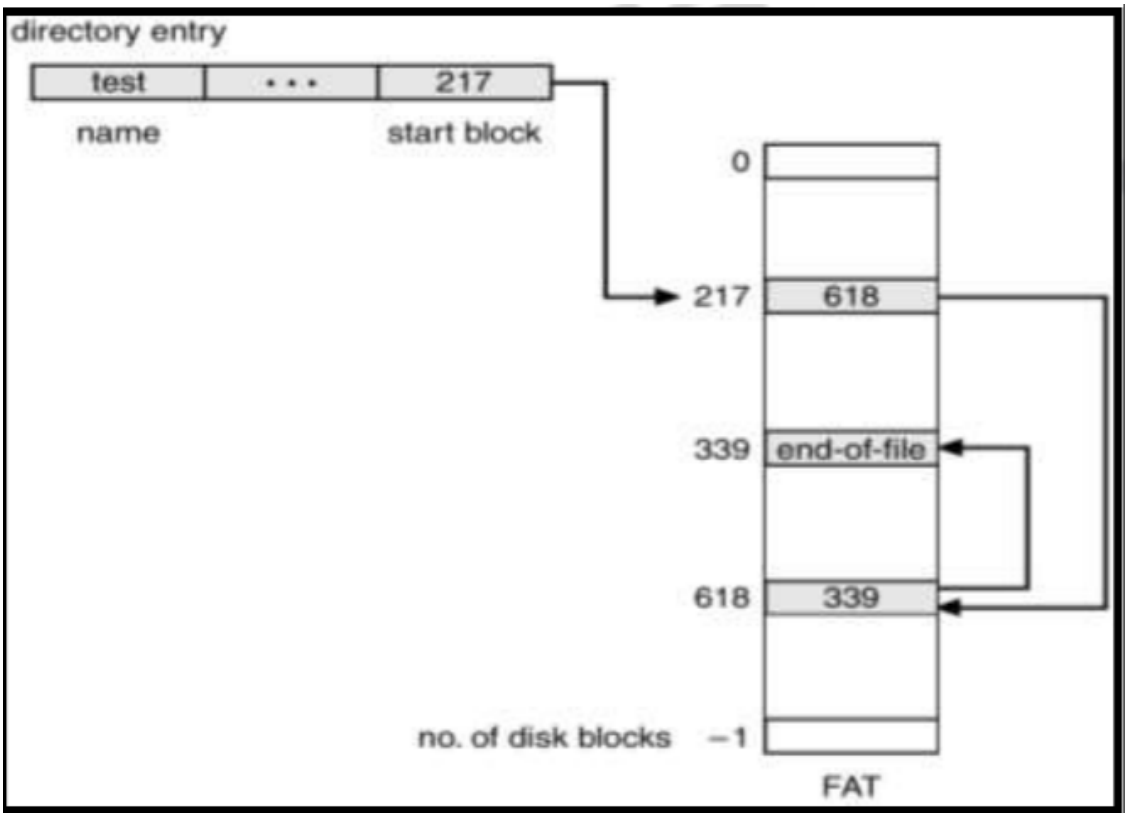


Figure 8: file allocation table

6.6.1.3 Indexed Allocation

The indexed allocation solves the external fragmentation and size-declaration problem of contiguous allocation. It also solves the direct access problem in linked allocation. In indexed allocation all pointers are brought together into one block called the index block. Each file has an index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file.

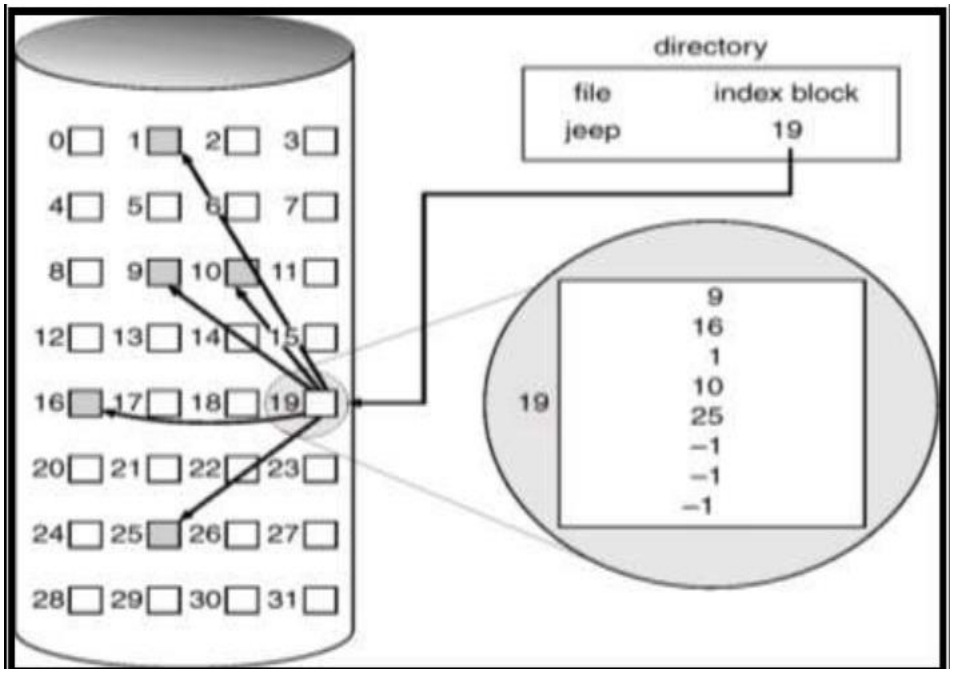


Figure 9: Index allocation

index block (disk block 19). The index block has the addresses of the disk blocks (9, 16, 1, 10 and 25) of the file. Indexed allocation supports sequential as well as direct access. For direct access, to access the i th block, the i th entry in the index block is taken. There is no external fragmentation because any disk block from any position can be used. The limitation of indexed allocation is wastage of space. When there is a file occupying only two blocks, in linked allocation, space for only two pointers is wasted, in indexed allocation, one whole block is wasted. Another issue to be considered is the size of the index block. How large the index block should be? The index block should be as small as possible. But, if the index block is too small, the index block may not be able to hold enough pointers for a large file. Consider a file of maximum size of 256K (218) words and block size of 512 (29) words. The file occupies 29 disk blocks (218 / 29). To accommodate the addresses of 29 disk blocks, one index block is needed (because, here, one disk block size is 29 words). There are different variants of index blocks. – Linked scheme – In this scheme, the index blocks are linked together. An index block will contain the addresses of disk blocks of the file as well as the address of the next index block. Therefore, there is no limit on the size of the index block and the size of the file. Two-level index – In this scheme, each entry in the first-level index block points to a second-level index block and each entry in the second-level index block points to disk blocks having the contents of the file. Figure 5 shows diagrammatically this

variant of index block. The directory entry points to an outer index block. Each entry in the outer index block points to an inner index block. Each entry in the inner index block points to a disk block with file contents. If the size of a disk block is 512 bytes, then the maximum file size possible with this scheme is 512^3

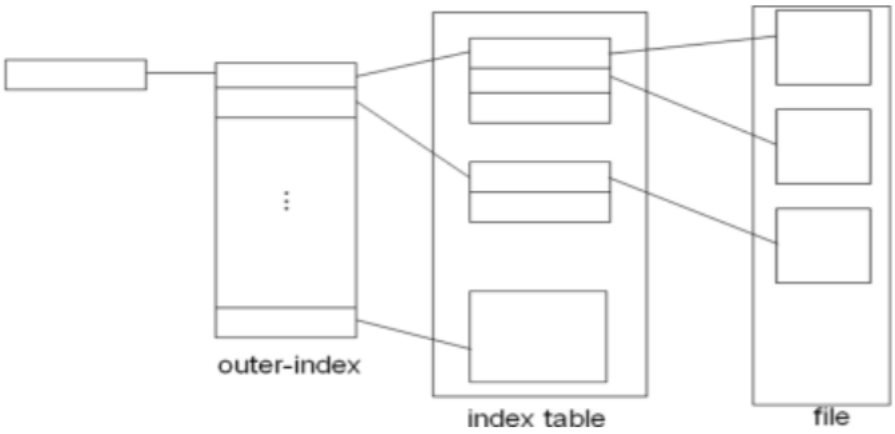


Figure 10 : Index allocation

Figure6 shows the scheme used in UNIX systems. In this scheme, the inode has the index of disk block addresses. There are 10 entries in the inode table which are the addresses of the disk blocks of the file. The 11th entry points to an index block, which in turn has the addresses of disk blocks of the file (single level indirection). The 12th entry uses two-level indirection and the 13th entry uses three-level indirection. When the size of the file is small, only the direct disk blocks are used. Since the size of each disk block is 4K bytes, a file of size upto 40K can be addressed using these direct disk blocks. When the size of the file is more than 40K, the next levels are used.

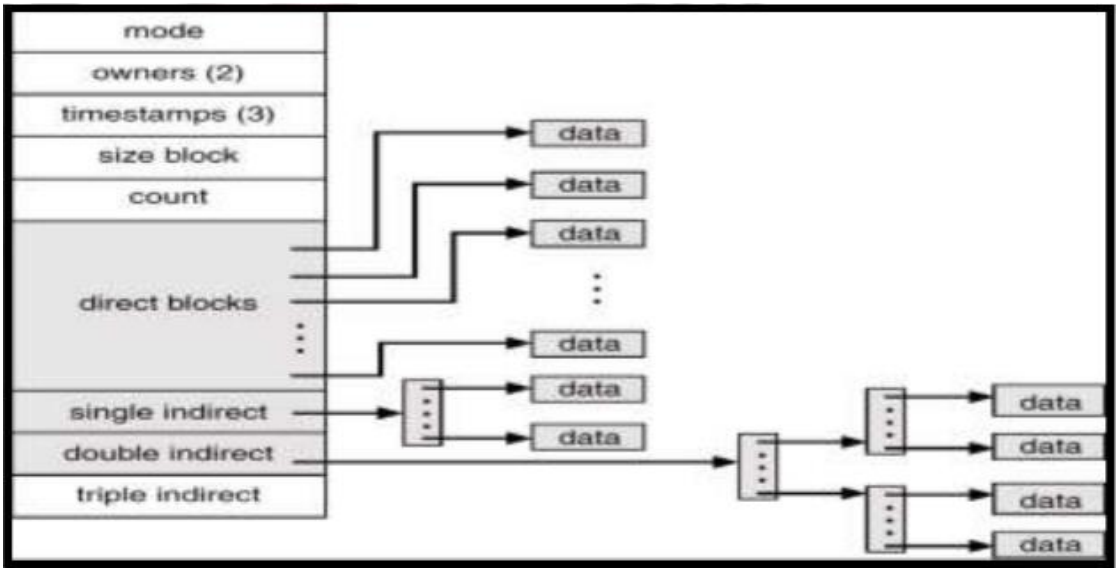


Figure 11 : Unix Index allocation

6.6.1.4 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.

6.6.1.5 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.

6.7 Check your progress

1. _____ is a unique tag, usually a number identifies the file within the file system.

a) File identifier

b) File name

c) File type

d) None of the mentioned

2. To create a file _____

a) allocate the space in file system

b) make an entry for new file in directory

c) allocate the space in file system & make an entry for new file in directory

d) none of the mentioned

3. By using the specific system call, we can _____

a) open the file

b) read the file

c) write into the file

d) all of the mentioned

4. File type can be represented by _____

a) file name

b) file extension

c) file identifier

d) none of the mentioned

5. Which file is a sequence of bytes organized into blocks understandable by the system's linker?

a) object file

b) source file

c) executable file

d) text file

6. What is the mounting of file system?

a) crating of a filesystem

b) deleting a filesystem

c) attaching portion of the file system into a directory structure

d) removing the portion of the file system into a directory structure

7. Mapping of file is managed by _____

a) file metadata

b) page table

c) virtual memory

d) file system

8. Mapping of network file system protocol to local file system is done by _____

a) network file system

b) local file system

c) volume manager

d) remote mirror

9. Which one of the following explains the sequential file access method?

a) random access according to the given byte number

b) read bytes one at a time, in order

c) read/write sequentially by record

d) read/write randomly by record

10. When will file system fragmentation occur?

a) unused space or single file are not contiguous

b) used space is not contiguous

c) unused space is non-contiguous

d) multiple files are non-contiguous

6.8 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 optimized.

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.

6.9 Keywords

FIFO Files :FIFO (first-in-first-out) are files that allow unrelated processes to communicate with each other.

Contiguous allocation, each file occupies a set of contiguous blocks on the disk.

Indexed allocation all pointers are brought together into one block called the index block.

Sequential file is the most primitive of all file structures it has no directory and no linking pointers

Direct access file organization, any records can be accessed irrespective of the current position in the file

6.10 Self-Assessments Test

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.
6. What are different types of access method explain with the help of example.

6.11 Answers to Check Your Progress

1. File identifier
2. allocate the space in file system & make an entry for new file in directory
3. all of the mentioned
4. file extension
5. object file
6. attaching portion of the file system into a directory structure
7. file metadata
8. network file system
9. read bytes one at a time, in order
10. unused space or single file are not contiguous

6.12 Reference/Suggested Readings

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: 7

Disk Scheduling

7.1 Learning 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.

7.2 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.

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

7.3 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

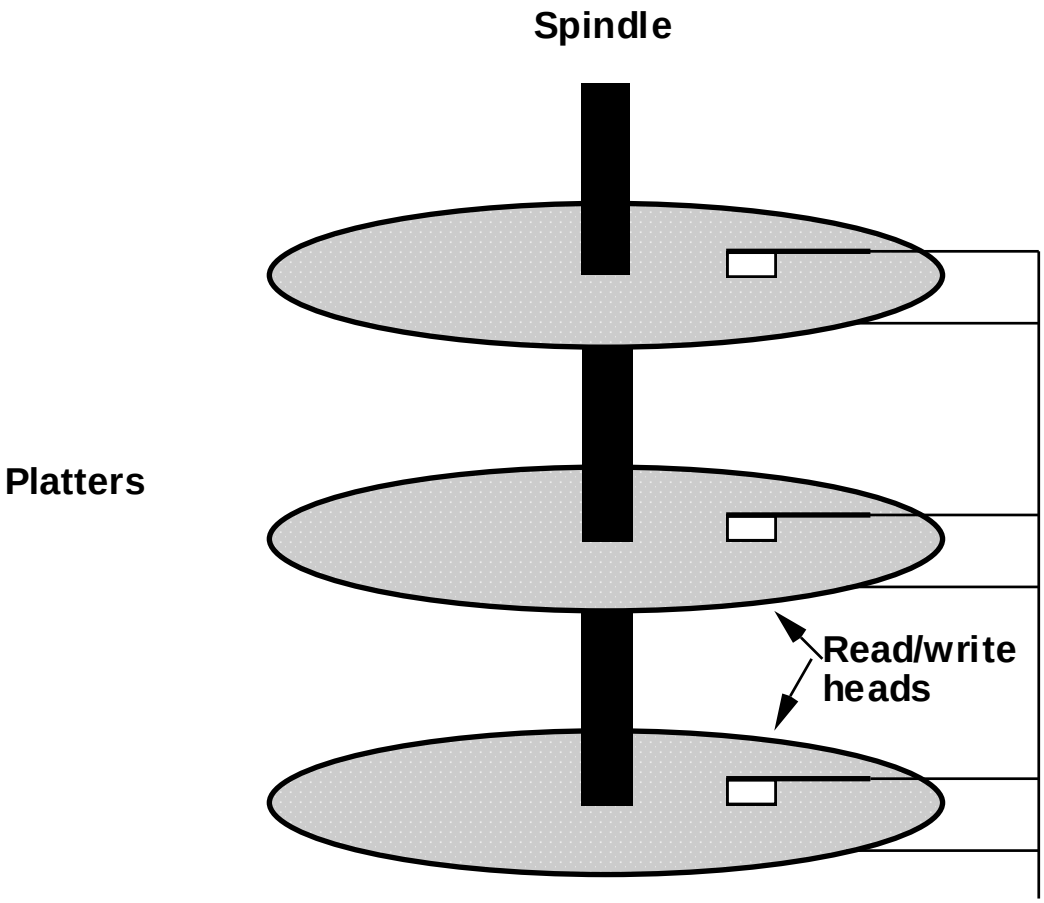


Figure 1: Moving head disk mechanism

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.

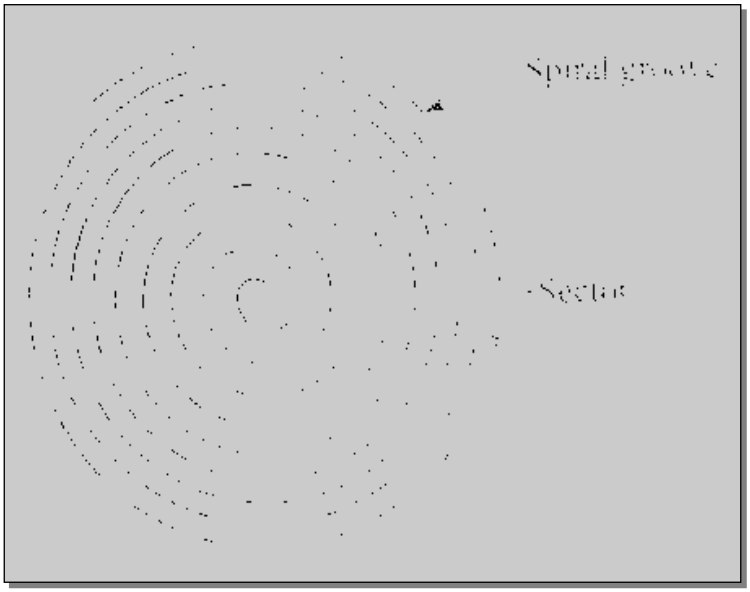


Figure 2: Tracks and sectors on a disk

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

7.3.1 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.

7.3.2 Latency Time

C: How far the head must travel.

$S = H * C + I$

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.

$L = 30 / R$

7.3.3 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).

7.4 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.

7.4.1 First come 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.

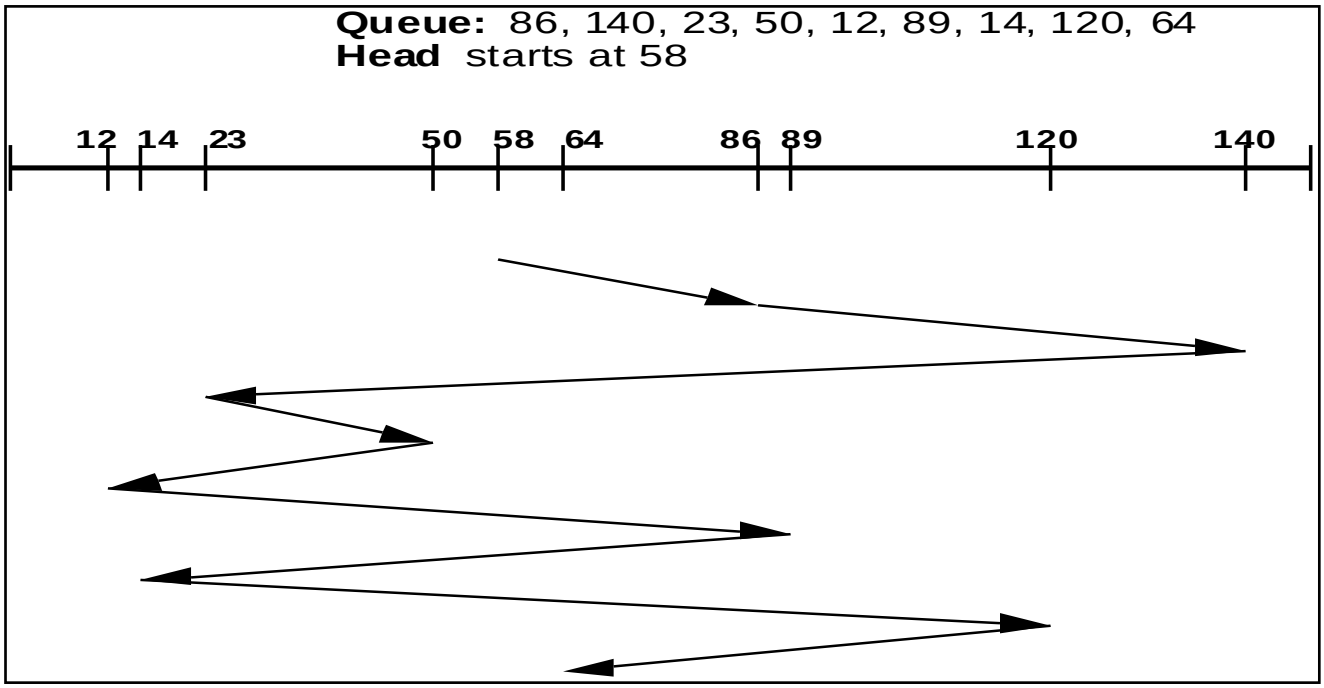


Figure 3:First Come First Serve Disk scheduling

7.4.2 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.

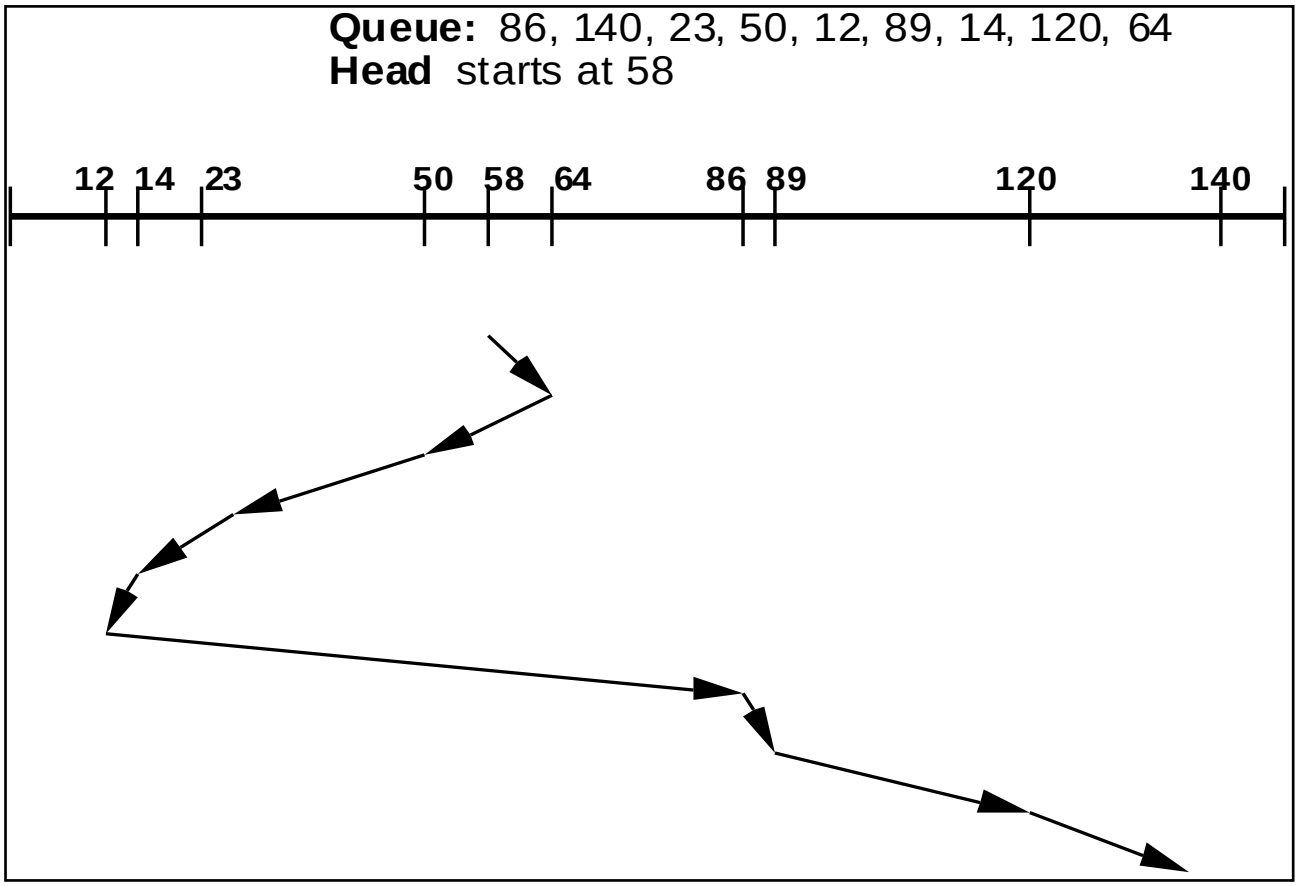


Figure 4: Shortest Seek Time First Scheduling

7.4.3 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.

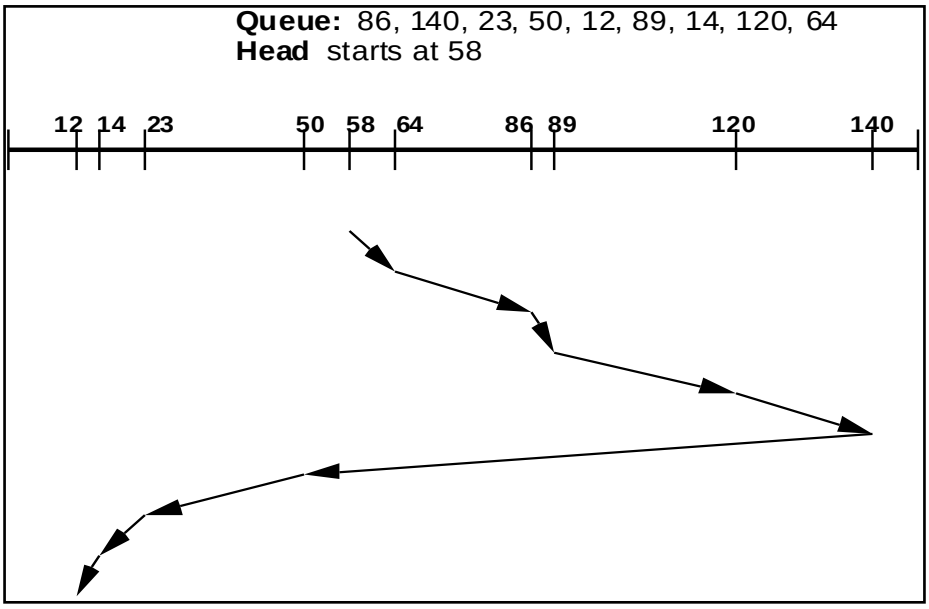


Figure 5: Scan

7.4.4 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.

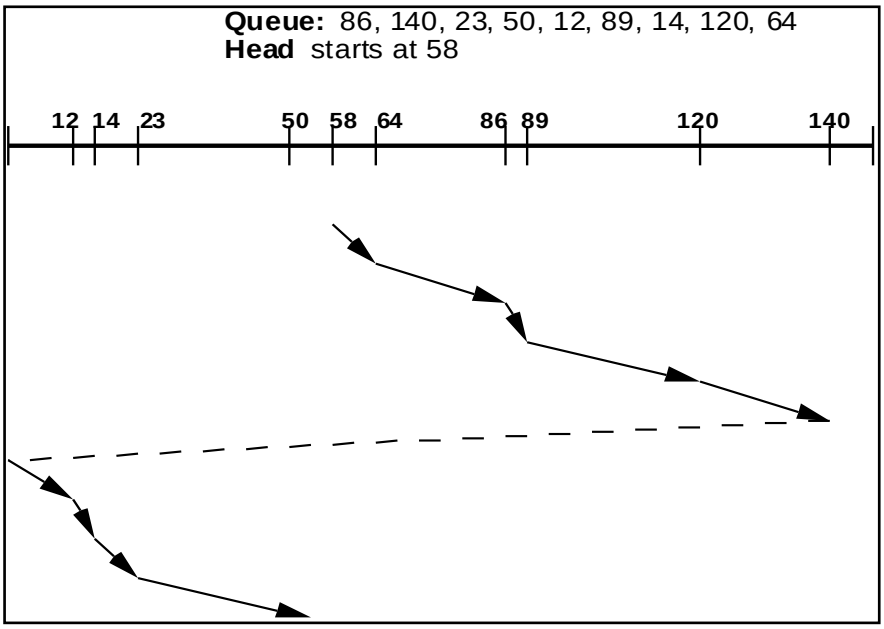


Figure 6: C-Scan

7.4.5 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.

7.4.6 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.

7.4.7 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.

7.4 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. queue are serviced, new requests are placed in other sub queue.

7.5 Check Your Progress

1. Disk scheduling includes deciding
   a. which should be accessed next
   b. order in which disk access requests must be serviced
   c. the physical location of the file
   d. the logical location of the file
2. Poor response times are caused by
   a. Busy processor
   b. High I/O rate
   c. High paging rates
   d. Any of above
3. On a movable head system, the time it takes to position the head at the track is known as ……
   A) seek time
   B) rotational delay
   C) access time
   D) Transfer time
4. . The time disk controller takes for the beginning of the sector to reach the head is known as ……..
   A)seek time
   B) rotational delay

C) access time

D) Transfer time

5. 14. The …….. consists of two key components: the initial startup time, and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed.

   A) seek time

   B) rotational delay

   C) access time

   D) Transfer time

6. The …… policy is to select the disk I/O request that requires the least movement of the disk arm from its current position.

   A) Last in first out

   B) Shortest service time first

   C) Priority by process

   D) Random scheduling

7. In ……. policy, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.

   A) Last in first out

   B) Shortest service time first

   C) SCAN

   D) Circular SCAN

8. Which of the following is/are the characteristics of RAID architecture.

   i) RAID is set of physical disk drives viewed by the operating system as a single logical drive

   ii) Data are distributed across the physical drives of an array

   iii) It is used to store parity information, which guarantees data recoverability in case of disk failure.

   A) i and ii only

   B) ii and iii only

   C) i and iiii only

   D) All i, ii and iii

9. …….. is not a true member of RAID family, because it does not include redundancy to improve performace.

A) RAID Level 0

B) RAID Level 1

C) RAID Level 2

D) RAID Level 3

10. …….. would only be an effective choice in a environment in which many disk errors occur.

A) RAID Level 0

B) RAID Level 1

C) RAID Level 2

D) RAID Level 3

11. 20. In the …… scheme, two different parity calculations are carried out an stored in separate blocks on different disks.

A) RAID Level 4

B) RAID Level 5

C) RAID Level 6

D) RAID Level 3

7.6 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.

7.7 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.

7.8 Self-Assessment Test

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?

7.9 Answers To Check Your Progress

1. Order in which disk access requests must be serviced
2. Any of above
3. Seek time
4. Rotational delay
5. Seek time
6. Shortest service time first
7. Circular scan
8. All i, ii and iii
9. Raid level 0
10. Raid level 2
11. Raid level 6

7.10 Reference/Suggested Readings

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.

NOTES

NOTES

NOTES

NOTES