OPERATING SYSTEM : AN OVERVIEW – [UNIT-I]

INTRODUCTION

Computer software can be divided into two main categories: application software and system software. Application software consists of the programs for performing tasks particular to the machine's utilisation. This software is designed to solve a particular problem for users. Examples of application software include spreadsheets, database systems, desktop publishing systems, program development software, and games.

On the other hand, system software is more transparent and less noticed by the typical computer user. This software provides a general programming environment in which programmers can create specific applications to suit their needs. This environment provides new functions that are not available at the hardware level and performs tasks related to executing the application program. System software acts as an interface between the hardware of the computer and the application software that users need to run on the computer. The most important type of system software is the operating system.

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

WHAT IS AN OPERATING SYSTEM?

In a computer system, we find four main components: the hardware, the operating system, the application software and the users. In a computer system the hardware provides the basic computing resources. The applications programs define the way in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various systems programs and application programs for the various users.

We can view an operating system as a **resource allocator**. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, files storage space, input/output devices etc. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks.

Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently.

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

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

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

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

An effective operating system should accomplish the following functions:

- Should act as a command interpreter by providing a user friendly environment.
- Should facilitate communication with other users.
- Facilitate the directory/file creation along with the security option.
- Provide routines that handle the intricate details of I/O programming.
- Provide access to compilers to translate programs from high-level languages to machine language
- Provide a loader program to move the compiled program code to the computer's memory for execution.
- Assure that when there are several active processes in the computer, each will get fair and non-interfering access to the central processing unit for execution.
- Take care of storage and device allocation.
- Provide for long term storage of user information in the form of files.

- Permit system resources to be shared among users when appropriate, and be protected from unauthorised or mischievous intervention as necessary.

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

The abstract view of the components of a computer system and the positioning of OS is shown in the Figure 1.

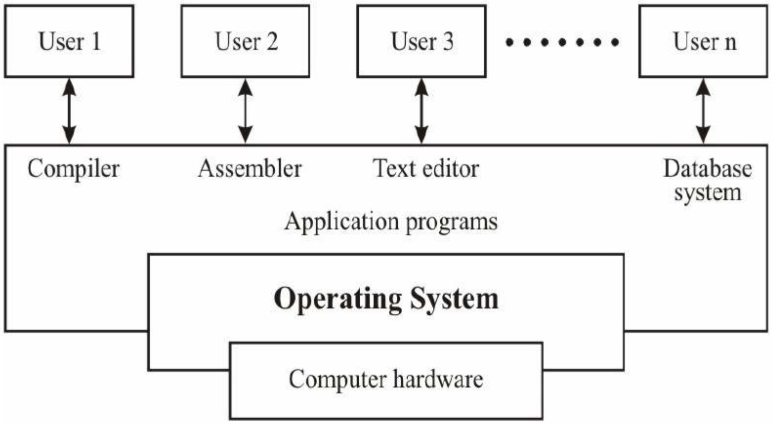


Figure 1: Abstract View of the Components of a Computer System

GOALS OF AN OPERATING SYSTEM

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

- The primary goal of an operating system is to make the computer **convenient** to use.

- The secondary goal is to use the hardware in an **efficient** manner.

TYPES OF OPERATING SYSTEMS

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

Batch Processing Operating System

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

Time Sharing

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

Real Time Operating System (RTOS)

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

These real time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.

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

Multiprogramming Operating System

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

Multiprocessing System

A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications. More on multiprocessor system can be studied in Unit-1 of Block-3 of this course.

Networking Operating System

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handing communication and transfer of program and data among the other computers with which it is connected.

Network operating systems are not fundamentally different from single processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote files access, but these additions do not change the essential structure of the operating systems.

Distributed Operating System

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

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

True distributed operating systems require more than just adding a little code to a uni-processor operating system, because distributed and centralised systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimise the amount of parallelism achieved. More on distributed systems can be studied in Unit-2 of Block-3 of this course.

Operating Systems for Embedded Devices

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

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

OPERATING SYSTEMS : SOME EXAMPLES

In the earlier section we had seen the types of operating systems. In this section we will study some popular operating systems.

DOS

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

UNIX

UNIX operating systems are used in widely-sold workstation products from Sun Microsystems, Silicon Graphics, IBM, and a number of other companies. The UNIX environment and the client/server program model were important elements in the development of the Internet and the reshaping of computing as centered in networks rather than in individual computers. Linux, a

UNIX derivative available in both “free software” and commercial versions, is increasing in popularity as an alternative to proprietary operating systems.

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

UNIX is made up of the kernel, file system and shell (command line interface). The major shells are the Bourne shell (original), C shell and Korn shell. The UNIX vocabulary is exhaustive with more than 600 commands that manipulate data and text in every way conceivable. Many commands are cryptic, but just as Windows hid the DOS prompt, the Motif GUI presents a friendlier image to UNIX users. Even with its many versions, UNIX is widely used in mission critical applications for client/server and transaction processing systems. The UNIX versions that are widely used are Sun’s Solaris, Digital’s UNIX, HP’s HP-UX, IBM’s AIX and SCO’s UnixWare. A large number of IBM mainframes also run UNIX applications, because the UNIX interfaces were added to MVS and OS/390, which have obtained UNIX branding. Linux, another variant of UNIX, is also gaining enormous popularity. More details can be studied in Unit-3 of Block-3 of this course.

WINDOWS

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

This networking support is the reason why Windows became successful in the first place. However, Windows 95, 98, ME, NT, 2000 and XP are complicated operating environments. Certain combinations of hardware and software running together can cause problems, and troubleshooting can be daunting. Each new version of Windows has interface changes that constantly confuse users and keep support people busy, and Installing Windows applications is problematic too. Microsoft has worked hard to make Windows 2000 and Windows XP more resilient to installation of problems and crashes in general. More details on Windows 2000 can be studied in Unit-4 of Block -3 of this course.

MACINTOSH

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

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

In this section we will discuss some services of the operating system used by its users. Users of operating system can be divided into two broad classes: command language users and system call users. Command language users are those who can interact with operating systems using the commands. On the other hand system call users invoke services of the operating system by means of run time system calls during the execution of programs.

FUNCTIONS OF OS

The main functions of an operating system are as follows:

- Process Management
- Memory Management
- Secondary Storage Management
- I/O Management
- File Management
- Protection
- Networking Management
- Command Interpretation.

Process Management

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

- The creation and deletion of both user and system processes
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

Memory Management

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

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

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

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and de-allocate memory space as needed.

Secondary Storage Management

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

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

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

- Free space management

- Storage allocation

- Disk scheduling.

I/O Management

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

- A buffer caching system

- To activate a general device driver code

- To run the driver software for specific hardware devices as and when required.

File Management

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

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

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

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

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

- The creation and deletion of files.

- The creation and deletion of directory.

- The support of primitives for manipulating files and directories.

- The mapping of files onto disk storage.

- Backup of files on stable (non volatile) storage.

- Protection and security of the files.

Protection

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

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

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

Networking

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

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

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

Command Interpretation

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

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

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

The Figure 2 depicts the role of the operating system in coordinating all the functions.

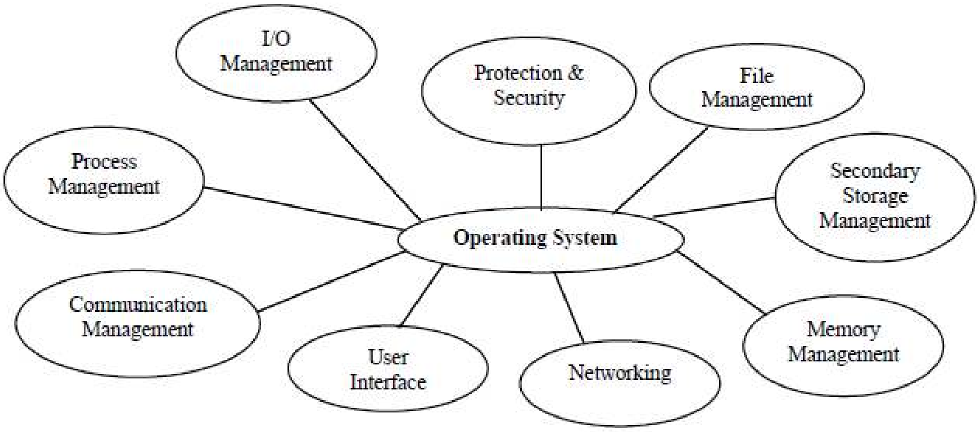


Figure 2: Functions Coordinated by the Operating System

PROCESS MANAGEMENT

INTRODUCTION

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

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when it is created, some initialisation data (input) may be passed along. For example, a process whose function is to display the status of a file, say F1, on the screen, will get the name of the file F1 as an input and execute the appropriate program to obtain the desired information.

We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program; they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes;
- The suspension is resumption of processes;
- The provision of mechanisms for process synchronization, and
- The provision of mechanisms for deadlock handling.

We will learn the operating system view of the processes, types of schedulers, different types of scheduling algorithms, in the subsequent sections of this unit.

THE CONCEPT OF PROCESS

The term “process” was first used by the operating system designers of the MULTICS system way back in 1960s. There are different definitions to explain the concept of process. Some of these are, a process is:

An instance of a program in execution

An asynchronous activity
The “animated spirit” of a procedure
The “locus of control” of a procedure in execution
The “dispatchable” unit
Unit of work individually schedulable by an operating system.

Formally, we can define a **process** is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the process is the activity.

In multiprogramming systems, processes are performed in a pseudo-parallelism as if each process has its own processor. In fact, there is only one processor but it switches back and forth from process to process. Henceforth, by saying execution of a process, we mean the processor’s operations on the process like changing its variables, etc. and I/O work means the interaction of the process with the I/O operations like reading something or writing to somewhere. They may also be named as “processor (CPU) burst” and “I/O burst” respectively. According to these definitions, we classify programs as:

- **Processor- bound program**: A program having long processor bursts (execution instants)

- **I/O- bound program**: A program having short processor bursts.

The operating system works as the computer system software that assists hardware in performing process management functions. Operating system keeps track of all the active processes and allocates system resources to them according to policies devised to meet design performance objectives. To meet process requirements OS must maintain many data structures efficiently. The process abstraction is fundamental means for the OS to manage concurrent program execution. OS must interleave the execution of a number of processes to maximize processor use while providing reasonable response time. It must allocate resources to processes in conformance with a specific policy. In general, a process will need certain resources such as CPU time, memory, files, I/O devices etc. to accomplish its tasks. These resources are allocated to the process when it is created. A single processor may be shared among several processes with some scheduling algorithm being used to determine when to stop work on one process and provide service to a different one which we will discuss later in this unit.

Operating systems must provide some way to create all the processes needed. In simple systems, it may be possible to have all the processes that will ever be needed be present when the system comes up. In almost all systems however, some way is needed to create and destroy processes as needed during operations. In UNIX, for instant, processes are created by the fork system call, which makes an identical copy of the calling process. In other systems, system calls exist to create a process, load its memory, and start it running. In general, processes need a way to create other processes. Each process has one parent process, but zero, one, two, or more children processes.

For an OS, the process management functions include:

- Process creation
- Termination of the process
- Controlling the progress of the process
- Process Scheduling
- Dispatching
- Interrupt handling / Exceptional handling
- Switching between the processes
- Process synchronization
- Interprocess communication support
- Management of Process Control Blocks.

Implicit and Explicit Tasking

A separate process at run time can be either–

- Implicit tasking and
- Explicit tasking.

Implicit tasking means that processes are defined by the system. It is commonly encountered in general purpose multiprogramming systems. In this approach, each program submitted for execution is treated by the operating system as an independent process. Processes created in this manner are usually transient in the sense that they are destroyed and disposed of by the system after each run.

Explicit tasking means that programs explicitly define each process and some of its attributes. To improve the performance, a single logical application is divided into various related processes. Explicit tasking is used in situations where high performance in desired system programs such as parts of the operating system and real time applications are common examples of programs defined processes. After dividing process into several independent processes, a system programs defines the confines of each individual process. A parent process is then commonly added to create the environment for and to control execution of individual processes.

Common reasons for and uses of explicit tasking include:

- **Speedup:** Explicit tasking can result in faster execution of applications.
- **Driving I/O devices that have latency:** While one task is waiting for I/O to complete, another portion of the application can make progress towards completion if it contains other tasks that can do useful work in the interim.

- **User convenience:** By creating tasks to handle individual actions, a graphical interface can allow users to launch several operations concurrently by clicking on action icons before completion of previous commands.
- **Multiprocessing and multicomputing:** A program coded as a collection of tasks can be relatively easily posted to a multiprocessor system, where individual tasks may be executed on different processors in parallel.

Distributed computing network server can handle multiple concurrent client sessions by dedicating an individual task to each active client session.

Processes Relationship

In the concurrent environment basically processes have two relationships, **competition** and **cooperation**. In the concurrent environment, processes compete with each other for allocation of system resources to execute their instructions. In addition, a collection of related processes that collectively represent a single logical application cooperate with each other. There should be a proper operating system to support these relations. In the competition, there should be proper resource allocation and protection in address generation.

We distinguish between independent process and cooperating process. A process is independent if it cannot affect or be affected by other processes executing in the system.

Independent process: These type of processes have following features:

- Their state is not shared in any way by any other process.
- Their execution is deterministic, i.e., the results of execution depend only on the input values.
- Their execution is reproducible, i.e., the results of execution will always be the same for the same input.
- Their execution can be stopped and restarted without any negative effect.

Cooperating process: In contrast to independent processes, cooperating processes can affect or be affected by other processes executing the system. They are characterised by:

- Their states are shared by other processes.
- Their execution is not deterministic, i.e., the results of execution depend on relative execution sequence and cannot be predicted in advance.

Their execution is irreproducible, i.e., the results of execution are not always the same for the same input.

Process States

As defined, a process is an independent entity with its own input values, output values, and internal state. A process often needs to interact with other processes. One process may generate some outputs that other process uses as input. For example, in the shell command

cat file1 file2 file3 | grep tree

The first process, running cat, concatenates and outputs three files. Depending on the relative speed of the two processes, it may happen that grep is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is ready and able to run to be blocked because the operating system is decided to allocate the CPU to other process for a while.

A process state may be in one of the following:

- **New :** The process is being created.
- **Ready :** The process is waiting to be assigned to a processor.
- **Running :** Instructions are being executed.
- **Waiting/Suspended/Blocked :** The process is waiting for some event to occur.
- **Halted/Terminated :** The process has finished execution.

The transition of the process states are shown in Figure 1.and their corresponding transition are described below:

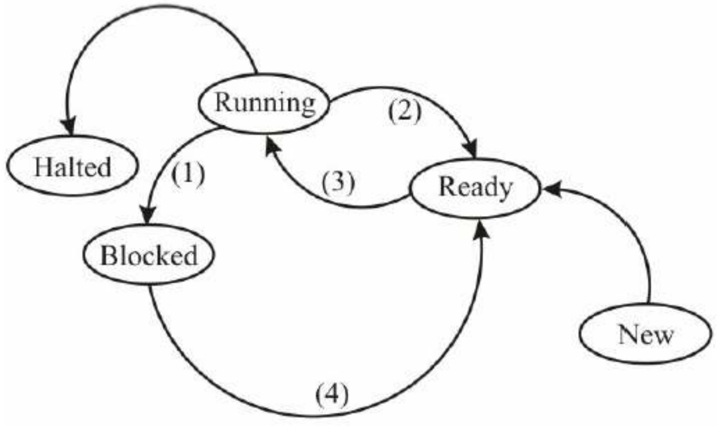


Figure 1: Typical process states

As shown in Figure 1, four transitions are possible among the states.

Transition 1 appears when a process discovers that it cannot continue. In order to get into blocked state, some systems must execute a system call block. In other systems, when a process reads from a pipe or special file and there is no input available, the process is automatically blocked.

Transition 2 and 3 are caused by the process scheduler, a part of the operating system. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all other processes have had their share and it is time for the first process to run again.

Transition 4 appears when the external event for which a process was waiting was happened. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes easier to think about what is going on inside the system. There are many processes like user processes, disk processes, terminal processes, and so on, which may be blocked when they are waiting for some thing to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is ready to run again.

The process model, an integral part of an operating system, can be summarized as follows. The lowest level of the operating system is the scheduler with a number of processes on top of it. All the process handling, such as starting and stopping processes are done by the scheduler. More on the schedulers can be studied is the subsequent sections.

Implementation of Processes

To implement the process model, the operating system maintains a table, an array of structures, called the process table or process control block (PCB) or Switch frame. Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information. In other words, it must contain everything about the process that must be saved when the process is switched from the running state to the ready state so that it can be restarted later as if it had never been stopped. The following is the information stored in a PCB.

- Process state, which may be new, ready, running, waiting or halted;
- Process number, each process is identified by its process number, called process ID;
- Program counter, which indicates the address of the next instruction to be executed for this process;
- CPU registers, which vary in number and type, depending on the concrete microprocessor architecture;
- Memory management information, which include base and bounds registers or page table;

- I/O status information, composed I/O requests, I/O devices allocated to this process, a list of open files and so on;
- Processor scheduling information, which includes process priority, pointers to scheduling queues and any other scheduling parameters;
- List of open files.

A process structure block is shown in Figure 2.

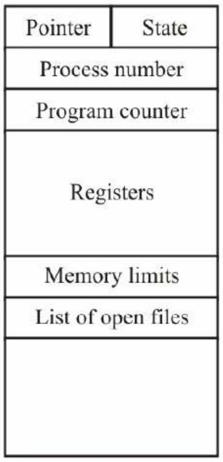


Figure 2: Process Control Block Structure

Context Switch

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

A context is the contents of a CPU's registers and program counter at any point in time.

A context switch is sometimes described as the kernel suspending execution of one process on the CPU and resuming execution of some other process that had previously been suspended.

Context Switch: Steps

In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue normally.

The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block (PCB). Now, in order to switch processes, the PCB for the first process must be created and saved. The PCBs are sometimes stored upon a per-process stack in the kernel memory, or there may be some specific operating system defined data structure for this information.

Let us understand with the help of an example. Suppose if two processes A and B are in ready queue. If CPU is executing Process A and Process B is in wait state. If an interrupt occurs for

Process A, the operating system suspends the execution of the first process, and stores the current information of Process A in its PCB and context to the second process namely Process B. In doing so, the program counter from the PCB of Process B is loaded, and thus execution can continue with the new process. The switching between two processes, Process A and Process B is illustrated in the Figure 3 given below:

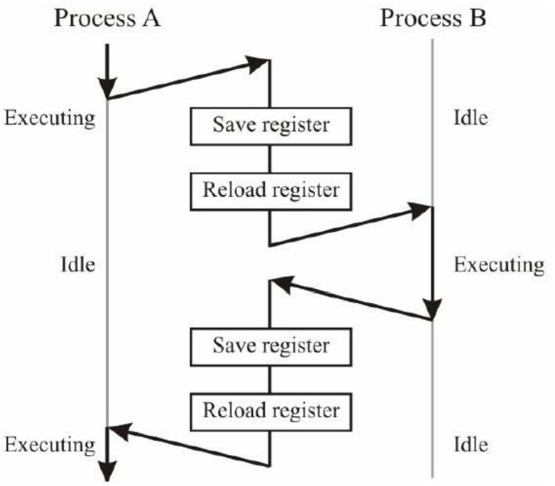


Figure 3: Process Switching between two processes

Process Hierarchy

Modern general purpose operating systems permit a user to create and destroy processes. A process may create several new processes during its time of execution. The creating process is called parent process, while the new processes are called child processes. There are different possibilities concerning creating new processes:

- **Execution**: The parent process continues to execute concurrently with its children processes or it waits until all of its children processes have terminated (sequential).

- **Sharing**: Either the parent and children processes share all resources (likes memory or files) or the children processes share only a subset of their parent's resources or the parent and children processes share no resources in common.

A parent process can terminate the execution of one of its children for one of these reasons:

- The child process has exceeded its usage of the resources it has been allocated. In order to do this, a mechanism must be available to allow the parent process to inspect the state of its children processes.

- The task assigned to the child process is no longer required.

Let us discuss this concept with an example. In UNIX this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process. After a fork both parent and child keep running (indeed they have the same program text) and each can fork off other processes. This results in a process tree. The root of the tree is a special process created by the OS during startup. A process can choose to wait for children to terminate. For example, if C issued a wait( ) system call it would block until G finished. This is shown in the Figure 4.

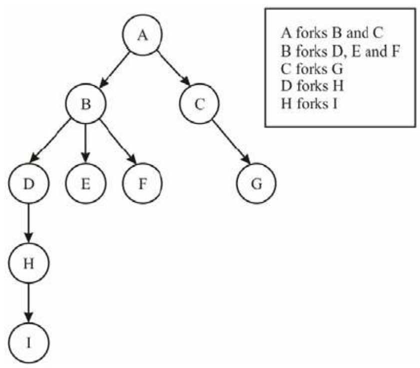


Figure 4: Process hierarchy

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is automatically blocked and waits until the second is finished.

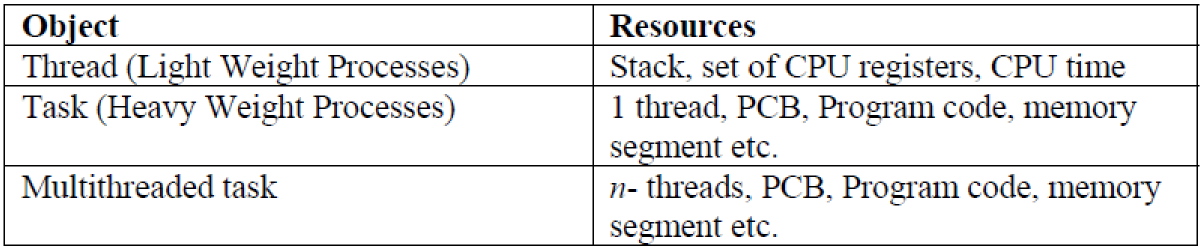
Threads

Threads, sometimes called lightweight processes (LWPs) are independently scheduled parts of a single program. We say that a task is multithreaded if it is composed of several independent sub-processes which do work on common data, and if each of those pieces could (at least in principle) run in parallel.

If we write a program which uses threads – there is only one program, one executable file, one task in the normal sense. Threads simply enable us to split up that program into logically separate pieces, and have the pieces run independently of one another, until they need to communicate. In a sense, threads are a further level of object orientation for multitasking systems. They allow certain functions to be executed in parallel with others.

On a truly parallel computer (several CPUs) we might imagine parts of a program (different subroutines) running on quite different processors, until they need to communicate. When one part of the program needs to send data to the other part, the two independent pieces must be synchronized, or be made to wait for one another. But what is the point of this? We can always run independent procedures in a program as separate programs, using the process mechanisms we have already introduced. They could communicate using normal interprocesses communication. Why introduce another new concept? Why do we need threads?

The point is that threads are cheaper than normal processes, and that they can be scheduled for execution in a user-dependent way, with less overhead. Threads are cheaper than a whole process because they do not have a full set of resources each. Whereas the process control block for a heavyweight process is large and costly to context switch, the PCBs for threads are much smaller, since each thread has only a stack and some registers to manage. It has no open file lists or resource lists, no accounting structures to update. All of these resources are shared by all threads within the process. Threads can be assigned priorities – a higher priority thread will get put to the front of the queue. Let's define heavy and lightweight processes with the help of a table.f



Object	Resources
Thread (Light Weight Processes)	Stack, set of CPU registers, CPU time
Task (Heavy Weight Processes)	1 thread, PCB, Program code, memory segment etc.
Multithreaded task	*n*- threads, PCB, Program code, memory segment etc.

Why use threads?

The sharing of resources can be made more effective if the scheduler knows known exactly what each program was going to do in advance. Of course, the scheduling algorithm can never know this – but the programmer who wrote the program does know. Using threads it is possible to organise the execution of a program in such a way that something is always being done, whenever the scheduler gives the heavyweight process CPU time.

- Threads allow a programmer to switch between lightweight processes when it is best for the program. (The programmer has control).

- A process which uses threads does not get more CPU time than an ordinary process – but the CPU time it gets is used to do work on the threads. It is possible to write a more efficient program by making use of threads.

- Inside a heavyweight process, threads are scheduled on a FCFS basis, unless the program decides to force certain threads to wait for other threads. If there is only one CPU, then only one thread can be running at a time.

- Threads context switch without any need to involve the kernel-the switching is performed by a user level library, so time is saved because the kernel doesn't need to know about the threads.

INTERPROCESS COMMUNICATION AND SYNCHRONIZATION – **[UNIT-II]**

INTRODUCTION

In the earlier unit we have studied the concept of processes. In addition to process scheduling, another important responsibility of the operating system is process synchronization. Synchronization involves the orderly sharing of system resources by processes.

Concurrency specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as a parallel process. For example, an airline reservation system that involves processing transactions from many terminals has a natural specification as a concurrent program in which each terminal is controlled by its own sequential process. Even when processes are not executed simultaneously, it is often easier to structure as a collection of cooperating sequential processes rather than as a single sequential program.

A simple batch operating system can be viewed as 3 processes-a reader process, an executor process and a printer process. The reader reads cards from card reader and places card images in an input buffer. The executor process reads card images from input buffer and performs the specified computation and store the result in an output buffer. The printer process retrieves the data from the output buffer and writes them to a printer. Concurrent processing is the basis of operating system which supports multiprogramming.

The operating system supports concurrent execution of a program without necessarily supporting elaborate form of memory and file management. This form of operation is also known as multitasking. One of the benefits of multitasking is that several processes can be made to cooperate in order to achieve their goals. To do this, they must do one of the following:

Communicate: Interprocess communication (IPC) involves sending information from one process to another. This can be achieved using a "mailbox" system, a socket which behaves like a virtual communication network (loopback), or through the use of "pipes". Pipes are a system construction which enable one process to open another process as if it were a file for writing or reading.

Share Data: A segment of memory must be available to both the processes. (Most memory is locked to a single process).

Waiting: Some processes wait for other processes to give a signal before continuing. This is an issue of synchronization.

In order to cooperate concurrently executing processes must communicate and synchronize. Interprocess communication is based on the use of shared variables (variables that can be referenced by more than one process) or message passing.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only works if the events perform an action or detect an action are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

In this unit, let us study the concept of interprocess communication and synchronization, need of semaphores, classical problems in concurrent processing, critical regions, monitors and message passing.

OBJECTIVES

After studying this unit, you should be able to:

- identify the significance of interprocess communication and synchronization;
- describe the two ways of interprocess communication namely shared memory and message passing;
- discuss the usage of semaphores, locks and monitors in interprocess and synchronization, and
- solve classical problems in concurrent programming.

INTERPROCESS COMMUNICATION

Interprocess communication (IPC) is a capability supported by operating system that allows one process to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprogramming systems, but it is not generally supported by single-process operating systems such as DOS. OS/2 and MS-Windows support an IPC mechanism called Dynamic Data Exchange.

IPC allows the process to communicate and to synchronize their actions without sharing the same address space. This concept can be illustrated with the example of a shared printer as given below:

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

Consider the following related definitions to understand the example in a better way:

Critical Resource: It is a resource shared with constraints on its use (e.g., memory, files, printers, etc).

Critical Section: It is code that accesses a critical resource.

Mutual Exclusion: At most one process may be executing a critical section with respect to a particular critical resource simultaneously.

In the example given above, the printer is the critical resource. Let's suppose that the processes which are sharing this resource are called process A and process B. The critical sections of process A and process B are the sections of the code which issue the print command. In order to ensure that both processes do not attempt to use the printer at the same, they must be granted mutually exclusive access to the printer driver.

First we consider the interprocess communication part. There exist two complementary inter-process communication types: a) shared-memory system and b) message-passing system. It is clear that these two schemes are not mutually exclusive, and could be used simultaneously within a single operating system.

Shared-Memory System

Shared-memory systems require communication processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory scheme, the responsibility for providing communication rests with the application programmers. The operating system only needs to provide shared memory.

A critical problem occurring in shared-memory system is that two or more processes are reading or writing some shared variables or shared data, and the final results depend on who runs precisely and when. Such situations are called race conditions. In order to avoid race conditions we must find some way to prevent more than one process from reading and writing shared variables or shared data at the same time, i.e., we need the concept of mutual exclusion (which we will discuss in the later section). It must be sure that if one process is using a shared variable, the other process will be excluded from doing the same thing.

Message-Passing System

Message passing systems allow communication processes to exchange messages. In this scheme,

the responsibility rests with the operating system itself.

The function of a message-passing system is to allow processes to communicate with each other without the need to resort to shared variable. An interprocess communication facility basically provides two operations: send (message) and receive (message). In order to send and to receive messages, a communication link must exist between two involved processes. This link can be implemented in different ways. The possible basic implementation questions are:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of process?
- What is the capacity of a link? That is, does the link have some buffer space? If so, how much?
- What is the size of the message? Can the link accommodate variable size or fixed-size message?
- Is the link unidirectional or bi-directional?

In the following we consider several methods for logically implementing a communication link and the send/receive operations. These methods can be classified into two categories: a) Naming, consisting of direct and indirect communication and b) Buffering, consisting of capacity and messages proprieties.

INTERPROCESS SYNCHRONIZATION

When two or more processes work on the same data simultaneously strange things can happen. Suppose, when two parallel threads attempt to update the same variable simultaneously, the result is unpredictable. The value of the variable afterwards depends on which of the two threads was the last one to change the value. This is called a race condition. The value depends on which of the threads wins the race to update the variable. What we need in a mulitasking system is a way of making such situations predictable. This is called serialization. Let us study the serialization concept in detail in the next section.

Serialization

The key idea in process synchronization is serialization. This means that we have to go to some pains to undo the work we have put into making an operating system perform several tasks in parallel. As we mentioned, in the case of print queues, parallelism is not always appropriate.

Synchronization is a large and difficult topic, so we shall only undertake to describe the problem and some of the principles involved here.

There are essentially two strategies to serializing processes in a multitasking environment.

- The scheduler can be disabled for a short period of time, to prevent control being given to another process during a critical action like modifying shared data. This method is very inefficient on multiprocessor machines, since all other processors have to be halted every time one wishes to execute a critical section.

- A protocol can be introduced which all programs sharing data must obey. The protocol ensures that processes have to queue up to gain access to shared data. Processes which ignore the protocol ignore it at their own peril (and the peril of the remainder of the system!). This method works on multiprocessor machines also, though it is more difficult to visualize. The responsibility of serializing important operations falls on programmers. The OS cannot impose any restrictions on silly behaviour-it can only provide tools and mechanisms to assist the solution of the problem.

Mutexes: Mutual Exclusion

When two or more processes must share some object, an arbitration mechanism is needed so that they do not try to use it at the same time. The particular object being shared does not have a great impact on the choice of such mechanisms. Consider the following examples: two processes sharing a printer must take turns using it; if they attempt to use it simultaneously, the output from the two processes may be mixed into an arbitrary jumble which is unlikely to be of any use. Two processes attempting to update the same bank account must take turns; if each process reads the current balance from some database, updates it, and then writes it back, one of the updates will be lost.

Both of the above examples can be solved if there is some way for each process to exclude the other from using the shared object during critical sections of code. Thus the general problem is described as the mutual exclusion problem. The mutual exclusion problem was recognised (and successfully solved) as early as 1963 in the Burroughs AOSP operating system, but the problem is sufficiently difficult widely understood for some time after that. A significant number of attempts to solve the mutual exclusion problem have suffered from two specific problems, the lockout problem, in which a subset of the processes can conspire to indefinitely lock some other process out of a critical section, and the deadlock problem, where two or more processes simultaneously trying to enter a critical section lock each other out.

On a uni-processor system with non-preemptive scheduling, mutual exclusion is easily obtained: the process which needs exclusive use of a resource simply refuses to relinquish the processor until it is done with the resource. A similar solution works on a preemptively scheduled uni-processor: the process which needs exclusive use of a resource disables interrupts to prevent preemption until the resource is no longer needed. These solutions are appropriate and have been widely used for short critical sections, such as those involving updating a shared variable in main memory. On the other hand, these solutions are not appropriate for long critical sections, for example, those which involve input/output. As a result, users are normally forbidden to use these solutions; when they are used, their use is restricted to system code.

Mutual exclusion can be achieved by a system of locks. A mutual exclusion lock is colloquially called a mutex. You can see an example of mutex locking in the multithreaded file reader in the previous section. The idea is for each thread or process to try to obtain locked-access to shared data:

```
Get_Mutex(m);

// Update shared data

Release_Mutex(m);
```

This protocol is meant to ensure that only one process at a time can get past the function Get_Mutex. All other processes or threads are made to wait at the function Get_Mutex until that one process calls Release_Mutex to release the lock. A method for implementing this is discussed below. Mutexes are a central part of multithreaded programming.

Critical Sections: The Mutex Solution

A critical section is a part of a program in which is it necessary to have exclusive access to shared data. Only one process or a thread may be in a critical section at any one time. The characteristic properties of the code that form a Critical Section are:

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

In the past it was possible to implement this by generalising the idea of interrupt masks. By switching off interrupts (or more appropriately, by switching off the scheduler) a process can guarantee itself uninterrupted access to shared data. This method has drawbacks:

i) Masking interrupts can be dangerous- there is always the possibility that important interrupts will be missed;

ii) It is not general enough in a multiprocessor environment, since interrupts will continue to be serviced by other processors-so all processors would have to be switched off;

iii) It is too harsh. We only need to prevent two programs from being in their critical sections simultaneously if they share the same data. Programs A and B might share different data to programs C and D, so why should they wait for C and D?

In 1981 G.L. Peterson discovered a simple algorithm for achieving mutual exclusion between two processes with PID equal to 0 or 1. The code is as follows:

```
int turn;
int interested[2];
void Get_Mutex (int pid)
{
int other;
other = 1 - pid;
interested[process] = true;
turn = pid;
while (turn == pid && interested[other]) // Loop until no one
{ // else is interested
}
}
Release_Mutex (int pid)
{
interested[pid] = false;
}
```

Where more processes are involved, some modifications are necessary to this algorithm. The key to serialization here is that, if a second process tries to obtain the mutex, when another already has it, it will get caught in a loop, which does not terminate until the other process has released the mutex. This solution is said to involve busy waiting-i.e., the program actively executes an empty loop, wasting CPU cycles, rather than moving the process out of the scheduling queue. This is also called a spin lock, since the system 'spins' on the loop while waiting. Let us see another algorithm which handles critical section problem for n processes.

3.3.4 Dekker's solution for Mutual Exclusion

Mutual exclusion can be assured even when there is no underlying mechanism such as the test-and-set instruction. This was first realised by T. J. Dekker and published (by Dijkstra) in 1965. Dekker's algorithm uses busy waiting and works for only two processes. The basic idea is that processes record their interest in entering a critical section (in Boolean variables called "need") and they take turns (using a variable called "turn") when both need entry at the same time. Dekker's solution is shown below:

```
type processid = 0..1;

 var need: array [ processid ] of boolean { initially false };
```

```
turn: processid { initially either 0 or 1 };

procedure dekkerwait( me: processid );
var other: processid;
begin
    other := 1 - me;
    need[ me ] := true;
    while need[ other ] do begin { there is contention }
        if turn = other then begin
            need[ me ] := false { let other take a turn };
            while turn = other do { nothing };
            need[ me ] := true { re-assert my interest };
        end;
    end;
end { dekkerwait };

procedure dekkersignal( me: processid );
begin
    need[ me ] := false;
    turn := 1 - me { now, it is the other's turn };
end { dekkersignal };
```

Dekkers solution to the mutual exclusion problem requires that each of the contending processes have a unique process identifier which is called “me” which is passed to the wait and signal operations. Although none of the previously mentioned solutions require this, most systems provide some form of process identifier which can be used for this purpose.

It should be noted that Dekker's solution does rely on one very simple assumption about the underlying hardware; it assumes that if two processes attempt to write two different values in the same memory location at the same time, one or the other value will be stored and not some mixture of the two. This is called the atomic update assumption. The atomically updatable unit of memory varies considerably from one system to another; on some machines, any update of a word in memory is atomic, but an attempt to update a byte is not atomic, while on others, updating a byte is atomic while words are updated by a sequence of byte updates.

Bakery's Algorithm

Bakery algorithm handles critical section problem for n processes as follows:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if i < j , then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #)

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

  o max(a_0, . . . , a_{n-1}) is a number, k , such that k >= a_i for i = 0, . . . , n - 1
- Shared data

  boolean choosing[n]; //initialise all to false **int** number[n]; //initialise all to 0
- Data structures are initialized to false and 0, respectively.

The algorithm is as follows:

```
do { choosing[i] = true; number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
choosing[i] = false; for(int j = 0; j < n; j++) { while (choosing[j]== true) { /*do nothing*/ }
while ((number[j]!=0) && (number[j],j)< (number[i],i)) // see Reference point { /*do
nothing*/ } } do critical section number[i] = 0; do remainder section }while (true)
```

In the next section we will study how the semaphores provides a much more organise approach of synchronization of processes.

SEMAPHORES

Semaphores provide a much more organised approach to controlling the interaction of multiple processes than would be available if each user had to solve all interprocess communications using simple variables, but more organization is possible. In a sense, semaphores are something like the **goto** statement in early programming languages; they can be used to solve a variety of problems, but they impose little structure on the solution and the results can be hard to understand without the aid of numerous comments. Just as there have been numerous control structures devised in sequential programs to reduce or even eliminate the need for **goto** statements, numerous specialized concurrent control structures have been developed which reduce or eliminate the need for semaphores.

Definition: The effective synchronization tools often used to realise mutual exclusion in more complex systems are semaphores. A semaphore **S** is an integer variable which can be accessed only through two standard atomic operations: **wait and signal**. The definition of the wait and signal operation are:

```
wait(S): while S ≤ 0 do skip;
        S := S – 1;
signal(S): S := S + 1;
```

or in C language notation we can write it as:

```
        wait(s) { while (S<=0)
        { /*do nothing*/ } S= S-1; } signal(S) { S = S + 1; }
```

It should be noted that the test ($S \leq 0$) and modification of the integer value of S which is S := S – 1 must be executed without interruption. In general, if one process modifies the integer value of S in the wait and signal operations, no other process can simultaneously modify that same S value. We briefly explain the usage of semaphores in the following example:

Consider two currently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed. This scheme can be implemented by letting P_1 and P_2 share a common semaphore synch, initialised to 0, and by inserting the statements:

S_1;

signal(synch);

in the process P_1 and the statements:

wait(synch);

S_2;

in the process P_2.

Since synch is initialised to 0, P_2 will execute S_2 only after P_1 has involved signal (synch), which is after S_1.

The disadvantage of the semaphore definition given above is that it requires busy-waiting, i.e., while a process is in its critical region, any either process it trying to enter its critical region must continuously loop in the entry code. It's clear that through busy-waiting, CPU cycles are wasted by which some other processes might use those productively.

To overcome busy-waiting, we modify the definition of the wait and signal operations. When a process executes the wait operation and finds that the semaphore value is not positive, the process blocks itself. The block operation places the process into a waiting state. Using a scheduler the CPU then can be allocated to other processes which are ready to run.

A process that is blocked, i.e., waiting on a semaphore S, should be restarted by the execution of a signal operation by some other processes, which changes its state from blocked to ready. To implement a semaphore under this condition, we define a semaphore as:

struct semaphore { int value; List *L; //a list of processes }

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to this list. A signal operation removes one process from the list of waiting processes, and awakens it. The semaphore operation can be now defined as follows:

```
wait(S) { S.value = S.value -1; if (S.value <0)
{ add this process to S.L; block; } } signal(S) { S.value = S.value + 1; if (S.value <= 0) {
remove a process P from S.L; wakeup(P); } }
```

The block operation suspends the process. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

One of the almost critical problem concerning implementing semaphore is the situation where two or more processes are waiting indefinitely for an event that can be only caused by one of the waiting processes: these processes are said to be deadlocked. To illustrate this, consider a system consisting of two processes P_1 and P_2, each accessing two semaphores S and Q, set to the value one:

P_1 P_2

wait(S); wait(Q);

wait(Q); wait(S);

... ...

signal(S); signal(Q);

signal(Q); signal(S);

Suppose P_1 executes wait(S) and then P_2 executes wait(Q). When P_1 executes wait(Q), it must wait until P_2 executes signal(Q). It is no problem, P_2 executes wait(Q), then signal(Q). Similarly, when P_2 executes wait(S), it must wait until P_1 executes signal(S). Thus these signal operations cannot be carried out, P_1 and P_2 are deadlocked. It is clear, that a set of processes are in a deadlocked state, when every process in the set is waiting for an event that can only be caused by another process in the set.

CLASSICAL PROBLEMS IN CONCURRENT PROGRAMMING

In this section, we present a large class of concurrency control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

Producers/Consumers Problem

Producer – Consumer processes are common in operating systems. The problem definition is that, a producer (process) produces the information that is consumed by a consumer (process). For example, a compiler may produce assembly code, which is consumed by an assembler. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized. These problems can be solved either through unbounded buffer or bounded buffer.

- ***With an unbounded buffer***

  The unbounded-buffer producer- consumer problem places no practical limit on the size of the buffer .The consumer may have to wait for new items, but the producer can always produce new items; there are always empty positions in the buffer.

- **With a bounded buffer**

  The bounded buffer producer problem assumes that there is a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Shared Data

```
    char item; //could be any data type
```

```
char buffer[n]; semaphore full = 0; //counting semaphore semaphore empty = n; //counting semaphore semaphore mutex = 1; //binary semaphore char nextp,nextc;
```

Producer Process

```
do { produce an item in nextp wait (empty); wait (mutex); add nextp to buffer signal (mutex); signal (full); } while (true)
```

Consumer Process

```
do { wait( full ); wait( mutex ); remove an item from buffer to nextc signal( mutex ); signal( empty ); consume the item in nextc; }
```

Readers and Writers Problem

The readers/writers problem is one of the classic synchronization problems. It is often used to compare and contrast synchronization mechanisms. It is also an eminently used practical problem. A common paradigm in concurrent applications is isolation of shared data such as a variable, buffer, or document and the control of access to that data. This problem has two types of clients accessing the shared data. The first type, referred to as readers, only wants to read the shared data. The second type, referred to as writers, may want to modify the shared data. There is also a designated central data server or controller. It enforces exclusive write semantics; if a writer is active then no other writer or reader can be active. The server can support clients that wish to both read and write. The readers and writers problem is useful for modeling processes which are competing for a limited shared resource. Let us understand it with the help of a practical example:

An airline reservation system consists of a huge database with many processes that read and write the data. Reading information from the database will not cause a problem since no data is changed. The problem lies in writing information to the database. If no constraints are put on access to the database, data may change at any moment. By the time a reading process displays the result of a request for information to the user, the actual data in the database may have changed. What if, for instance, a process reads the number of available seats on a flight, finds a value of one, and reports it to the customer? Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the number of available seats to zero.

The following is the solution using semaphores:

Semaphores can be used to restrict access to the database under certain conditions. In this example, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

semaphore mutex = 1; // Controls access to the reader count

semaphore db = 1; // Controls access to the database

```
int reader_count; // The number of reading processes accessing the data
Reader()
{
while (TRUE) { // loop forever
down(&mutex); // gain access to reader_count
reader_count = reader_count + 1; // increment the reader_count
if (reader_count == 1)
down(&db); //If this is the first process to read the database,
// a down on db is executed to prevent access to the
// database by a writing process
up(&mutex); // allow other processes to access reader_count
read_db(); // read the database
down(&mutex); // gain access to reader_count
reader_count = reader_count - 1; // decrement reader_count
if (reader_count == 0)
up(&db); // if there are no more processes reading from the
// database, allow writing process to access the data
up(&mutex); // allow other processes to access reader_countuse_data();
// use the data read from the database (non-critical)
}
Writer()
{
while (TRUE) { // loop forever
create_data(); // create data to enter into database (non-critical)
down(&db); // gain access to the database
write_db(); // write information to the database

up(&db); // release exclusive access to the database

}
```

Dining Philosophers Problem

Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the centre of the table is a large bowl of rice. A philosopher needs two chopsticks to eat. Only 5 chop sticks are available and a chopstick is placed between each pair of philosophers. They agree that each will only use the chopstick to his immediate right and left. From time to time, a philosopher gets hungry and tries to grab the two chopsticks that are immediate left and right to him. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he finishes eating, he puts down both his chopsticks and starts thinking again.

Here's a solution for the problem which does not require a process to write another process's state, and gets equivalent parallelism.

```
#define N 5 /* Number of philosphers */

#define RIGHT(i) (((i)+1) %N)

#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;

phil_state state[N];

semaphore mutex =1;

semaphore s[N];

/* one per philosopher, all 0 */

void get_forks(int i) {

state[i] = HUNGRY;

while ( state[i] == HUNGRY ) {

        P(mutex);

        if ( state[i] == HUNGRY &&

        state[LEFT] != EATING &&
          state[RIGHT(i)] != EATING ) {
                    state[i] = EATING;
          V(s[i]);
}
V(mutex);
P(s[i]);
}
}
void put_forks(int i) {
P(mutex);
state[i]= THINKING;
if ( state[LEFT(i)] == HUNGRY ) V(s[LEFT(i)]);
if ( state[RIGHT(i)] == HUNGRY) V(s[RIGHT(i)]);
V(mutex);
}
void philosopher(int process) {
while(1) {
think();
get_forks(process);
eat();
put_forks();
}
}
```

Sleeping Barber Problem

A barber shop consists of a waiting room with chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barber shop and all chairs are occupied, then the customer leaves the shop. if the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

The following is a sample solution for the sleeping barber problem.

```
# define CHAIRS 5 // chairs for waiting customers
typedef int semaphore; // use this for imagination
semaphore customers = 0; // number of customers waiting for service
semaphore barbers – 0; // number of barbers waiting for customers
semaphore mutex = 1; // for mutual exclusion
int waiting = 0; //customers who are waiting for a haircut
void barber(void)
{
while (TRUE) {
down(&customers); //go to sleep if no of customers are zero
down(&mutex); //acquire access to waiting
waiting = waiting -1 ; //decrement count of waiting customers
up(&barbers); //one barber is now ready for cut hair
up(&mutex); //release waiting
cut_hair(); //this is out of critical region for hair cut
}
}
void customer(void)
{
down(&mutex); //enter critical region
if (waiting < CHAIRS) //if there are no free chairs, leave
            {
waiting = waiting +1; //increment count of waiting customers
up(&customers); //wake up barber if necessary
up(&mutex); //release access to waiting
down(&barbers); //go to sleep if no of free barbers is zero
get_haircut(); //be seated and be serviced
} else
{
up (&mutex); // shop is full: do no wait
}
}
```

Explanation:

- This problem is similar to various queuing situations
- The problem is to program the barber and the customers without getting into race conditions
    - Solution uses three semaphores:
        - customers; counts the waiting customers
        - barbers; the number of barbers (0 or 1)
        - mutex; used for mutual exclusion
        - also need a variable waiting; also counts the waiting customers; (reason; no way to read the current value of semaphore)
    - The barber executes the procedure barber, causing him to block on the semaphore customers (initially 0);
    - The barber then goes to sleep;
    - When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region;

- If another customer enters, shortly thereafter, the second one will not be able to do anything until the first one has released mutex;
- The customer then checks to see if the number of waiting customers is less than the number of chairs;
- If not, he releases mutex and leaves without a haircut;
- If there is an available chair, the customer increments the integer variable, waiting;
- Then he does an **up** on the semaphore customers;
- When the customer releases mutex, the barber begins the haircut.

LOCKS

Locks are another synchronization mechanism. A lock has got two atomic operations (similar to semaphore) to provide mutual exclusion. These two operations are Acquire and Release. A process will acquire a lock before accessing a shared variable, and later it will be released. A process locking a variable will run the following code:

Lock-Acquire(); critical section Lock-Release();
The difference between a lock and a semaphore is that a lock is released only by the process that have acquired it earlier. As we discussed above any process can increment the value of the semaphore. To implement locks, here are some things you should keep in mind:

- To make Acquire () and Release () atomic
- Build a wait mechanism.
- Making sure that only the process that acquires the lock will release the lock.

MONITORS AND CONDITION VARIABLES

When you are using semaphores and locks you must be very careful, because a simple misspelling may lead that the system ends up in a deadlock. Monitors are written to make synchronization easier and correctly. Monitors are some procedures, variables, and data structures grouped together in a package.

An early proposal for organising the operations required to establish mutual exclusion is the explicit critical section statement. In such a statement, usually proposed in the form “critical x do y”, where “x” is the name of a semaphore and “y” is a statement, the actual wait and signal operations used to ensure mutual exclusion were implicit and automatically balanced. This allowed the compiler to trivially check for the most obvious errors in concurrent programming, those where a wait or signal operation was accidentally forgotten. The problem with this statement is that it is not adequate for many critical sections.

A common observation about critical sections is that many of the procedures for manipulating shared abstract data types such as files have critical sections making up their entire bodies. Such abstract data types have come to be known as monitors, a term coined by C. A. R. Hoare. Hoare proposed a programming notation where the critical sections and semaphores implicit in the use of a monitor were all implicit. All that this notation requires is that the programmer encloses the declarations of the procedures and the representation of the data type in a monitor block; the compiler supplies the semaphores and the wait and signal operations that this implies. Using Hoare's suggested notation, shared counters might be implemented as shown below:

```
var value: integer;

procedure increment;

begin

value := value + 1;

end { increment };

end { counter };

var i, j: counter;
```

Calls to procedures within the body of a monitor are done using record notation; thus, to increment one of the counters declared in above example, one would call “i.increment”. This call would implicitly do a wait operation on the semaphore implicitly associated with “i”, then execute the body of the “increment” procedure before doing a signal operation on the semaphore. Note that the call to “i.increment” implicitly passes a specific instance of the monitor as a parameter to the “increment” procedure, and that fields of this instance become global variables to the body of the procedure, as if there was an implicit “with” statement.

There are a number of problems with monitors which have been ignored in the above example. For example, consider the problem of assigning a meaning to a call from within one monitor procedure to a procedure within another monitor. This can easily lead to a deadlock. For example, when procedures within two different monitors each calling the other. It has sometimes been proposed that such calls should never be allowed, but they are sometimes useful! We will study more on deadlocks in the next units of this course.

The most important problem with monitors is that of waiting for resources when they are not available. For example, consider implementing a queue monitor with internal procedures for the enqueue and dequeue operations. When the queue empties, a call to dequeue must wait, but this wait must not block further entries to the monitor through the enqueue procedure. In effect, there must be a way for a process to temporarily step outside of the monitor, releasing mutual exclusion while it waits for some other process to enter the monitor and do some needed action.

Hoare’s suggested solution to this problem involves the introduction of condition variables which may be local to a monitor, along with the operations wait and signal. Essentially, if s is the monitor semaphore, and c is a semaphore representing a condition variable, “wait c” is equivalent to “signal(s); wait(c); wait(s)” and “signal c” is equivalent to “signal(c)”. The details of Hoare’s wait and signal operations were somewhat more complex than is shown here because the waiting process was given priority over other processes trying to enter the monitor, and condition variables had no memory; repeated signalling of a condition had no effect and signaling a condition on which no process was waiting had no effect. Following is an example monitor:

monitor synch integer i; condition c;

procedure producer(x); . .

end;

procedure consumer(x); . . end; end monitor;

There is only one process that can enter a monitor, therefore every monitor has its own waiting list with process waiting to enter the monitor.

Let us see the dining philosopher’s which was explained in the above section with semaphores, can be re-written using the monitors as:

Example: Solution to the Dining Philosophers Problem using Monitors

monitor dining-philosophers { enum **state** {thinking, hungry, eating}; **state** state[5]; **condition** self[5]; void pickup (int i) { state[i] = hungry; test(i); if (state[i] != eating) self[i].wait; } void putdown (int i) { state[i] = thinking; test(i+4 % 5); test(i+1 % 5); } void test (int k) { if ((state[k+4 % 5] != eating) && (state[k]==hungry) && state[k+1 % 5] != eating)) { state[k] = eating; self[k].signal; } } init { for (int i = 0; i< 5; i++) state[i] = thinking; } }

Condition Variables

If a process cannot enter the monitor it must block itself. This operation is provided by the condition variables. Like locks and semaphores, the condition has got a wait and a signal function. But it also has the broadcast signal. Implementation of condition variables is part of a synch.h; it is your job to implement it. Wait (), Signal () and Broadcast () have the following semantics:

- Wait() releases the lock, gives up the CPU until signaled and then re-acquire the lock.

- Signal() wakes up a thread if there are any waiting on the condition variable.

- Broadcast() wakes up all threads waiting on the condition.

When you implement the condition variable, you must have a queue for the processes waiting on the condition variable.

DEADLOCKS

INTRODUCTION

In a computer system, we have a finite number of resources to be distributed among a number of competing processes. These system resources are classified in several types which 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. Each resource type can have some identical instances.

A process must request a resource before using it and release the resource after using it. It is clear that the number of resources requested cannot exceed the total number of resources available in the system.

In a normal operation, a process may utilise a resource only in the following sequence:

- Request: if the request cannot be immediately granted, then the requesting process must wait until it can get the resource.
- Use: the requesting process can operate on the resource.
- Release: the process releases the resource after using it.

Examples for request and release of system resources are:

- Request and release the device,
- Opening and closing file,
- Allocating and freeing the memory.

The operating system is responsible for making sure that the requesting process has been allocated the resource. A system table indicates if each resource is free or allocated, and if allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

In some cases, several processes may compete for a fixed number of resources. A process requests resources and if the resources are not available at that time, it enters a wait state. It may happen that it will never gain access to the resources, since those resources are being held by other waiting processes.

For example, assume a system with one tape drive and one plotter. Process P1 requests the tape drive and process P2 requests the plotter. Both requests are granted. Now PI requests the plotter (without giving up the tape drive) and P2 requests the tape drive (without giving up the plotter). Neither request can be granted so both processes enter a situation called the **deadlock** situation.

A **deadlock** is a situation where a group of processes is permanently blocked as a result of each process having acquired a set of resources needed for its completion and having to wait for the release of the remaining resources held by others thus making it impossible for any of the deadlocked processes to proceed.

In the earlier units, we have gone through the concept of process and the need for the interprocess communication and synchronization. In this unit we will study about the deadlocks, its characterisation, deadlock avoidance and its recovery.

DEADLOCK

Before studying about deadlocks, let us look at the various types of resources. There are two types of resources namely: Pre-emptable and Non-pre-emptable Resources.

- **Pre-emptable resources**: This resource can be taken away from the process with no ill effects. Memory is an example of a pre-emptable resource.
- **Non-Preemptable resource**: This resource cannot be taken away from the 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. Let us see how a deadlock occurs.

Definition: A set of processes 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.

Let us understand the deadlock situation with the help of examples.

Example 1: The simplest example of deadlock is where process 1 has been allocated a non-shareable resource A, say, a tap drive, and process 2 has been allocated a 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 has blocked the other and all useful work in the system stops. This situation is termed as deadlock.

The system is in deadlock state because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.

Example 2: Consider a system with three disk drives. Suppose there are three processes, each is holding one of these three disk drives. If each process now requests another disk drive, three processes will be in a deadlock state, because each process is waiting for the event "disk drive is released", which can only be caused by one of the other waiting processes. Deadlock state involves processes competing not only for the same resource type, but also for different resource types.

Deadlocks occur most commonly in multitasking and client/server environments and are also known as a "Deadly Embrace". Ideally, the programs that are deadlocked or the operating system should resolve the deadlock, but this doesn't always happen.
From the above examples, we have understood the concept of deadlocks. In the examples we were given some instances, but we will study the necessary conditions for a deadlock to occur, in the next section.

CHARACTERISATION OF A DEADLOCK

Coffman (1971) identified **four necessary conditions** that must hold simultaneously for a deadlock to occur.

1. Mutual Exclusion Condition

The resources involved are non-shareable. At least one resource 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

In this condition, a requesting process already holds resources and waiting for the requested resources. A process, holding a resource allocated to it waits for an additional resource(s) that is/are currently being held by other processes.

3. No-Preemptive Condition

Resources already allocated to a process cannot be preempted. Resources cannot be removed forcibly from the processes. After completion, they will be 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.

Let us understand this by a common example. Consider the traffic deadlock shown in the Figure 1.

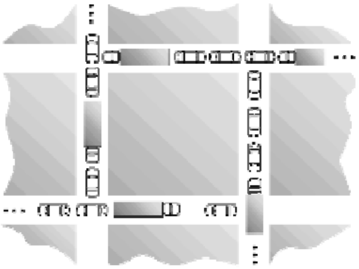


Figure 1: Traffic Deadlock

Consider each section of the street as a resource. In this situation:

- Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.

- 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.
- Non-preemptive condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
- Circular wait condition applies, since each vehicle is waiting for the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of the 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.

A RESOURCE ALLOCATION GRAPH

The idea behind the resource allocation graph is to have a graph which has two different types of nodes, the process nodes and resource nodes (process represented by circles, resource node represented by rectangles). For different instances of a resource, there is a dot in the resource node rectangle. For example, if there are two identical printers, the printer resource might have two dots to indicate that we don't really care which is used, as long as we acquire the resource.

The edges among these nodes represent resource allocation and release. Edges are directed, and if the edge goes from resource to process node that means the process has acquired the resource. If the edge goes from process node to resource node that means the process has requested the resource.

We can use these graphs to determine if a deadline has occurred or may occur. If for example, all resources have only one instance (all resource node rectangles have one dot) and the graph is circular, then a deadlock has occurred. If on the other hand some resources have several instances, then a deadlock may occur. If the graph is not circular, a deadlock cannot occur (the circular wait condition wouldn't be satisfied).

The following are the tips which will help you to check the graph easily to predict the presence of cycles.

- If no cycle exists in the resource allocation graph, there is no deadlock.

- If there is a cycle in the graph and each resource has only one instance, then there is a deadlock. In this case, a cycle is a necessary and sufficient condition for deadlock.

- If there is a cycle in the graph, and each resource has more than one instance, there may or may not be a deadlock. (A cycle may be broken if some process outside the cycle has a resource instance that can break the cycle). Therefore, a cycle in the resource allocation graph is a necessary but not sufficient condition for deadlock, when multiple resource instances are considered.

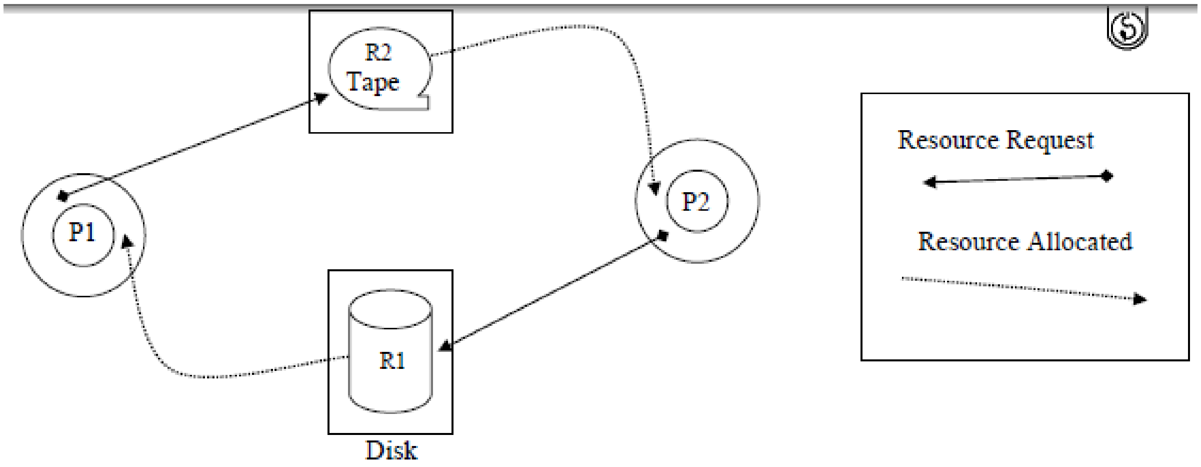


Figure 2: Resource Allocation Graph Showing Deadlock

The above graph shown in *Figure 2* has a cycle and is in Deadlock.

R1 → P1   P1 → R2
R2 → P2   P2 → R1

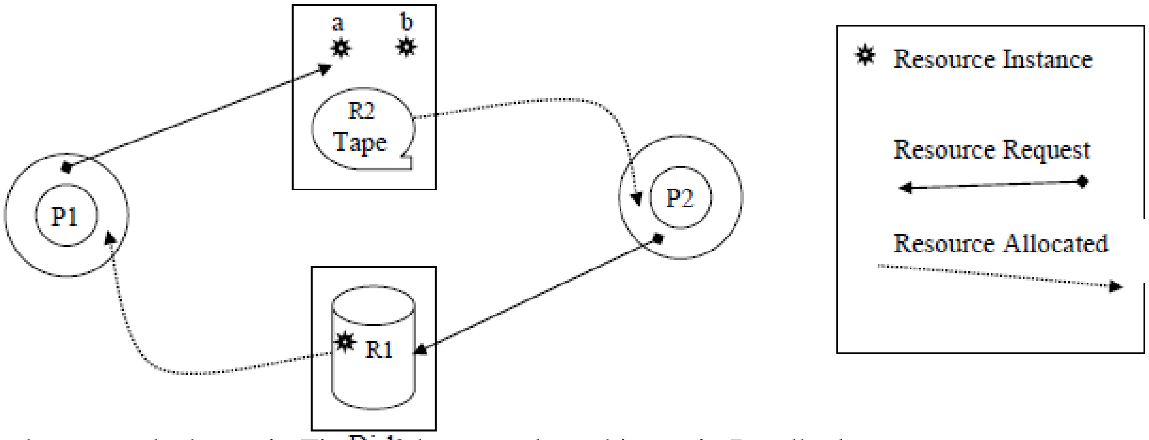


Figure 3: Resource Allocation Graph having a cycle and not in a Deadlock

The above graph shown in Figure 3 has a cycle and is not in Deadlock.

(Resource 1 has one instance shown by a star)

(Resource 2 has two instances a and b, shown as two stars)

R1→ P1 P1→ R2 **(a)**

R2 **(b)** → P2 P2→ R1
If P1 finishes, P2 can get R1 and finish, so there is no Deadlock.

DEALING WITH DEADLOCK SITUATIONS

There are possible strategies to deal with deadlocks. They are:

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

Let's examine each strategy one by one to evaluate their respective strengths and weaknesses.

Deadlock Prevention

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. Let us study Havender's algorithm.

Havender's Algorithm

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-shareable 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.

Elimination of "Hold and Wait" Condition

There are two possibilities for the elimination of the second condition. The first alternative is that a process request be granted all 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 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 drives 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.

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 their needs. 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 the second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources, the process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the “no-preemptive” condition effectively.

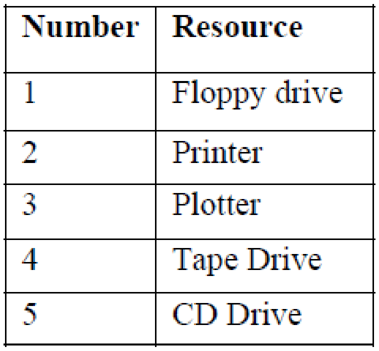
High Cost

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.

Elimination of “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 imposes a total ordering of all resource types, and requires that each process requests resources in a numerical order 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 in the given Table 1:

Table 1: Numbering the resources



Number	Resource
1	Floppy drive
2	Printer
3	Plotter
4	Tape Drive
5	CD Drive

Now we will see the rule for this:

Rule: 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.

This strategy, if adopted, may result in low resource utilisation and in some cases starvation is possible too.

4.5.2 Deadlock Avoidance

This approach to the deadlock problem anticipates a deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. The most famous deadlock avoidance algorithm, from Dijkstra [1965], is the Banker's algorithm. It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not.

The Banker's Algorithm is based on the banking system, which never allocates its available cash in such a manner that it can no longer satisfy the needs of all its customers. Here we must have the advance knowledge of the maximum possible claims for each process, which is limited by the resource availability. During the run of the system we should keep monitoring the resource allocation status to ensure that no circular wait condition can exist.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. The following are the features that are to be considered for avoidance of the deadlock s per the Banker's Algorithms.

- Each process declares maximum number of resources of each type that it may need.
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock.
- Check for the safe state by finding a safe sequence: <P1, P2, ..., Pn> where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < i.
- Resource allocation graph algorithm uses claim edges to check for a safe state.

The resource allocation state is now defined by the number of available and allocated resources, and the maximum demands of the processes. Subsequently the system can be in either of the following states:

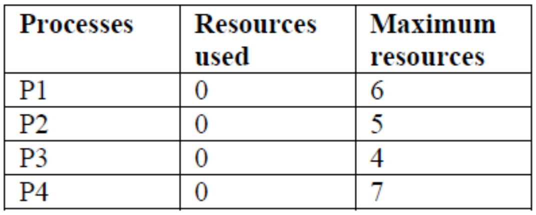
- **Safe state:** Such a state occur when the system can allocate resources to each process (up to its maximum) in some order and avoid a deadlock. This state will be characterised by a safe sequence. It must be mentioned here that we should not falsely conclude that all unsafe states are deadlocked although it may eventually lead to a deadlock.

- **Unsafe State:** If the system did not follow the safe sequence of resource allocation from the beginning and it is now in a situation, which may lead to a deadlock, then it is in an unsafe state.
- **Deadlock State:** If the system has some circular wait condition existing for some processes, then it is in deadlock state.

Let us study this concept with the help of an example as shown below:

Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker.

Table 2

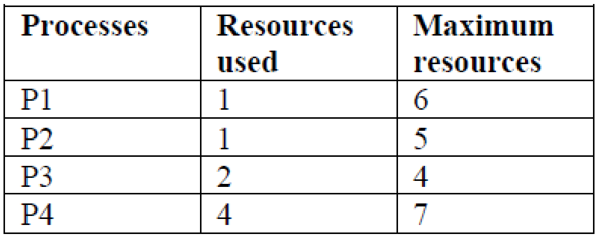


Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Let us assume that total available resources = 10

In the above table, we see four processes, each of which has been granted a number of maximum resources that can be used. The Operating system reserved only 10 resources rather than 22 units to service them. At a certain moment, the situation becomes:

Table 3



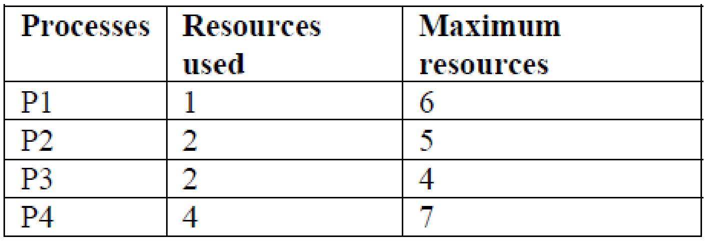
Processes	Resources used	Maximum resources
P1	1	6
P2	1	5
P3	2	4
P4	4	7

Available resources = 2

Safe State: The key to a state being safe is that there is at least one way for all users to finish. In other words the state of Table 2 is safe because with 2 units left, the operating system can delay any request except P3, thus letting P3 finish and release all four resources. With four units in hand, the Operating system can let either P4 or P2 have the necessary units and so on.

Unsafe State: Consider what would happen if a request from P2 for one more unit was granted in Table 3. We would have following situation as shown in Table 4.

Table 4



Processes	Resources used	Maximum resources
P1	1	6
P2	2	5
P3	2	4
P4	4	7

Available resource = 1

This is an unsafe state.

If all the processes request for their maximum resources respectively, then the operating system 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 of 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 used 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 is postponed until later.

Haberman [1969] has shown that executing of the algorithm has a 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.

Limitations of the Banker's Algorithm

There are some problems with the Banker's algorithm as given below:

- It is time consuming to execute on the operation of every resource.
- If the claim information is not accurate, system resources may be underutilised.
- Another difficulty can occur when a system is heavily loaded. Lauesen states that in this situation "so many resources are granted away that very few safe sequences remain, and as a consequence, the jobs will be executed sequentially". Therefore, the Banker's algorithm

is referred to as the “Most Liberal” granting policy; that is, it gives away everything that it can without regard to the consequences.

- New processes arriving may cause a problem.
  - The process’s claim must be less than the total number of units of the resource in the system. If not, the process is not accepted by the manager.
  - Since the state without the new process is safe, so is the state with the new process. Just use the order you had originally and put the new process at the end.
  - Ensuring fairness (starvation freedom) needs a little more work, but isn’t too hard either (once every hour stop taking new processes until all current processes finish).
- A resource becoming unavailable (e.g., a tape drive breaking), can result in an unsafe state.

Deadlock Detection and Recovery

Detection of deadlocks is the most practical policy, which being both liberal and cost efficient, most operating systems deploy. To detect a deadlock, we must go about in a recursive manner and simulate the most favoured execution of each unblocked process.

- An unblocked process may acquire all the needed resources and will execute.
- It will then release all the acquired resources and remain dormant thereafter.
- The now released resources may wake up some previously blocked process.
- Continue the above steps as long as possible.
- If any blocked processes remain, they are **deadlocked.**

Recovery from Deadlock

Recovery by process termination

In this approach we terminate deadlocked processes in a systematic way taking into account their priorities. The moment, enough processes are terminated to recover from deadlock, we stop the process terminations. Though the policy is simple, there are some problems associated with it.

Consider the scenario where a process is in the state of updating a data file and it is terminated. The file may be left in an incorrect state by the unexpected termination of the updating process. Further, processes should be terminated based on some criterion/policy. Some of the criteria may be as follows:

- Priority of a process
- CPU time used and expected usage before completion
- Number and type of resources being used (can they be preempted easily?)
- Number of resources needed for completion
- Number of processes needed to be terminated
- Are the processes interactive or batch?

Recovery by Checkpointing and Rollback (Resource preemption)

Some systems facilitate deadlock recovery by implementing checkpointing and rollback. Checkpointing is saving enough state of a process so that the process can be restarted at the point in the computation where the checkpoint was taken. Autosaving file edits are a form of checkpointing. Checkpointing costs depend on the underlying algorithm. Very simple algorithms (like linear primality testing) can be checkpointed with a few words of data. More complicated processes may have to save all the process state and memory.

If a deadlock is detected, one or more processes are restarted from their last checkpoint. Restarting a process from a checkpoint is called rollback. It is done with the expectation that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low.

Process checkpointing can also be used to improve reliability (long running computations), assist in process migration, or reduce startup costs.

MEMORRY MANAGEMENT – [UNIT III]

INTRODUCTION

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each location with its own address. Interaction is achieved through a sequence of reads/writes of specific memory address. The CPU fetches from the program from the hard disk and stores in memory. If a program is to be executed, it must be mapped to absolute addresses and loaded into memory.

In a multiprogramming environment, in order to improve both the CPU utilisation and the speed of the computer's response, several processes must be kept in memory. There are many different algorithms depending on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The Operating System is responsible for the following activities in connection with memory management:

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

In the multiprogramming environment operating system dynamically allocates memory to multiple processes. Thus memory plays a significant role in the important aspects of computer system like performance, S/W support, reliability and stability.

Memory can be broadly classified into two categories–the primary memory (like cache and RAM) and the secondary memory (like magnetic tape, disk etc.). The memory is a resource that needs effective and efficient management. The part of OS that perform this vital task of memory management is known **as memory manager**. In multiprogramming system, as available memory is shared among number of processes, so the allocation speed and the efficient memory utilisation (in terms of minimal overheads and reuse/relocation of released memory block) are of prime concern. Protection is difficult to achieve with relocation requirement, as location of process and absolute address in memory is unpredictable. But at run-time, it can be done. Fortunately, we have mechanisms supporting protection like processor (hardware) support that is able to abort the instructions violating protection and trying to interrupt other processes.

This unit collectively depicts such memory management related responsibilities in detail by the OS. Further we will discuss, the basic approaches of allocation are of two types:

Contiguous Memory Allocation: Each programs data and instructions are allocated a single contiguous space in memory.
Non-Contiguous Memory Allocation: Each programs data and instructions are allocated memory space that is not continuous. This unit focuses on contiguous memory allocation scheme.

OVERLAYS AND SWAPPING

Usually, programs reside on a disk in the form of executable files and for their execution they must be brought into memory and must be placed within a process. Such programs form the ready queue. In general scenario, processes are fetched from ready queue, loaded into memory and then executed. During these stages, addresses may be represented in different ways like in source code addresses or in symbolic form (ex. LABEL). Compiler will bind this symbolic address to relocatable addresses (for example, 16 bytes from base address or start of module). The linkage editor will bind these relocatable addresses to absolute addresses. Before we learn a program in memory we must bind the memory addresses that the program is going to use. Binding is basically assigning which address the code and data are going to occupy. You can bind at compile-time, load-time or execution time.

Compile-time: If memory location is known a priori, absolute code can be generated.

Load-time: If it is not known, it must generate relocatable at complete time.

Execution-time: Binding is delayed until run-time; process can be moved during its execution. We need H/W support for address maps (base and limit registers).

For better memory utilisation all modules can be kept on disk in a relocatable format and only main program is loaded into memory and executed. Only on need the other routines are called, loaded and address is updated. Such scheme is called *dynamic loading*, which is user's responsibility rather than OS.But Operating System provides library routines to implement dynamic loading.

In the above discussion we have seen that entire program and its related data is loaded in physical memory for execution. But what if process is larger than the amount of memory allocated to it? We can overcome this problem by adopting a technique called as *Overlays*. Like dynamic loading, overlays can also be implemented by users without OS support. The entire program or application is divided into instructions and data sets such that when one instruction set is needed it is loaded in memory and after its execution is over, the space is released. As and when requirement for other instruction arises it is loaded into space that was released previously by the instructions that are no longer needed. Such instructions can be called as overlays, which are loaded and unloaded by the program.

Definition: An overlay is a part of an application, which has been loaded at same origin where previously some other part(s) of the program was residing.

A program based on overlay scheme mainly consists of following:

- A "root" piece which is always memory resident
- Set of overlays.

Overlay gives the program a way to extend limited main storage. An important aspect related to overlays identification in program is concept of mutual exclusion i.e., routines which do not invoke each other and are not loaded in memory simultaneously.
For example, suppose total available memory is 140K. Consider a program with four subroutines named as: ***Read ( ), Function1( ), Function2( )*** and ***Display( ).*** First, *Read* is invoked that reads a set of data. Based on this data set values, conditionally either one of routine *Function1* or *Function2* is called. And then *Display* is called to output results. Here, *Function1* and *Function2*

are mutually exclusive and are not required simultaneously in memory. The memory requirement can be shown as in *Figure 1*:

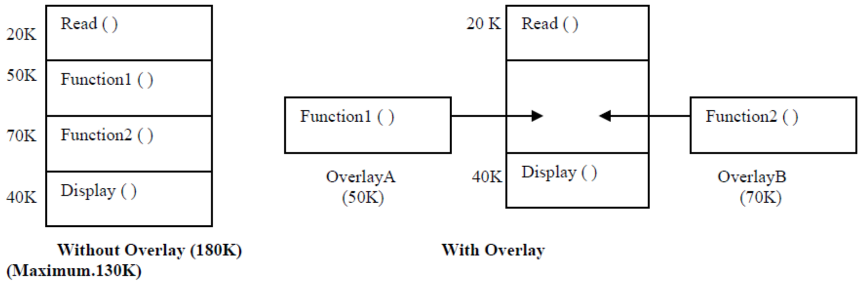


Figure 1: Example of overlay

Without the overlay it requires 180 K of memory and with the overlay support memory requirement is 130K. Overlay manager/driver is responsible for loading and unloading on overlay segment as per requirement. But this scheme suffers from following limitations:

- Require careful and time-consuming planning.
- Programmer is responsible for organising overlay structure of program with the help of file structures etc. and also to ensure that piece of code is already loaded when it is called.
- Operating System provides the facility to load files into overlay region.

Swapping

Swapping is an approach for memory management by bringing each process in entirety, running it and then putting it back on the disk, so that another program may be loaded into that space. Swapping is a technique that lets you use a disk file as an extension of memory. Lower priority user processes are swapped to backing store (disk) when they are waiting for I/O or some other event like arrival of higher priority processes. This is *Rollout Swapping*. Swapping the process back into store when some event occurs or when needed (may be in a different partition) is known as *Roll-in swapping*. *Figure 2* depicts technique of swapping:

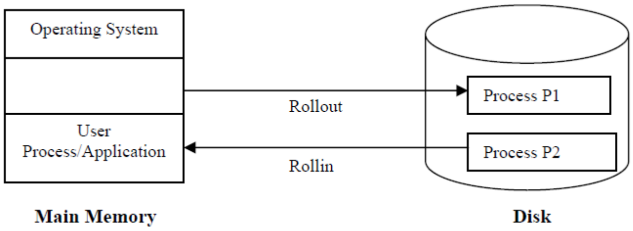


Figure 2: Swapping

Major benefits of using swapping are:

- Allows higher degree of multiprogramming.

- Allows dynamic relocation, i.e., if address binding at execution time is being used we can swap in different location else in case of compile and load time bindings processes have to be moved to same location only.

- Better memory utilisation.

- Less wastage of CPU time on compaction, and

- Can easily be applied on priority-based scheduling algorithms to improve performance.

Though swapping has these benefits but it has few limitations also like entire program must be resident in store when it is executing. Also processes with changing memory requirements will need to issue system calls for requesting and releasing memory. It is necessary to know exactly how much memory a user process is using and also that it is blocked or waiting for I/O.

If we assume a data transfer rate of 1 megabyte/sec and access time of 10 milliseconds, then to actually transfer a 100Kbyte process we require:

Transfer Time = 100K / 1,000 = 1/10 seconds
= 100 milliseconds

Access time = 10 milliseconds

Total time = 110 milliseconds

As both the swap out and swap in should take place, the total swap time is then about 220 milliseconds (above time is doubled). A round robin CPU scheduling should have a time slot size much larger relative to swap time of 220 milliseconds. Also if process is not utilising memory space and just waiting for I/O operation or blocked, it should be swapped.

LOGICAL AND PHYSICAL ADDRESS SPACE

The computer interacts via logical and physical addressing to map memory. Logical address is the one that is generated by CPU, also referred to as virtual address. The program perceives this address space. Physical address is the actual address understood by computer hardware i.e., memory unit. Logical to physical address translation is taken care by the Operating System. The term *virtual memory* refers to the abstraction of separating LOGICAL memory (i.e., memory as seen by the process) from PHYSICAL memory (i.e., memory as seen by the processor). Because of this separation, the programmer needs to be aware of only the logical memory space while the operating system maintains two or more levels of physical memory space.

In compile-time and load-time address binding schemes these two tend to be the same. These differ in execution-time address binding scheme and the MMU (Memory Management Unit) handles translation of these addresses.

Definition: MMU (as shown in the *Figure 3*) is a hardware device that maps logical address to the physical address. It maps the virtual address to the real store location. The simple MMU scheme adds the relocation register contents to the base address of the program that is generated at the time it is sent to memory.

CPU ←------→MMU ←------→ Memory
↕
Address Translation Hardware

Figure 3: Role of MMU

The entire set of logical addresses forms logical address space and set of all corresponding physical addresses makes physical address space.

SINGLE PROCESS MONITOR (MONOPROGRAMMING)

In the simplest case of single-user system everything was easy as at a time there was just one process in memory and no address translation was done by the operating system dynamically during execution. Protection of OS (or part of it) can be achieved by keeping it in ROM.We can also have a separate OS address space only accessible in supervisor mode as shown in *Figure 4*. The user can employ overlays if memory requirement by a program exceeds the size of physical memory. In this approach only one process at a time can be in running state in the system. Example of such system is MS-DOS which is a single tasking system having a command interpreter. Such an arrangement is limited in capability and performance.

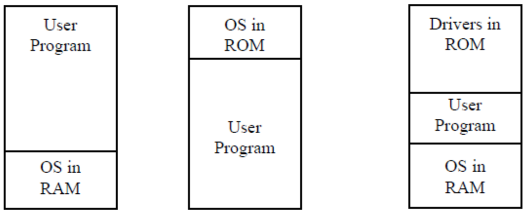


Figure 4: Single Partition System

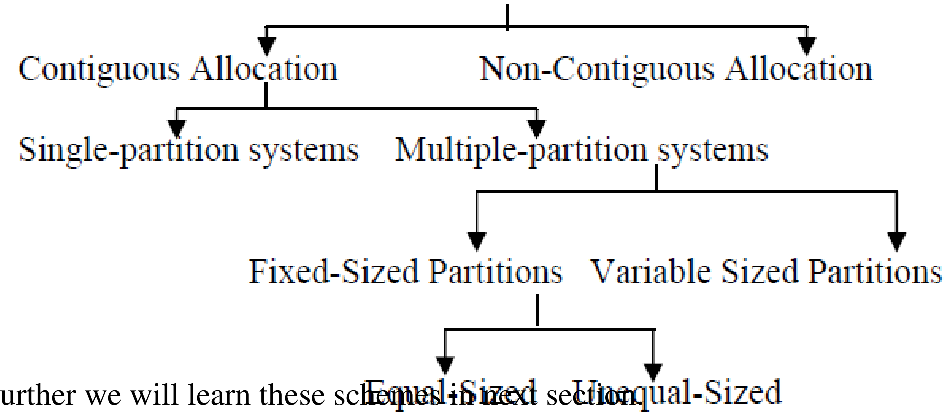
CONTIGUOUS ALLOCATION METHODS

In a practical scenario Operating System could be divided into several categories as shown in the hierarchical chart given below:

1) Single process system

2) Multiple process system with two types: Fixed partition memory and variable partition memory.

Memory Allocation



Further we will learn these schemes in next section.

Partitioned Memory allocation:

The concept of multiprogramming emphasizes on maximizing CPU utilisation by overlapping CPU and I/O.Memory may be allocated as:

- Single large partition for processes to use or
- Multiple partitions with a single process using a single partition.

Single-Partition System

This approach keeps the Operating System in the lower part of the memory and other user processes in the upper part. With this scheme, Operating System can be protected from updating in user processes. Relocation-register scheme known as *dynamic relocation* is useful for this purpose. It not only protects user processes from each other but also from changing OS code and data. Two registers are used: relocation register, contains value of the smallest physical address and limit register, contains logical addresses range. Both these are set by Operating System when the job starts. At load time of program (i.e., when it has to be relocated) we must establish "addressability" by adjusting the relocation register contents to the new starting address for the program. The scheme is shown in *Figure 5.*

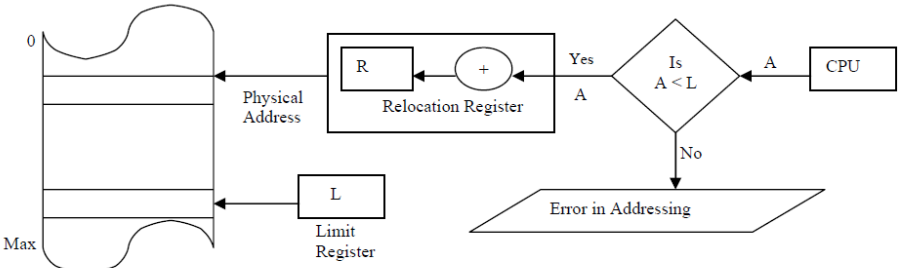


Figure 5: Dynamic Relocation

The contents of a relocation register are implicitly added to any address references generated by the program. Some systems use base registers as relocation register for easy addressability as these are within programmer's control. Also, in some systems, relocation is managed and accessed by Operating System only.

To summarize this, we can say, in dynamic relocation scheme if the logical address space range is 0 to *Max* then physical address space range is R+0 to R+Max (where R is relocation register contents). Similarly, a limit register is checked by H/W to be sure that logical address generated by CPU is not bigger than size of the program.

Multiple Partition System: Fixed-sized partition

This is also known as *static partitioning* scheme as shown in *Figure 6*. Simple memory management scheme is to divide memory into *n* (possibly unequal) fixed-sized partitions, each of which can hold exactly one process. The degree of multiprogramming is dependent on the number of partitions. IBM used this scheme for systems 360 OS/MFT (Multiprogramming with a fixed number of tasks). The partition boundaries are not movable (must reboot to move a job). We can have one queue per partition or just a single queue for all the partitions.

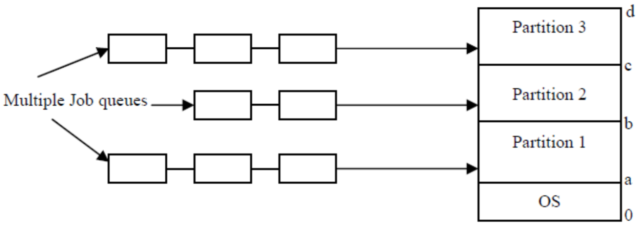


Figure 6: Multiple Partition System

Initially, whole memory is available for user processes and is like a large block of available memory. Operating System keeps details of available memory blocks and occupied blocks in tabular form. OS also keeps track on memory requirements of each process. As processes enter into the input queue and when sufficient space for it is available, process is allocated space and loaded. After its execution is over it releases its occupied space and OS fills this space with other processes in input queue. The block of available memory is known as a Hole. Holes of various sizes are scattered throughout the memory. When any process arrives, it is allocated memory from a hole that is large enough to accommodate it. This example is shown in Figure 7:

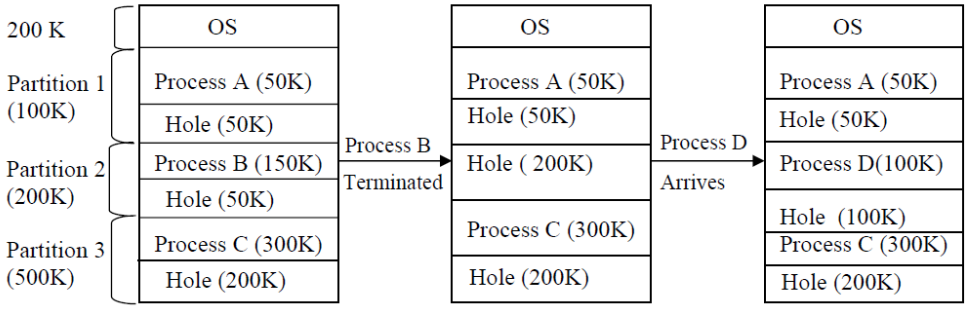


Figure 7: Fixed-sized Partition Scheme

If a hole is too large, it is divided into two parts:

1) One that is allocated to next process of input queue
2) Added with set of holes.

Within a partition if two holes are adjacent then they can be merged to make a single large hole. But this scheme suffers from fragmentation problem. Storage fragmentation occurs either because the user processes do not completely accommodate the allotted partition or partition remains unused, if it is too small to hold any process from input queue. Main memory utilisation is extremely inefficient. Any program, no matter how small, occupies entire partition. In our example, process B takes 150K of partition2 (200K size). We are left with 50K sized hole. This phenomenon, in which there is wasted space internal to a partition, is known as *internal fragmentation*. It occurs because initially process is loaded in partition that is large enough to hold it (i.e., allocated memory may be slightly larger than requested memory). “Internal” here means memory that is internal to a partition, but is not in use.

Variable-sized Partition:

This scheme is also known as *dynamic partitioning*. In this scheme, boundaries are not fixed. Processes accommodate memory according to their requirement. There is no wastage as partition size is exactly same as the size of the user process. Initially Partition 1 (100K) Partition 2 (200K) Partition 3 (500K) 200 when processes start this wastage can be avoided but later on when they terminate they leave holes in the main storage. Other processes can accommodate these, but eventually they become too small to accommodate new jobs as shown in *Figure 8*.

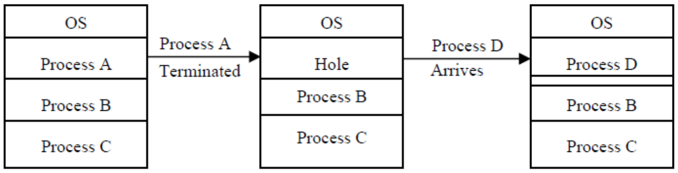


Figure 8: Variable sized partitions

IBM used this technique for OS/MVT (Multiprogramming with a Variable number of Tasks) as the partitions are of variable length and number. But still fragmentation anomaly exists in this scheme. As time goes on and processes are loaded and removed from memory, fragmentation increases and memory utilisation declines. This wastage of memory, which is external to partition, is known as *external fragmentation*. In this, though there is enough total memory to satisfy a request but as it is not contiguous and it is fragmented into small holes, that can't be utilised.

External fragmentation problem can be resolved by **coalescing holes** and **storage compaction**. **Coalescing** holes is process of merging existing hole adjacent to a process that will terminate and free its allocated space. Thus, new adjacent holes and existing holes can be viewed as a single large hole and can be efficiently utilised.

There is another possibility that holes are distributed throughout the memory. For utilising such scattered holes, shuffle all occupied areas of memory to one end and leave all free memory space as a single large block, which can further be utilised. This mechanism is known as *Storage Compaction*, as shown in *Figure 9*.

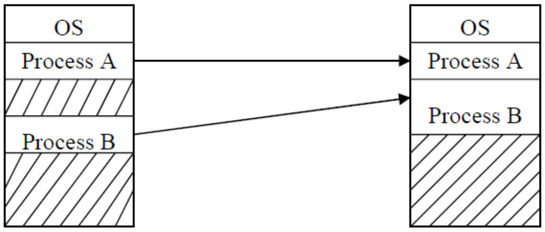


Figure 9: Storage Compaction

But storage compaction also has its limitations as shown below:

1) It requires extra overheads in terms of resource utilisation and large response time.

2) Compaction is required frequently because jobs terminate rapidly. This enhances system resource consumption and makes compaction expensive.

3) Compaction is possible only if dynamic relocation is being used (at run-time). This is because the memory contents that are shuffled (i.e., relocated) and executed in new location require all internal addresses to be relocated.

In a multiprogramming system memory is divided into a number of fixed size or variable sized partitions or regions, which are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where *n* is greater than or equal to *m.* The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmentation and external fragmentation. The difference (*n-m*) is called internal fragmentation, memory which is internal to a partition but is not being use. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

PAGING

We will see the principles of operation of the paging in the next section.

Paging scheme solves the problem faced in variable sized partitions like external fragmentation.

Principles of Operation

In a paged system, logical memory is divided into a number of fixed sizes 'chunks' called *pages*. The physical memory is also predivided into same fixed sized blocks (as is the size of pages) called *page frames*. The page sizes (also the frame sizes) are always powers of 2, and vary between 512 bytes to 8192 bytes per page. The reason behind this is implementation of paging mechanism using page number and page offset. This is discussed in detail in the following sections:

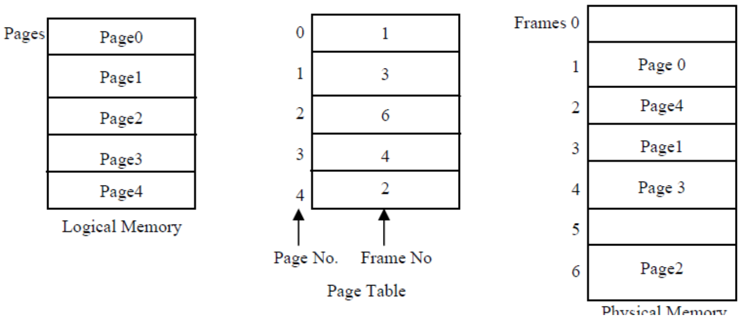


Figure 10: Principle of operation of paging

Each process page is loaded to some memory frame. These pages can be loaded into contiguous frames in memory or into noncontiguous frames also as shown in *Figure 10*. The external fragmentation is alleviated since processes are loaded into separate holes.

Page Allocation

In variable sized partitioning of memory every time when a process of size *n* is to be loaded, it is important to know the best location from the list of available/free holes. This dynamic storage allocation is necessary to increase efficiency and throughput of system. Most commonly used strategies to make such selection are:

1) **Best-fit Policy:** Allocating the hole in which the process fits most "tightly" i.e., the difference between the hole size and the process size is the minimum one.

2) **First-fit Policy:** Allocating the first available hole (according to memory order), which is big enough to accommodate the new process.

3) **Worst-fit Policy:** Allocating the largest hole that will leave maximum amount of unused space i.e., leftover space is maximum after allocation.

Now, question arises which strategy is likely to be used? In practice, best-fit and first-fit are better than worst-fit. Both these are efficient in terms of time and storage requirement. Best-fit minimize the leftover space, create smallest hole that could be rarely used. First-fit on the other hand requires least overheads in its implementation because of its simplicity. Possibly worst-fit also sometimes leaves large holes that could further be used to accommodate other processes. Thus all these policies have their own merits and demerits.

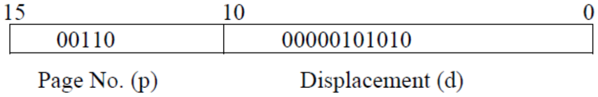
1.6.3 Hardware Support for Paging

Every logical page in paging scheme is divided into two parts:

1) A page number (p) in logical address space

2) The displacement (or offset) in page p at which item resides (i.e., from start of page).

This is known as Address Translation scheme. For example, a 16-bit address can be divided as given in *Figure* below:



Here, as page number takes 5bits, so range of values is 0 to 31(i.e. 2^5-1). Similarly, offset value uses 11-bits, so range is 0 to 2023(i.e., 2^{11} –1). Summarizing this we can say paging scheme uses 32 pages, each with 2024 locations.

The table, which holds virtual address to physical address translations, is called the **page table**. As displacement is constant, so only translation of virtual page number to physical page is required. This can be seen diagrammatically in *Figure 11.*

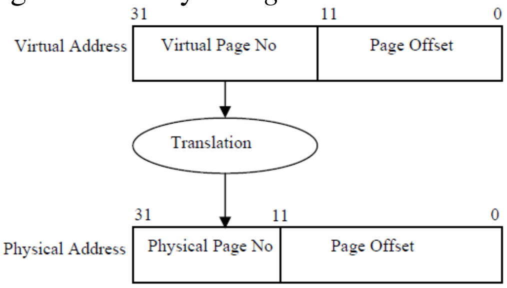
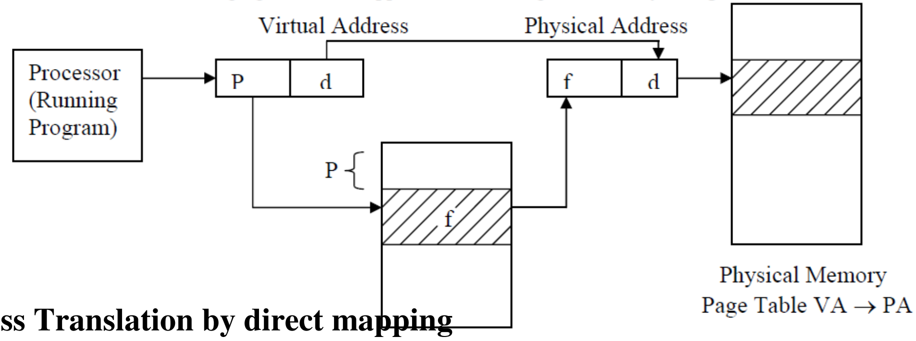


Figure 11: Address Translation scheme

Page number is used as an index into a page table and the latter contains base address of each corresponding physical memory page number (Frame). This reduces dynamic relocation efforts. The Paging hardware support is shown diagrammatically in *Figure 12*:



Paging address Translation by direct mapping

Figure 12: Direct Mapping

This is the case of direct mapping as page table sends directly to physical memory page. This is shown in *Figure 12*. But disadvantage of this scheme is its speed of translation. This is because page table is kept in primary storage and its size can be considerably large which increases instruction execution time (also access time) and hence decreases system speed. To overcome this additional hardware support of registers and buffers can be used. This is explained in next section.

Paging Address Translation with Associative Mapping

This scheme is based on the use of dedicated registers with high speed and efficiency. These small, fast-lookup cache help to place the entire page table into a content-addresses associative storage, hence speed-up the lookup problem with a cache. These are known as associative registers or Translation Look-aside Buffers (TLB's). Each register consists of two entries:

1) Key, which is matched with logical page p.
2) Value which returns page frame number corresponding to p.

It is similar to direct mapping scheme but here as TLB's contain only few page table entries, so search is fast. But it is quite expensive due to register support. So, both direct and associative mapping schemes can also be combined to get more benefits. Here, page number is matched with all associative registers simultaneously. The percentage of the number of times the page is found in TLB's is called hit ratio. If it is not found, it is searched in page table and added into TLB. But if TLB is already full then page replacement policies can be used. Entries in TLB can be limited only. This combined scheme is shown in *Figure 13*.

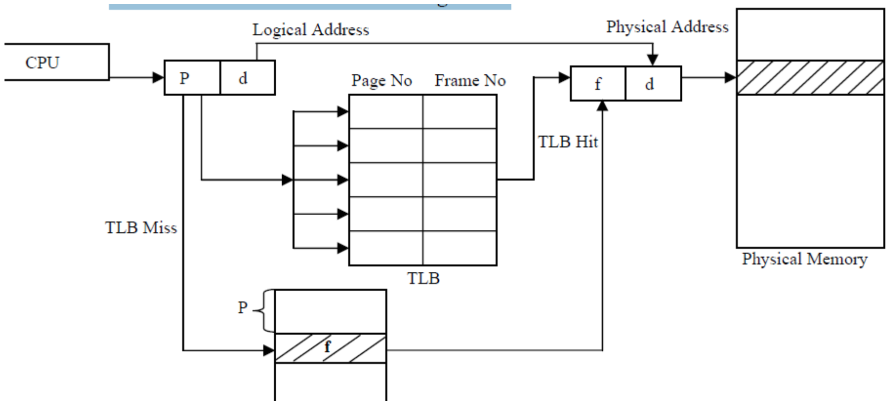


Figure 13: Combined Associative/ Direct Mapping

Protection and Sharing

Paging hardware typically also contains some protection mechanism. In page table corresponding to each frame a protection bit is associated. This bit can tell if page is read-only or read-write. Sharing code and data takes place if two page table entries in different processes point to same physical page, the processes share the memory. If one process writes the data, other process will see the changes. It is a very efficient way to communicate. Sharing must also be controlled to protect modification and accessing data in one process by another process. For this programs are kept separately as procedures and data, where procedures and data that are non-modifiable (pure/reentrant code) can be shared. Reentrant code cannot modify itself and must make sure that it has a separate copy of per-process global variables. Modifiable data and procedures cannot be shared without concurrency controls. Non-modifiable procedures are also known as pure procedures or reentrant codes (can't change during execution). For example, only one copy of editor or compiler code can be kept in memory, and all editor or compiler processes can execute that single copy of the code. This helps memory utilisation. Major advantages of paging scheme are:

1) Virtual address space must be greater than main memory size.i.e., can execute program with large logical address space as compared with physical address space.

2) Avoid external fragmentation and hence storage compaction.

3) Full utilisation of available main storage.

Disadvantages of paging include internal fragmentation problem i.e., wastage within allocated page when process is smaller than page boundary. Also, extra resource consumption and overheads for paging hardware and virtual address to physical address translation takes place.

SEGMENTATION

In the earlier section we have seen the memory management scheme called as paging.
In general, a user or a programmer prefers to view system memory as a collection of variable-sized segments rather than as a linear array of words. Segmentation is a memory management scheme that supports this view of memory.

Principles of Operation

Segmentation presents an alternative scheme for memory management. **This scheme** divides the logical address space into variable length chunks, called segments, with no proper ordering among them. Each segment has a name and a length. For simplicity, segments are referred by a segment number, rather than by a name. Thus, the logical addresses are expressed as a pair of segment number and offset within segment. It allows a program to be broken down into logical parts according to the user view of the memory, which is then mapped into physical memory. Though logical addresses are two-dimensional but actual physical addresses are still one-dimensional array of bytes only.

Address Translation

This mapping between two is done by segment table, which contains segment base and its limit. The segment base has starting physical address of segment, and segment limit provides the length of segment. This scheme is depicted in *Figure 14*.

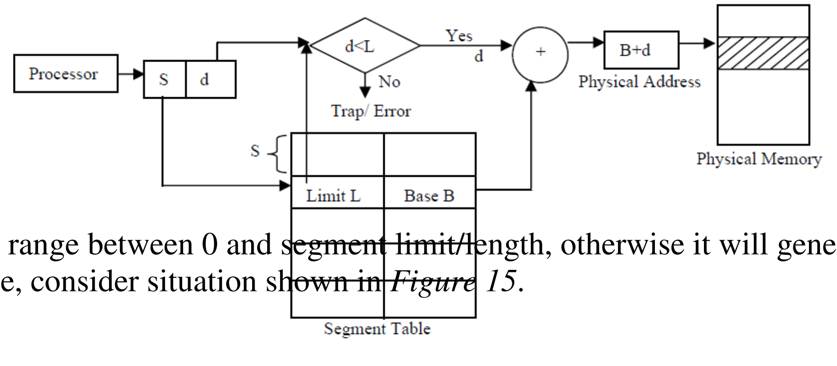
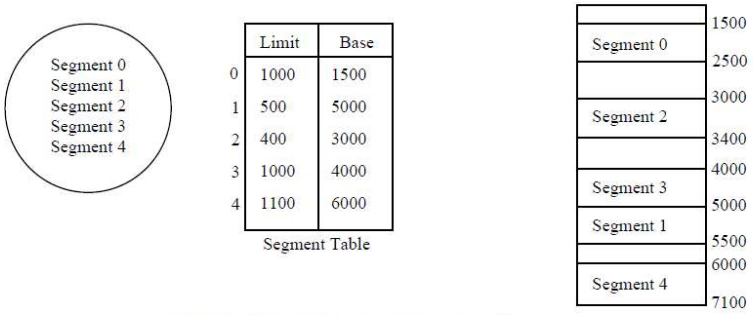


Figure 14: Address Translation

The offset d must range between 0 and segment limit/length, otherwise it will generate address error. For example, consider situation shown in *Figure 15*.



	Limit	Base
0	1000	1500
1	500	5000
2	400	3000
3	1000	4000
4	1100	6000

Figure 15: Principle pf operation of representation

This scheme is similar to variable partition allocation method with improvement that the process is divided into parts. For fast retrieval we can use registers as in paged scheme. This is known as a segment-table length register (STLR). The segments in a segmentation scheme correspond to logical divisions of the process and are defined by program names. Extract the segment number and offset from logical address first. Then use segment number as index into segment table to obtain segment base address and its limit /length. Also, check that the offset is not greater than given limit in segment table. Now, general physical address is obtained by adding the offset to the base address.

Protection and Sharing

This method also allows segments that are read-only to be shared, so that two processes can use shared code for better memory efficiency. The implementation is such that no program can read from or write to segments belonging to another program, except the segments that have been set up to be shared. With each segment-table entry protection bit specifying segment as read-only or execute only can be used. Hence illegal attempts to write into a read-only segment can be prevented.

Sharing of segments can be done by making common /same entries in segment tables of two different processes which point to same physical location. Segmentation may suffer from external fragmentation i.e., when blocks of free memory are not enough to accommodate a segment. Storage compaction and coalescing can minimize this drawback.

I/O AND FILE MANAGEMENT – [UNIT-IV]

INTRODUCTION

Input and output devices are components that form part of the computer system. These devices are controlled by the operating system. Input devices such as keyboard, mouse, and sensors provide input signals such as commands to the operating system. These commands received from input devices instruct the operating system to perform some task or control its behaviour. Output devices such as monitors, printers and speakers are the devices that receive commands or information from the operating system.

In the earlier unit, we had studied the memory management of primary memory. The physical memory, as we have already seen, is not large enough to accommodate all of the needs of a computer system. Also, it is not permanent. Secondary storage consists of disk units and tape drives onto which data can be moved for permanent storage. Though there are actual physical differences between tapes and disks, the principles involved in controlling them are the same, so we shall only consider disk management here in this unit.

The operating system implements the abstract concept of the file by managing mass storage devices, such as types and disks. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage Unit, the **file**. Files are mapped by the operating system, onto physical devices.

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

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

Normally files are organised into directories to ease their use. When multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. The operating system is responsible for the following activities in connection with file management:
The creation and deletion of files.
The creation and deletion of directory.
The support of primitives for manipulating files and directories.
The mapping of files onto disk storage.
Backup of files on stable (non volatile) storage.

The most significant problem in I/O system is the speed mismatch between I/O devices and the memory and also with the processor. This is because I/O system involves both H/W and S/W support and there is large variation in the nature of I/O devices, so they cannot compete with the speed of the processor and memory.

A well-designed file management structure makes the file access quick and easily movable to a new machine. Also it facilitates sharing of files and protection of non-public files. For security and privacy, file system may also provide encryption and decryption capabilities. This makes information accessible to the intended user only.

In this unit we will study the I/O and the file management techniques used by the operating system in order to manage them efficiently.

OBJECTIVES

After going through this unit, you should be able to:

- describe the management of the I/O activities independently and simultaneously with processor activities;
- summarise the full range of views that support file systems, especially the operating system view;
- compare and contrast different approaches to file organisations;
- discuss the disk scheduling techniques, and
- know how to implement the file system and its protection against unauthorised usage.

ORGANISATION OF THE I/O FUNCTION

The range of I/O devices and the large variation in their nature, speed, design, functioning, usage etc. makes it difficult for the operating system to handle them with any generality. The key concept in I/O software designing is device independence achieved by using uniform naming.

The name of the file or device should simply be a string or an integer and not dependent on the device in any way. Another key issue is sharable versus dedicated devices. Many users can share some I/O devices such as disks, at any instance of time. The devices like printers have to be dedicated to a single user until that user has finished an operation.

The basic idea is to organise the I/O software as a series of layers with the lower ones hiding the physical H/W and other complexities from the upper ones that present simple, regular interface interaction with users. Based on this I/O software can be structured in the following four layers given below with brief descriptions:

(i) **Interrupt handlers**: The CPU starts the transfer and goes off to do something else until the interrupt arrives. The I/O device performs its activity independently and simultaneously with CPU activity. This enables the I/O devices to run asynchronously with the processor. The device sends an interrupt to the processor when it has completed the task, enabling CPU to initiate a further task. These interrupts can be hidden with the help of device drivers discussed below as the next I/O software layer.

(ii) **Device Drivers**: Each device driver handles one device type or group of closely related devices and contains a device dependent code. It accepts individual requests from device independent software and checks that the request is carried out. A device driver manages communication with a specific I/O device by converting a logical request from a user into specific commands directed to the device.

(iii) **Device-independent Operating System Software**: Its basic responsibility is to perform the I/O functions common to all devices and to provide interface with user-level software. It also takes care of mapping symbolic device names onto the proper driver. This layer supports device naming, device protection, error reporting, allocating and releasing dedicated devices etc.

(iv) **User level software**: It consists of library procedures linked together with user programs. These libraries make system calls. It makes I/O call, format I/O and also support spooling. Spooling is a way of dealing with dedicated I/O devices like printers in a multiprogramming environment.

I/O BUFFERING

A buffer is an intermediate memory area under operating system control that stores data in transit between two devices or between user's work area and device. It allows computation to proceed in parallel with I/O.

In a typical unbuffered transfer situation the processor is idle for most of the time, waiting for data transfer to complete and total read-processing time is the sum of all the transfer/read time and processor time as shown in Figure 1.

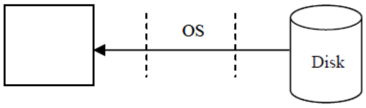


Figure 1: Unbuffered Transfers

In case of single-buffered transfer, blocks are first read into a buffer and then moved to the user's work area. When the move is complete, the next block is read into the buffer and processed in parallel with the first block. This helps in minimizing speed mismatch between devices and the processor. Also, this allows process computation in parallel with input/output as shown in Figure 2.

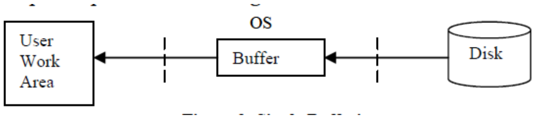


Figure 2: Single Buffering

Double buffering is an improvement over this. A pair of buffers is used; blocks/records generated by a running process are initially stored in the first buffer until it is full. Then from this buffer it is transferred to the secondary storage. During this transfer the other blocks generated are deposited in the second buffer and when this second buffer is also full and first buffer transfer is complete, then transfer from the second buffer is initiated. This process of alternation between buffers continues which allows I/O to occur in parallel with a process's computation. This scheme increases the complexity but yields improved performance as shown in Figure 3.

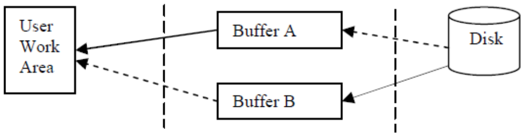


Figure 3: Double Buffering

DISK ORGANISATION

Disks come in different shapes and sizes. The most obvious distinction between floppy disks, diskettes and hard disks is: floppy disks and diskettes consist, of a single disk of magnetic material, while hard-disks normally consist of several stacked on top of one another. Hard disks are totally enclosed devices which are much more finely engineered and therefore require protection from dust. A hard disk spins at a constant speed, while the rotation of floppy drives is switched on and off. On the Macintosh machine, floppy drives have a variable speed operation, whereas most floppy drives have only a single speed of rotation. As hard drives and tape units become more efficient and cheaper to produce, the role of the floppy disk is diminishing. We look therefore mainly at hard drives.

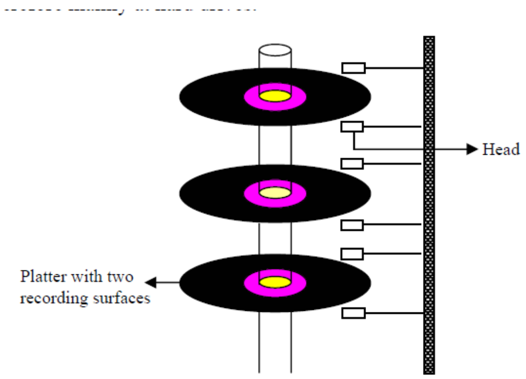


Figure 4: Hard Disk with 3 platters

Looking at the Figure 4, we see that a hard disk is composed of several physical disks stacked on top of each other. The disk shown in the Figure 4 has 3 platters and 6 recording surfaces (two on each platter). A separate read head is provided for each surface. Although the disks are made of continuous magnetic material, there is a limit to the density of information which can be stored on the disk. The heads are controlled by a stepper motor which moves them in fixed-distance intervals across each surface. i.e., there is a fixed number of tracks on each surface. The tracks on all the surfaces are aligned, and the sum of all the tracks at a fixed distance from the edge of the disk is called a cylinder. To make the disk access quicker, tracks are usually divided up into sectors- or fixed size regions which lie along tracks. When writing to a disk, data are written in units of a whole number of sectors. (In this respect, they are similar to pages or frames in physical memory). On some disks, the sizes of sectors are decided by the manufacturers in hardware. On other systems (often microcomputers) it might be chosen in software when the disk is prepared for use (formatting). Because the heads of the disk move together on all surfaces, we can increase read-write efficiency by allocating blocks in parallel across all surfaces. Thus, if a file is stored in consecutive blocks, on a disk with n surfaces and n heads, it could read n sectors per-track without any head movement. When a disk is supplied by a manufacturer, the physical properties of the disk (number of tracks, number of heads, sectors per track, speed of revolution) are provided with the disk. An operating system must be able to adjust to different types of disk. Clearly sectors per track is not a constant, nor is the number of tracks. The numbers given are just a convention used to work out a consistent set of addresses on a disk and may not have anything to do with the hard and fast physical limits of the disk. To address any portion of a disk, we need a three component address consisting of (surface, track and sector).

The seek time is the time required for the disk arm to move the head to the cylinder with the desired sector. The rotational latency is the time required for the disk to rotate the desired sector until it is under the read-write head.

Device drivers and IDs

A hard-disk is a device, and as such, an operating system must use a device controller to talk to it. Some device controllers are simple microprocessors which translate numerical addresses into head motor movements, while others contain small decision making computers of their own. The most popular type of drive for larger personal computers and workstations is the SCSI drive. SCSI (pronounced scuzzy) (Small Computer System Interface) is a protocol and now exists in four variants SCSI 1, SCSI 2, fast SCSI 2, and SCSI 3. SCSI disks live on a data bus which is a fast parallel data link to the CPU and memory, rather like a very short network. Each drive coupled to the bus identifies itself by a SCSI address and each SCSI controller can address up to seven units. If more disks are required, a second controller must be added. SCSI is more efficient at multiple accesses sharing than other disk types for microcomputers. In order to talk to a SCSI disk, an operating system must have a SCSI device driver. This is a layer of software which translates disk requests from the operating system's abstract command-layer into the language of signals which the SCSI controller understands.

Checking Data Consistency and Formatting

Hard drives are not perfect: they develop defects due to magnetic dropout and imperfect manufacturing. On more primitive disks, this is checked when the disk is formatted and these damaged sectors are avoided. If the sector becomes damaged under operation, the structure of the disk must be patched up by some repair program. Usually the data are lost.

On more intelligent drives, like the SCSI drives, the disk itself keeps a defect list which contains a list of all bad sectors. A new disk from the manufacturer contains a starting list and this is updated as time goes by, if more defects occur. Formatting is a process by which the sectors of the disk are:

- (If necessary) created by setting out 'signposts' along the tracks,
- Labelled with an address, so that the disk controller knows when it has found the correct sector.

On simple disks used by microcomputers, formatting is done manually. On other types, like SCSI drives, there is a low-level formatting already on the disk when it comes from the manufacturer. This is part of the SCSI protocol, in a sense. High level formatting on top of this is not necessary, since an advanced enough filesystem will be able to manage the hardware sectors. Data consistency is checked by writing to disk and reading back the result. If there is disagreement, an error occurs. This procedure can best be implemented inside the hardware of the disk-modern disk drives are small computers in their own right. Another cheaper way of checking data consistency is to calculate a number for each sector, based on what data are in the sector and store it in the sector. When the data are read back, the number is recalculated and if there is disagreement then an error is signalled. This is called a cyclic redundancy check (CRC) or error correcting code. Some device controllers are intelligent enough to be able to detect bad sectors and move data to a spare 'good' sector if there is an error. Disk design is still a subject of considerable research and disks are improving both in speed and reliability by leaps and bounds.

DISK SCHEDULING

The disk is a resource which has to be shared. It is therefore has to be scheduled for use, according to some kind of scheduling system. The secondary storage media structure is one of the vital parts of the file system. Disks are the one, providing lot of the secondary storage. As compared to magnetic tapes, disks have very fast access time and disk bandwidth. The access time has two major constituents: seek time and the rotational latency.

The seek time is the time required for the disk arm to move the head to the cylinder with the desired sector. The rotational latency is the time required for the disk to rotate the desired sector until it is under the read-write head. The disk bandwidth is the total number of bytes transferred per unit time.

Both the access time and the bandwidth can be improved by efficient disk I/O requests scheduling. Disk drivers are large single dimensional arrays of logical blocks to be transferred. Because of large usage of disks, proper scheduling algorithms are required.

A scheduling policy should attempt to maximize throughput (defined as the number of requests serviced per unit time) and also to minimize mean response time (i.e., average waiting time plus service time). These scheduling algorithms are discussed below:

FCFS Scheduling

First-Come, First-Served (FCFS) is the basis of this simplest disk scheduling technique. There is no reordering of the queue. Suppose the requests for inputting/ outputting to blocks on the cylinders have arrived, forming the following disk queue:

50, 91, 150, 42, 130, 18, 140, 70, 60

Also assume that the disk head is initially at cylinder 50 then it moves to 91, then to 150 and so on. The total head movement in this scheme is 610 cylinders, which makes the system slow because of wild swings. Proper scheduling while moving towards a particular direction could decrease this. This will further improve performance. FCFS scheme is clearly depicted in Figure 5.

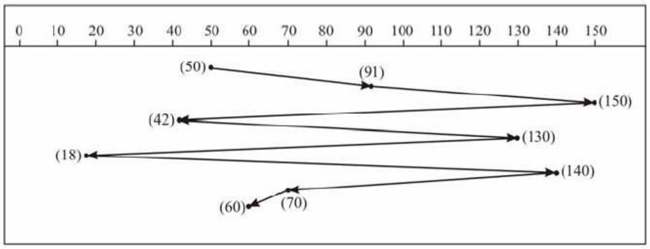


Figure 5: FCFS Scheduling

SSTF Scheduling

The basis for this algorithm is Shortest-Seek-Time-First (SSTF) i.e., service all the requests close to the current head position and with minimum seeks time from current head position.

In the previous disk queue sample the cylinder close to critical head position i.e., 50, is 42 cylinder, next closest request is at 60. From there, the closest one is 70, then 91,130,140,150 and then finally 18-cylinder. This scheme has reduced the total head movements to 248 cylinders and hence improved the performance. Like SJF (Shortest Job First) for CPU scheduling SSTF also suffers from starvation problem. This is because requests may arrive at any time. Suppose we have the requests in disk queue for cylinders 18 and 150, and while servicing the 18-cylinder request, some other request closest to it arrives and it will be serviced next. This can continue further also making the request at 150-cylinder wait for long. Thus a continual stream of requests near one another could arrive and keep the far away request waiting indefinitely. The SSTF is not the optimal scheduling due to the starvation problem. This whole scheduling is shown in Figure 6.

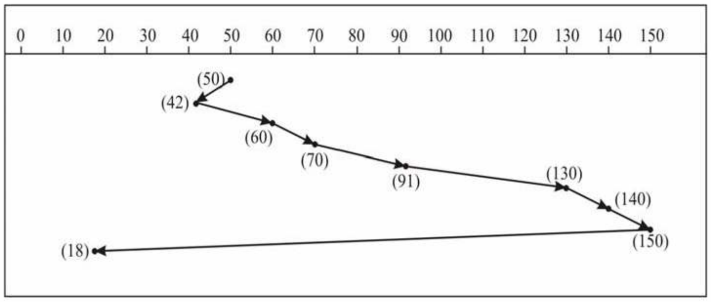


Figure 6: SSTF Scheduling

SCAN Scheduling

The disk arm starts at one end of the disk and service all the requests in its way towards the other end, i.e., until it reaches the other end of the disk where the head movement is reversed and continue servicing in this reverse direction. This scanning is done back and forth by the head continuously.

In the example problem two things must be known before starting the scanning process. Firstly, the initial head position i.e., 50 and then the head movement direction (let it towards 0, starting cylinder). Consider the disk queue again:

91, 150, 42, 130, 18, 140, 70, 60

Starting from 50 it will move towards 0, servicing requests 42 and 18 in between. At cylinder 0 the direction is reversed and the arm will move towards the other end of the disk servicing the requests at 60, 70, 91,130,140 and then finally 150.

As the arm acts like an elevator in a building, the SCAN algorithm is also known as elevator algorithm sometimes. The limitation of this scheme is that few requests need to wait for a long time because of reversal of head direction. This scheduling algorithm results in a total head movement of only 200 cylinders. Figure 7 shows this scheme:

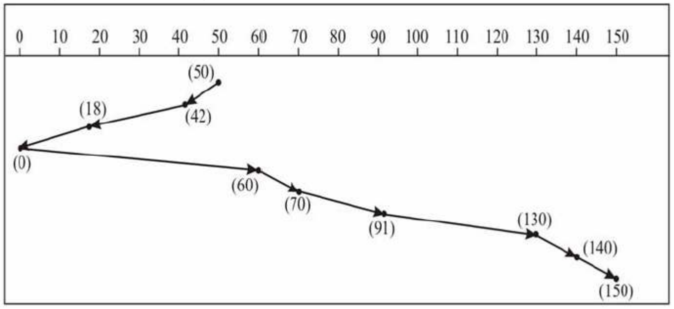


Figure 7: SCAN Scheduling

C-SCAN Scheduling

Similar to SCAN algorithm, C-SCAN also moves head from one end to the other servicing all the requests in its way. The difference here is that after the head reaches the end it immediately returns to beginning, skipping all the requests on the return trip. The servicing of the requests is done only along one path. Thus comparatively this scheme gives uniform wait time because cylinders are like circular lists that wrap around from the last cylinder to the first one.

LOOK and C-LOOK Scheduling

These are just improvements of SCAN and C-SCAN but difficult to implement. Here the head moves only till final request in each direction (first and last ones), and immediately reverses direction without going to end of the disk. Before moving towards any direction the requests are looked, avoiding the full width disk movement by the arm.

The performance and choice of all these scheduling algorithms depend heavily on the number and type of requests and on the nature of disk usage. The file allocation methods like contiguous, linked or indexed, also affect the requests. For example, a contiguously allocated file will generate nearby requests and hence reduce head movements whereas linked or indexed files may generate requests from blocks that are scattered throughout the disk and hence increase the head movements. While searching for files the directories will be frequently accessed, hence location of directories and also blocks of data in them are also important criteria. All these peculiarities force the disk scheduling algorithms to be written as a separate module of the operating system, so that these can easily be replaced. For heavily used disks the SCAN / LOOK algorithms are well suited because they take care of the hardware and access requests in a reasonable order. There is no real danger of starvation, especially in the C-SCAN case. The arrangement of data on a disk plays an important role in deciding the efficiency of data-retrieval.

RAID

Disks have high failure rates and hence there is the risk of loss of data and lots of downtime for restoring and disk replacement. To improve disk usage many techniques have been implemented. One such technology is RAID (Redundant Array of Inexpensive Disks). Its organisation is based on disk striping (or interleaving), which uses a group of disks as one storage unit. Disk striping is a way of increasing the disk transfer rate up to a factor of N, by splitting files across N different disks. Instead of saving all the data from a given file on one disk, it is split across many. Since the N heads can now search independently, the speed of transfer is, in principle, increased manifold. Logical disk data/blocks can be written on two or more separate physical disks which can further transfer their sub-blocks in parallel. The total transfer rate system is directly proportional to the number of disks. The larger the number of physical disks striped together, the larger the total transfer rate of the system. Hence, the overall performance and disk accessing speed is also enhanced. The enhanced version of this scheme is mirroring or shadowing. In this RAID organisation a duplicate copy of each disk is kept. It is costly but a much faster and more reliable approach. The disadvantage with disk striping is that, if one of the N disks becomes damaged, then the data on all N disks is lost. Thus striping needs to be combined with a reliable form of backup in order to be successful.

Another RAID scheme uses some disk space for holding parity blocks. Suppose, three or more disks are used, then one of the disks will act as a parity block, which contains corresponding bit positions in all blocks. In case some error occurs or the disk develops a problems all its data bits can be reconstructed. This technique is known as disk striping with parity or block interleaved parity, which increases speed. But writing or updating any data on a disk requires corresponding recalculations and changes in parity block. To overcome this the parity blocks can be distributed over all disks.

DISK CACHE

Disk caching is an extension of buffering. Cache is derived from the French word cacher, meaning to hide. In this context, a **cache** is a collection of blocks that logically belong on the disk, but are kept in memory for performance reasons. It is used in multiprogramming environment or in disk file servers, which maintain a separate section of main memory called disk cache. These are sets of buffers (cache) that contain the blocks that are recently used. The cached buffers in memory are copies of the disk blocks and if any data here is modified only its local copy is updated. So, to maintain integrity, updated blocks must be transferred back to the disk. Caching is based on the assumption that most shortly accessed blocks are likely to be accessed again soon. In case some new block is required in the cache buffer, one block already there can be selected for “flushing” to the disk. Also to avoid loss of updated information in case of failures or loss of power, the system can periodically flush cache blocks to the disk. The key to disk caching is keeping frequently accessed records in the disk cache buffer in primary storage.

COMMAND LANGUAGE USER’S VIEW OF THE FILE SYSTEM

The most important aspect of a file system is its appearance from the user’s point of view. The user prefers to view the naming scheme of files, the constituents of a file, what the directory tree looks like, protection specifications, file operations allowed on them and many other interface issues. The internal details like, the data structure used for free storage management, number of sectors in a logical block etc. are of less interest to the users. From a user’s perspective, a file is the smallest allotment of logical secondary storage. Users should be able to refer to their files by symbolic names rather than having to use physical device names.

The operating system allows users to define named objects called files which can hold interrelated data, programs or any other thing that the user wants to store/save.

THE SYSTEM PROGRAMMER’S VIEW OF THE FILE SYSTEM

As discussed earlier, the system programmer’s and designer’s view of the file system is mainly concerned with the details/issues like whether linked lists or simple arrays are used for keeping track of free storage and also the number of sectors useful in any logical block. But it is rare that physical record size will exactly match the length of desired logical record. The designers are mainly interested in seeing how disk space is managed, how files are stored and how to make everything work efficiently and reliably.

THE OPERATING SYSTEM'S VIEW OF FILE MANAGEMENT

As discussed earlier, the operating system abstracts (maps) from the physical properties of its storage devices to define a logical storage unit i.e., the file. The operating system provides various system calls for file management like creating, deleting files, read and write, truncate operations etc. All operating systems focus on achieving device-independence by making the access easy regardless of the place of storage (file or device). The files are mapped by the operating system onto physical devices. Many factors are considered for file management by the operating system like directory structure, disk scheduling and management, file related system services, input/output etc. Most operating systems take a different approach to storing information. Three common file organisations are byte sequence, record sequence and tree of disk blocks. UNIX files are structured in simple byte sequence form. In record sequence, arbitrary records can be read or written, but a record cannot be inserted or deleted in the middle of a file. CP/M works according to this scheme. In tree organisation each block hold **n** keyed records and a new record can be inserted anywhere in the tree. The mainframes use this approach. The OS is responsible for the following activities in regard to the file system:

The creation and deletion of files
The creation and deletion of directory
The support of system calls for files and directories manipulation
The mapping of files onto disk
Backup of files on stable storage media (non-volatile).

The coming sub-sections cover these details as viewed by the operating system.

Directories

A file directory is a group of files organised together. An entry within a directory refers to the file or another directory. Hence, a tree structure/hierarchy can be formed. The directories are used to group files belonging to different applications/users. Large-scale time sharing systems and distributed systems store thousands of files and bulk of data. For this type of environment a file system must be organised properly. A File system can be broken into partitions or volumes. They provide separate areas within one disk, each treated as separate storage devices in which files and directories reside. Thus directories enable files to be separated on the basis of user and user applications, thus simplifying system management issues like backups, recovery, security, integrity, name-collision problem (file name clashes), housekeeping of files etc.

The device directory records information like name, location, size, and type for all the files on partition. A root refers to the part of the disk from where the root directory begins, which points to the user directories. The root directory is distinct from sub-directories in that it is in a fixed position and of fixed size. So, the directory is like a symbol table that converts file names into their corresponding directory entries. The operations performed on a directory or file system are:

1) Create, delete and modify files.
2) Search for a file.
3) Mechanisms for sharing files should provide controlled access like read, write, execute or various combinations.
4) List the files in a directory and also contents of the directory entry.
5) Renaming a file when its contents or uses change or file position needs to be changed.

6) Backup and recovery capabilities must be provided to prevent accidental loss or malicious destruction of information.
7) Traverse the file system.

The most common schemes for describing logical directory structure are:

(i) **Single-level directory**

All the files are inside the same directory, which is simple and easy to understand; but the limitation is that all files must have unique names. Also, even with a single user as the number of files increases, it is difficult to remember and to track the names of all the files. This hierarchy is depicted in Figure 8.

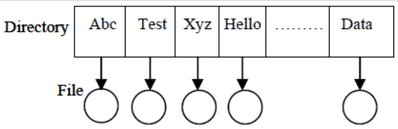


Figure 8: Single-level directory

(ii) **Two-level directory**

We can overcome the limitations of the previous scheme by creating a separate directory for each user, called User File Directory (UFD). Initially when the user logs in, the system's Master File Directory (MFD) is searched which is indexed with respect to username/account and UFD reference for that user. Thus different users may have same file names but within each UFD they should be unique. This resolves name-collision problem up to some extent but this directory structure isolates one user from another, which is not desired sometimes when users need to share or cooperate on some task. Figure 9 shows this scheme clearly.

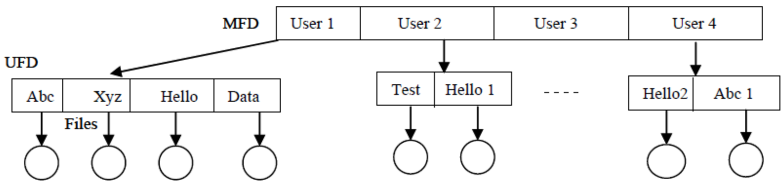


Figure 9: Two-level directory

(iii) **Tree-structured directory**

The two-level directory structure is like a 2-level tree. Thus to generalise, we can extend the directory structure to a tree of arbitrary height. Thus the user can create his/her own directory and subdirectories and can also organise files. One bit in each directory entry defines entry as a file (0) or as a subdirectory (1).

The tree has a root directory and every file in it has a unique path name (path from root, through all subdirectories, to a specified file). The pathname prefixes the filename, helps to reach the required file traversed from a base directory. The pathnames can be of 2 types: absolute path names or relative path names, depending on the base directory. An absolute path name begins at the root and follows a path to a particular file. It is a full pathname and uses the root directory. Relative defines the path from the current directory. For example, if we assume that the current directory is /Hello2 then the file F4.doc has the absolute pathname /Hello/Hello2/Test2/F4.doc and the relative pathname is /Test2/F4.doc. The pathname is used to simplify the searching of a file in a tree-structured directory hierarchy. Figure 10 shows the layout:

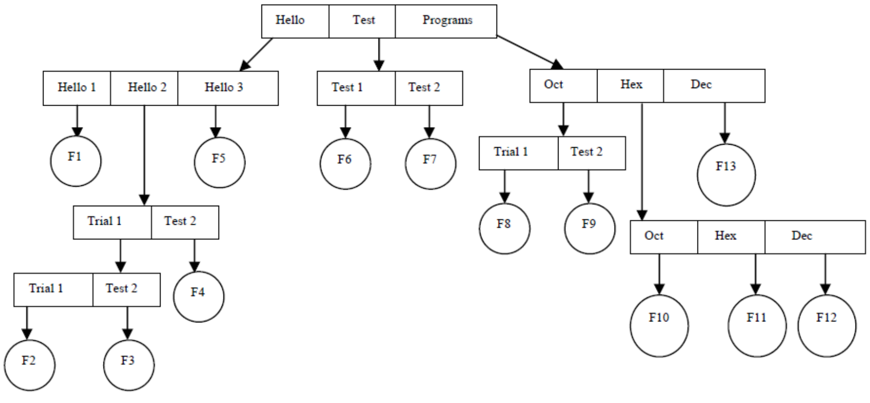


Figure 10: Tree-structured directory

(iv) **Acyclic-graph directory**:

As the name suggests, this scheme has a graph with no cycles allowed in it. This scheme added the concept of shared common subdirectory / file which exists in file system in two (or more) places at once. Having two copies of a file does not reflect changes in one copy corresponding to changes made in the other copy.

But in a shared file, only one actual file is used and hence changes are visible. Thus an acyclic graph is a generalisation of a tree-structured directory scheme. This is useful in a situation where several people are working as a team, and need access to shared files kept together in one directory. This scheme can be implemented by creating a new directory entry known as a link which points to another file or subdirectory. Figure 11 depicts this structure for directories.

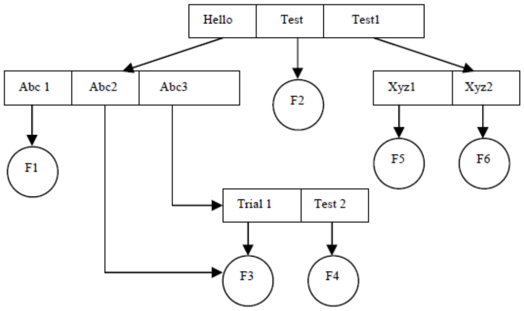


Figure 11: Acyclic-graph directory

The limitations of this approach are the difficulties in traversing an entire file system because of multiple absolute path names. Another issue is the presence of dangling pointers to the files that are already deleted, though we can overcome this by preserving the file until all references to it are deleted. For this, every time a link or a copy of directory is established, a new entry is added to the file-reference list. But in reality as the list is too lengthy, only a count of the number of references is kept. This count is then incremented or decremented when the reference to the file is added or it is deleted respectively.

(v) **General graph Directory**:

Acyclic-graph does not allow cycles in it. However, when cycles exist, the reference count may be non-zero, even when the directory or file is not referenced anymore. In such situation garbage collection is useful. This scheme requires the traversal of the whole file system and marking accessible entries only. The second pass then collects everything that is unmarked on a free-space list. This is depicted in Figure 12.

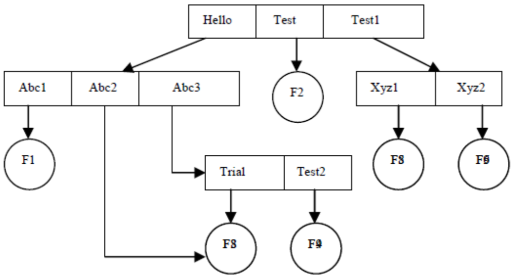


Figure 12: General-graph directory

Disk Space Management

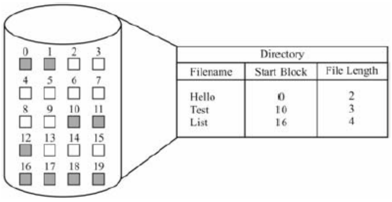
The direct-access of disks and keeping files in adjacent areas of the disk is highly desirable**.** But the problem is how to allocate space to files for effective disk space utilisation and quick access. Also, as files are allocated and freed, the space on a disk become fragmented. The major methods of allocating disk space are:

i) Continuous
ii) Non-continuous (Indexing and Chaining)

i) **Continuous**

This is also known as contiguous allocation as each file in this scheme occupies a set of contiguous blocks on the disk. A linear ordering of disk addresses is seen on the disk. It is used in VM/CMS– an old interactive system. The advantage of this approach is that successive logical records are physically adjacent and require no head movement. So disk seek time is minimal and speeds up access of records. Also, this scheme is relatively simple to implement. The technique in which the operating system provides units of file space on demand by user running processes, is known as dynamic allocation of disk space. Generally space is allocated in units of a fixed size, called an allocation unit or a 'cluster' in MS-DOS. Cluster is a simple multiple of the disk physical sector size, usually 512 bytes. Disk space can be considered as a one-dimensional array of data stores, each store being a cluster. A larger cluster size reduces the potential for fragmentation, but increases the likelihood that clusters will have unused space. Using clusters larger than one sector reduces fragmentation, and reduces the amount of disk space needed to store the information about the used and unused areas on the disk.

Contiguous allocation merely retains the disk address (start of file) and length (in block units) of the first block. If a file is n blocks long and it begins with location b (blocks), then it occupies b, b+1, b+2,…, b+n-1 blocks. First-fit and best-fit strategies can be used to select a free hole from available ones. But the major problem here is searching for sufficient space for a new file. Figure 13 depicts this scheme:



Directory		
Filename	Start Block	File Length
Hello	0	2
Test	10	3
List	16	4

Figure 13: Contiguous Allocation on the Disk

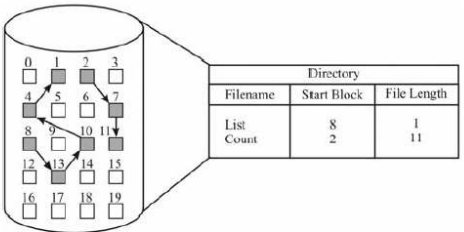
This scheme exhibits similar fragmentation problems as in variable memory partitioning. This is because allocation and deal location could result in regions of free disk space broken into chunks (pieces) within active space, which is called external fragmentation. A repacking routine called compaction can solve this problem. In this routine, an entire file system is copied on to tape or another disk and the original disk is then freed completely. Then from the copied disk, files are again stored back using contiguous space on the original disk. But this approach can be very expensive in terms of time. Also, size-declaration in advance is a problem because each time, the size of file is not predictable. But it supports both sequential and direct accessing. For sequential access, almost no seeks are required. Even direct access with seek and read is fast. Also, calculation of blocks holding data is quick and easy as we need just offset from the start of the file.

ii) Non-Continuous (Chaining and Indexing)

This scheme has replaced the previous ones. The popular non-contiguous storages allocation schemes are:

Linked/Chained allocation
Indexed Allocation.

Linked/Chained allocation: All files are stored in fixed size blocks and adjacent blocks are linked together like a linked list. The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last block of the file. Also each block contains pointers to the next block, which are not made available to the user. There is no external fragmentation in this as any free block can be utilised for storage. So, compaction and relocation is not required. But the disadvantage here is that it is potentially inefficient for direct-accessible files since blocks are scattered over the disk and have to follow pointers from one disk block to the next. It can be used effectively for sequential access only but there also it may generate long seeks between blocks. Another issue is the extra storage space required for pointers. Yet the reliability problem is also there due to loss/damage of any pointer. The use of doubly linked lists could be a solution to this problem but it would add more overheads for each file. A doubly linked list also facilitates searching as blocks are threaded both forward and backward. The Figure 14 depicts linked /chained allocation where each block contains the information about the next block (i.e., pointer to next block).



Directory		
Filename	Start Block	File Length
List	8	1
Count	2	11

Figure 14: Linked Allocation on the Disk 51 **I/O and File Management**

MS-DOS and OS/2 use another variation on linked list called FAT (File Allocation Table). The beginning of each partition contains a table having one entry for each disk block and is indexed by the block number. The directory entry contains the block number of the first block of file. The table entry indexed by block number contains the block number of the next block in the file. The Table pointer of the last block in the file has EOF pointer value. This chain continues until EOF (end of file) table entry is encountered. We still have to linearly traverse next pointers, but at least we don't have to go to the disk for each of them. 0(Zero) table value indicates an unused block. So, allocation of free blocks with FAT scheme is straightforward, just search for the first block with 0 table pointer. MS-DOS and OS/2 use this scheme. This scheme is depicted in Figure 15.

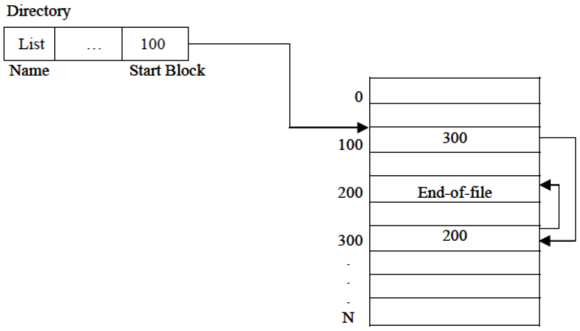


Figure 15: File-Allocation Table (FAT)

Indexed Allocation: In this each file has its own index block. Each entry of the index points to the disk blocks containing actual file data i.e., the index keeps an array of block pointers for each file. So, index block is an array of disk block addresses. The ith entry in the index block points to the ith block of the file. Also, the main directory contains the address where the index block is on the disk. Initially, all the pointers in index block are set to **NIL**. The advantage of this scheme is that it supports both sequential and random access. The searching may take place in index blocks themselves. The index blocks may be kept close together in secondary storage to minimize seek time. Also space is wasted only on the index which is not very large and there's no external fragmentation. But a few limitations of the previous scheme still exists in this, like, we still need to set maximum file length and we can have overflow scheme of the file larger than the predicted value. Insertions can require complete reconstruction of index blocks also. The indexed allocation scheme is diagrammatically shown in Figure 16.

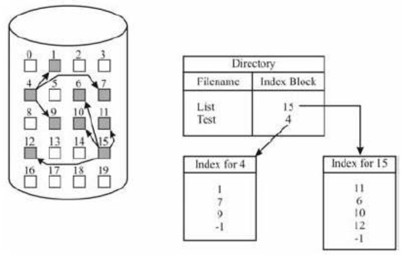


Figure 16: Indexed Allocation on the Disk

Disk Address Translation

We have seen in Unit-1 memory management that the virtual addresses generated by a program is different from the physical. The translation of virtual addresses to physical addresses is performed by MMU. Disk address translation considers the aspects of data storage on the disk. Hard disks are totally enclosed devices, which are more finely engineered and therefore require protection from dust. A hard disk spins at a constant speed. Briefly, hard disks consist of one or more rotating platters. A read-write head is positioned above the rotating surface and is able to read or write the data underneath the current head position. The hard drives are able to present the "geometry" or "addressing scheme" of the drive. Consider the disk internals first. Each track of the disk is divided into sections called sectors. A sector is the smallest physical storage unit on the disk. The size of a sector is always a power of two, and is almost always 512 bytes. A sector is the part of a slice that lies within a track. The position of the head determines which track is being read. A cylinder is almost the same as a track, except that it means all tracks lining up under each other on all the surfaces. The head is equivalent to side(s). It simply means one of the rotating platters or one of the sides on one of the platters. If a hard disk has three rotating platters, it usually has 5 or 6 readable sides, each with its own read-write head.

The MS-DOS file systems allocate storage in clusters, where a cluster is one or more contiguous sectors. MS-DOS bases the cluster size on the size of the partition. As a file is written on the disk, the file system allocates the appropriate number of clusters to store the file's data. For the purposes of isolating special areas of the disk, most operating systems allow the disk surface to be divided into partitions. A partition (also called a cylinder group) is just that: a group of cylinders, which lie next to each other. By defining partitions we divide up the storage of data to special areas, for convenience. Each partition is assigned a separate logical device and each device can only write to the cylinders, which are defined as being its own. To access the disk the computer needs to convert physical disk geometry (the number of cylinders on the disk, number of heads per cylinder, and sectors per track) to a logical configuration that is compatible with the operating system. This conversion is called translation. Since sector translation works between the disk itself and the system BIOS or firmware, the operating system is unaware of the actual characteristics of the disk, if the number of cylinders, heads, and sectors per track the computer needs is within the range supported by the disk. MS-DOS presents disk devices as logical volumes that are associated with a drive code (A, B, C, and so on) and have a volume name (optional), a root directory, and from zero to many additional directories and files.

File Related System Services

A file system enables applications to store and retrieve files on storage devices. Files are placed in a hierarchical structure. The file system specifies naming conventions for files and the format for specifying the path to a file in the tree structure. OS provides the ability to perform input and output (I/O) operations on storage components located on local and remote computers. In this section we briefly describe the system services, which relate to file management. We can broadly classify these under categories:

i) Online services
ii) Programming services.

i) **Online-services**: Most operating systems provide interactive facilities to enable the on-line users to work with files. Few of these facilities are built-in commands of the system while others are provided by separate utility programs. But basic operating systems like MS-DOS with limited security provisions can be potentially risky because of these user owned powers. So, these must be used by technical support staff or experienced users only. For example: DEL *. * Command can erase all the files in the current directory. Also, FORMAT c: can erase the entire contents of the mentioned drive/disk. Many such services provided by the operating system related to directory operations are listed below:

Create a file

Delete a file

Copy a file

Rename a file

Display a file

Create a directory

Remove an empty directory

List the contents of a directory

Search for a file

Traverse the file system.

ii) Programming services: The complexity of the file services offered by the operating system vary from one operating system to another but the basic set of operations like: open (make the file ready for processing), close (make a file unavailable for processing), read (input data from the file), write (output data to the file), seek (select a position in file for data transfer).

All these operations are used in the form of language syntax procedures or built-in library routines, of high-level language used like C, Pascal, and Basic etc. More complicated file operations supported by the operating system provide wider range of facilities/services. These include facilities like reading and writing records, locating a record with respect to a primary key value etc. The software interrupts can also be used for activating operating system functions. For example, Interrupt 21(hex) function call request in MS-DOS helps in opening, reading and writing operations on a file.

In addition to file functions described above the operating system must provide directory operation support also like:

Create or remove directory
Change directory
Read a directory entry
Change a directory entry etc.

These are not always implemented in a high level language but language can be supplied with these procedure libraries. For example, UNIX uses C language as system programming language, so that all system calls requests are implemented as C procedures.

Asynchronous Input/Output

Synchronous I/O is based on blocking concept while asynchronous is interrupt-driven transfer. If an user program is doing several things simultaneously and request for I/O operation, two possibilities arise. The simplest one is that the I/O is started, then after its completion, control is transferred back to the user process. This is known as synchronous I/O where you make an I/O request and you have to wait for it to finish. This could be a problem where you would like to do some background processing and wait for a key press. Asynchronous I/O solves this problem, which is the second possibility. In this, control is returned back to the user program without waiting for the I/O completion. The I/O then continues while other system operations occur. The CPU starts the transfer and goes off to do something else until the interrupt arrives.

Asynchronous I/O operations run in the background and do not block user applications. This improves performance, because I/O operations and applications processing can run simultaneously. Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. To manage asynchronous I/O, each asynchronous I/O request has a corresponding control block in the application’s address space. This control block contains the control and status information for the request. It can be used again when the I/O operation is completed. Most physical I/O is asynchronous. After issuing an asynchronous I/O request, the user application can determine when and how the I/O operation is completed. This information is provided in any of three ways:

- The application can poll the status of the I/O operation.
- The system can asynchronously notify the application when the I/O operation is done.
- The application can block until the I/O operation is complete.

Each I/O is handled by using a device-status table. This table holds entry for each I/O device indicating device’s type, address, and its current status like bust or idle. When any I/O device needs service, it interrupts the operating system. After determining the device, the Operating System checks its status and modifies table entry reflecting the interrupt occurrence. Control is then returned back to the user process.

VIRTUAL MEMORY

Storage allocation has always been an important consideration in computer programming due to the high cost of the main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in the main memory but the main memory may not be large enough to accommodate the needs of an entire process. Early computer programmers divided programs into the sections that were transferred into the main memory for the period of processing time. As the program proceeded, new sections moved into the main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

As higher-level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.

Two theories for solving the problem of inefficient memory management emerged -- static and dynamic allocation. **Static** allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted. **Dynamic** allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

Program objectives and machine advancements in the 1960s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible for storage allocation, which would be accomplished by system calls to allocate or deal locate memory. The second group supported **automatic storage allocation** performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to its very high cost. The second proposal is known as virtual memory.

VIRTUAL MEMORY

It is common for modern processors to be running multiple processes at one time. Each process has an address space associated with it. To create a whole complete address space for each process would be much too expensive, considering that processes may be created and killed often, and also considering that many processes use only a tiny bit of their possible address space. Last but not the least, even with modern improvements in hardware technology, machine resources are still finite. Thus, it is necessary to share a smaller amount of physical memory among many processes, with each process being given the appearance of having its own exclusive address space.

The most common way of doing this is a technique called virtual memory, which has been known since the 1960s but has become common on computer systems since the late 1980s. The virtual memory scheme divides physical memory into blocks and allocates blocks to different processes. Of course, in order to do this sensibly it is highly desirable to have a protection scheme that restricts a process to be able to access only those blocks that are assigned to it. Such a protection scheme is thus a necessary, and somewhat involved, aspect of any virtual memory implementation.

One other advantage of using virtual memory that may not be immediately apparent is that it often reduces the time taken to launch a program, since not all the program code and data need to be in physical memory before the program execution can be started.

Although sharing the physical address space is a desirable end, it was not the sole reason that virtual memory became common on contemporary systems. Until the late 1980s, if a program became too large to fit in one piece in physical memory, it was the programmer's job to see that it fit. Programmers typically did this by breaking programs into pieces, each of which was mutually exclusive in its logic. When a program was launched, a main piece that initiated the execution would first be loaded into physical memory, and then the other parts, called overlays, would be loaded as needed.

It was the programmer's task to ensure that the program never tried to access more physical memory than was available on the machine, and also to ensure that the proper overlay was loaded into physical memory whenever required. These responsibilities made for complex challenges for programmers, who had to be able to divide their programs into logically separate fragments, and specify a proper scheme to load the right fragment at the right time. Virtual memory came about as a means to relieve programmers creating large pieces of software of the wearisome burden of designing overlays.

Virtual memory automatically manages two levels of the memory hierarchy, representing the main memory and the secondary storage, in a manner that is invisible to the program that is running. The program itself never has to bother with the physical location of any fragment of the virtual address space. A mechanism called relocation allows for the same program to run in any location in physical memory, as well. Prior to the use of virtual memory, it was common for machines to include a relocation register just for that purpose. An expensive and messy solution to the hardware solution of a virtual memory would be software that changed all addresses in a program each time it was run. Such a solution would increase the running times of programs significantly, among other things.

Virtual memory enables a program to ignore the physical location of any desired block of its address space; a process can simply seek to access any block of its address space without concern for where that block might be located. If the block happens to be located in the main memory, access is carried out smoothly and quickly; else, the virtual memory has to bring the block in from secondary storage and allow it to be accessed by the program.

The technique of virtual memory is similar to a degree with the use of processor caches. However, the differences lie in the block size of virtual memory being typically much larger (64 kilobytes and up) as compared with the typical processor cache (128 bytes and up). The hit time, the miss penalty (the time taken to retrieve an item that is not in the cache or primary storage), and the transfer time are all larger in case of virtual memory. However, the miss rate is typically much smaller. (This is no accident-since a secondary storage device, typically a magnetic storage device with much lower access speeds, has to be read in case of a miss, designers of virtual memory make every effort to reduce the miss rate to a level even much lower than that allowed in processor caches).

Virtual memory systems are of two basic kinds—those using fixed-size blocks called pages, and those that use variable-sized blocks called segments.

Suppose, for example, that a main memory of 64 megabytes is required but only 32 megabytes is actually available. To create the illusion of the larger memory space, the memory manager would divide the required space into units called pages and store the contents of these pages in mass storage. A typical page size is no more than four kilobytes. As different pages are actually required in main memory, the memory manager would exchange them for pages that are no longer required, and thus the other software units could execute as though there were actually 64 megabytes of main memory in the machine.

In brief we can say that virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that the program can be larger than physical memory. Virtual memory can be implemented via demand paging and demand segmentation.

Principles of Operation

The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilise. For example, virtual memory might contain twice as many addresses as main memory. A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into the main memory those portions of the program needed at any given point during execution.

To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit (MMU) translates virtual addresses to physical addresses at run-time. The process of translating virtual addresses into real addresses is called **mapping**. The copying of virtual pages from disk to main memory is known as **paging** or **swapping**.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimisation it provides is that it is faster to read the information from physical memory than use the relevant I/O channel to get that information. This is called **caching**. It is implemented inside the OS.

Virtual Memory Management

This section provides the description of how the virtual memory manager provides virtual memory. It explains how the logical and physical address spaces are mapped to one another and when it is required to use the services provided by the Virtual Memory Manager.

Before going into the details of the management of the virtual memory, let us see the functions of the virtual memory manager. It is responsible to:

- Make portions of the logical address space resident in physical RAM
- Make portions of the logical address space immovable in physical RAM
- Map logical to physical addresses

- Defer execution of application-defined interrupt code until a safe time.

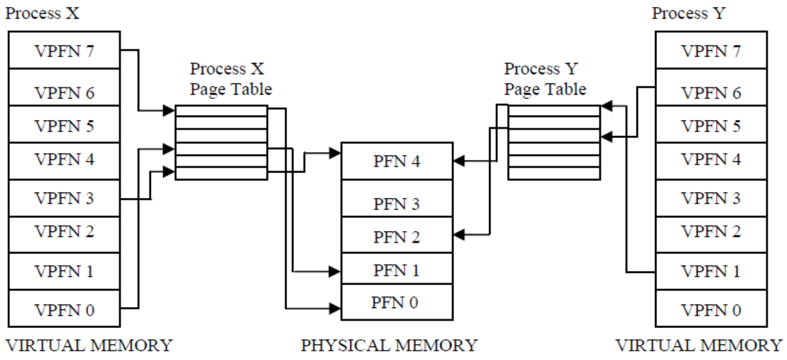


Figure 1: Abstract model of Virtual to Physical address mapping

Before considering the methods that various operating systems use to support virtual memory, it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location of operands in the memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into small blocks called **pages**. These pages are all of the same size. (It is not necessary that all the pages should be of same size but if they were not, the system would be very hard to administer). Linux on Alpha AXP systems uses 8 Kbytes pages and on Intel x86 systems it uses 4 Kbytes pages. Each of these pages is given a unique number; the page frame number (PFN) as shown in the Figure 1. In this paged model, a virtual address is composed of two parts, an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses **page tables**.

The Figure1 shows the virtual address spaces of two processes, process X and process Y, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process X's virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process Y's virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- Valid flag : This indicates if this page table entry is valid.

- PFN : The physical page frame number that this entry is describing.

- Access control information : This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at the Figure1 and assuming a page size of 0x2000 bytes (which is decimal 8192) and an address of 0x2194 in process Y's virtual address space then the processor would translate that address into offset 0x194 into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However, the processor delivers it, this is known as a **page fault** and the operating system is notified of the faulting virtual address and the reason for the page fault. A page fault is serviced in a number of steps:

i) Trap to the OS.

ii) Save registers and process state for the current process.

iii) Check if the trap was caused because of a page fault and whether the page reference is legal.

iv) If yes, determine the location of the required page on the backing store.

v) Find a free frame.

vi) Read the required page from the backing store into the free frame. (During this I/O, the processor may be scheduled to some other process).

vii) When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process.

viii) Modify the corresponding PT entry to show that the recently copied page is now in memory.

ix) Resume execution with the instruction that caused the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process Y's virtual page frame number 1 is mapped to physical page frame number 4 which starts at 0x8000 (4 x 0x2000). Adding in the 0x194 byte offset gives us a final physical address of 0x8194. By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order.

Protection and Sharing

Memory protection in a paged environment is realised by protection bits associated with each page, which are normally stored in the page table. Each bit determines a page to be read-only or read/write. At the same time the physical address is calculated with the help of the page table, the protection bits are checked to verify whether the memory access type is correct or not. For example, an attempt to write to a read-only memory page generates a trap to the operating system indicating a memory protection violation.

We can define one more protection bit added to each entry in the page table to determine an invalid program-generated address. This bit is called valid/invalid bit, and is used to generate a trap to the operating system. Valid/invalid bits are also used to allow and disallow access to the corresponding page. This is shown in Figure 2.

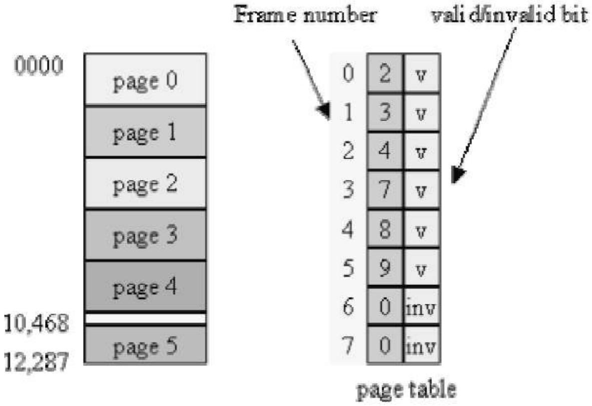


Figure 2: Protection bit in the page table Shared pages

The paging scheme supports the possibility of sharing common program code. For example, a system that supports 40 users, each of them executes a text editor. If the text editor consists of 30 KB of code and 5 KB of data, we need 1400 KB. If the code is reentrant, i.e., it never changes by any write operation during execution (non-self-modifying code) it could be shared as presented in Figure 3.

Only one copy of the editor needs to be stored in the physical memory. In each page table, the included editor page is mapped onto the same physical copy of the editor, but the data pages are mapped onto different frames. So, to support 40 users, we only need one copy of the editor, i.e., 30 KB, plus 40 copies of the 5 KB of data pages per user; the total required space is now 230 KB instead of 1400 KB.

Other heavily used programs such as assembler, compiler, database systems etc. can also be shared among different users. The only condition for it is that the code must be reentrant. It is crucial to correct the functionality of shared paging scheme so that the pages are unchanged. If one user wants to change a location, it would be changed for all other users.

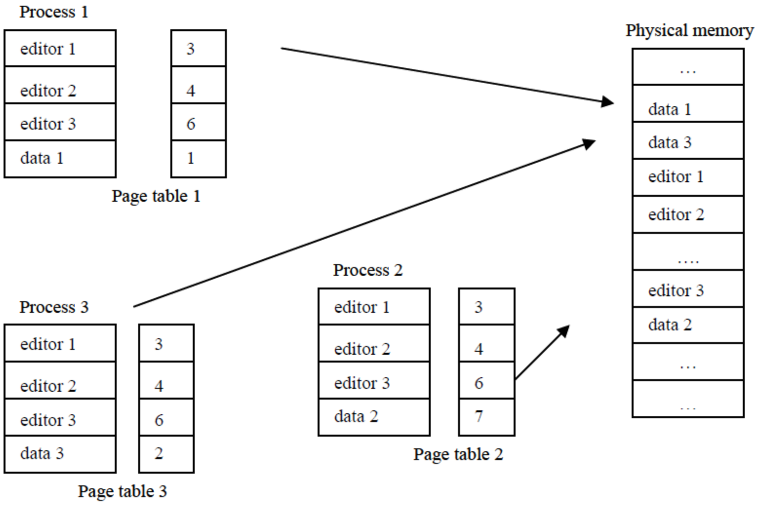


Figure 3: Paging scheme supporting the sharing of program code

DEMAND PAGING

In a multiprogramming system memory is divided into a number of fixed-size or variable-sized partitions or regions that are allocated to running processes. For example: a process needs m words of memory may run in a partition of n words where n ≥ m. The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmented into many scattered blocks. We distinguish between internal fragmentation and external fragmentation. The difference (n – m) is called internal fragmentation, memory that is internal to a partition but is not being used. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either **compact** the memory making large free memory blocks, or implement **paging** scheme which allows a program's memory to be non-contiguous, thus permitting a program to be allocated to physical memory.

Physical memory is divided into fixed size blocks called **frames**. Logical memory is also divided into blocks of the same, fixed size called **pages**. When a program is to be executed, its pages are loaded into any available memory frames from the disk. The disk is also divided into fixed sized blocks that are the same size as the memory frames.

A very important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. Normally, a user believes that memory is one contiguous space containing only his/her program. In fact, the logical memory is scattered through the physical memory that also contains other programs. Thus, the user can work correctly with his/her own view of memory because of the address translation or address mapping. The address mapping, which is controlled by the operating system and transparent to users, translates logical memory addresses into physical addresses.

Because the operating system is managing the memory, it must be sure about the nature of physical memory, for example: which frames are available, which are allocated; how many total frames there are, and so on. All these parameters are kept in a data structure called **frame table** that has one entry for each physical frame of memory indicating whether it is free or allocated, and if allocated, to which page of which process.

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to load only virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it is not necessary to load the code from the database that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as **demand paging**.

When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in Figure 1, there is no entry in Process X's page table for virtual PFN 2 and so if Process X attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a **page fault** has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory.

This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

The valid/invalid bit of the page table entry for a page, which is swapped in, is set as valid. Otherwise it is set as invalid, which will have no effect as long as the program never attempts to access this page. If all and only those pages actually needed are swapped in, the process will execute exactly as if all pages were brought in.

If the process tries to access a page, which was not swapped in, i.e., the valid/invalid bit of this page table, entry is set to invalid, then a page fault trap will occur. Instead of showing the "invalid address error" as usually, it indicates the operating system's failure to bring a valid part of the program into memory at the right time in order to minimize swapping overhead.

In order to continue the execution of process, the operating system schedules a disk read operation to bring the desired page into a newly allocated frame. After that, the corresponding page table entry will be modified to indicate that the page is now in memory. Because the state (program counter, registers etc.) of the interrupted process was saved when the page fault trap occurred, the interrupted process can be restarted at the same place and state. As shown, it is possible to execute programs even though parts of it are not (yet) in memory.

In the extreme case, a process without pages in memory could be executed. Page fault trap would occur with the first instruction. After this page was brought into memory, the process would continue to execute. In this way, page fault trap would occur further until every page that is needed was in memory. This kind of paging is called **pure demand paging**. Pure demand paging says that "never bring a page into memory until it is required".

PAGE REPLACEMENT POLICIES

Basic to the implementation of virtual memory is the concept of **demand paging**. This means that the operating system, and not the programmer, controls the swapping of pages in and out of main memory, as the active processes require them. When a process needs a non-resident page, the operating system must decide which resident page is to be replaced by the requested page. The part of the virtual memory which makes this decision is called the **replacement policy**.

There are many approaches to the problem of deciding which page is to replace but the object is the same for all-the policy that selects the page that will not be referenced again for the longest time. A few page replacement policies are described below.

First In First Out (FIFO)

The First In First Out (FIFO) replacement policy chooses the page that has been in the memory the longest to be the one replaced.

Belady's Anomaly

Normally, as the number of page frames increases, the number of page faults should decrease. However, for FIFO there are cases where this generalisation will fail! This is called Belady's Anomaly. Notice that OPT's never suffers from Belady's anomaly.

Second Chance (SC)

The Second Chance (SC) policy is a slight modification of FIFO in order to avoid the problem of replacing a heavily used page. In this policy, a reference bit R is used to keep track of pages that have been recently referenced. This bit is set to 1 each time the page is referenced. Periodically, all the reference bits are set to 0 by the operating system to distinguish pages that have not been referenced recently from those that have been. Using this bit, the operating system can determine whether old pages are still being used (i.e., R = 1). If so, the page is moved to the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. Thus heavily accessed pages are given a "second chance."

Least Recently Used (LRU)

The Least Recently Used (LRU) replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

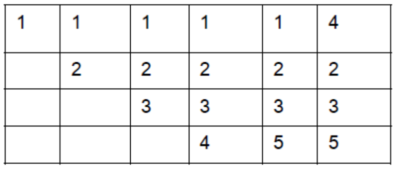
Optimal Algorithm (OPT)

Optimal algorithm is defined as replace the page that will not be used for the longest period of time. It is optimal in the performance but not feasible to implement because we cannot predict future time.

Example:

Let us consider a 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



1	1	1	1	1	4
	2	2	2	2	2
		3	3	3	3
			4	5	5

In the above Figure, we have 6 page faults.

Least Frequently Used (LFU)

The Least Frequently Used (LFU) replacement policy selects a page for replacement if the page had not been used often in the past. This policy keeps count of the number of times that a page is accessed. Pages with the lowest counts are replaced while pages with higher counts remain in primary memory.

UNIX Case Study – [UNIT V]

The Creation of the UNIX Operating System

- After three decades of use, the UNIX computer operating system from Bell Labs is still regarded as one of the most powerful, versatile, and flexible operating systems (OS) in the computer world.
- Its popularity is due to many factors, including its ability to run a wide variety of machines, from micros to supercomputers, and its portability -- all of which led to its adoption by many manufacturers.
- Like another legendary creature whose name also ends in 'x,' UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system.
- The joint effort was not successful, but a few survivors from Bell Labs tried again, and what followed was a system that offers its users a work environment that has been described as "of unusual simplicity, power, and elegance".
- The system also fostered a distinctive approach to software design -- solving a problem by interconnecting simpler tools, rather than creating large monolithic application programs.
- Its development and evolution led to a new philosophy of computing, and it has been a never-ending source of both challenges and joy to programmers around the world.

An Overview of the UNIX Operating System

- The UNIX Operating system was designed to let a number of programmers access the computer at the same time and share its resources.
- The operating system coordinates the use of the computer's resources, allowing one person, for example, to run a spell check program while another creates a document, lets another edit a document while another creates graphics, and lets another user format a document -- all at the same time, with each user oblivious to the activities of the others.
- The operating system controls all of the commands from all of the keyboards and all of the data being generated, and permits each user to believe he or she is the only person working on the computer.

- This real-time sharing of resources make UNIX one of the most powerful operating systems ever.
- Although UNIX was developed by programmers for programmers, it provides an environment so powerful and flexible that it is found in businesses, sciences, academia, and industry.
- Many telecommunications switches and transmission systems also are controlled by administration and maintenance systems based on UNIX.
- While initially designed for medium-sized minicomputers, the operating system was soon moved to larger, more powerful mainframe computers.
- As personal computers grew in popularity, versions of UNIX found their way into these boxes, and a number of companies produce UNIX-based machines for the scientific and programming communities.

The Features of UNIX:

The features that made UNIX most popular are under :

- Multitasking capability.
- Multiuser capability.
- Portability.
- Hierarchical file system.
- Tools for program development.

1 Multitasking

Many computers do just one thing at a time, as anyone who uses a PC or laptop can attest.

Try logging onto your company's network while opening your browser while opening a word processing program.

Chances are the processor will freeze for a few seconds while it sorts out the multiple instructions.

UNIX, on the other hand, lets a computer do several things at once, such as printing out one file while the user edits another file.

This is a major feature for users, since users don't have to wait for one application to end before starting another one.

2 Multiusers

The same design that permits multitasking permits multiple users to use the computer.

The computer can take the commands of a number of users determined by the design of the computer -- to run programs, access files, and print documents at the same time.

The computer can't tell the printer to print all the requests at once, but it does prioritize the requests to keep everything orderly.

It also lets several users access the same document by compartmentalizing the document so that the changes of one user don't override the changes of another user.

3 System portability

A major contribution of the UNIX system was its portability, permitting it to move from one brand of computer to another with a minimum of code changes.

At a time when different computer lines of the same vendor didn't talk to each other -- yet alone machines of multiple vendors -- that meant a great savings in both hardware and software upgrades.

It also meant that the operating system could be upgraded without having all the customer's data inputted again.

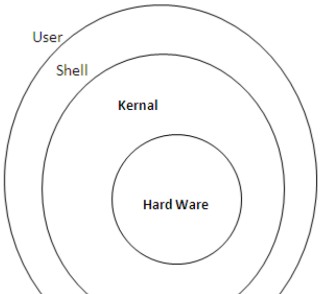
And new versions of UNIX were backward compatible with older versions, making it easier for companies to upgrade in an orderly manner.

- **Hierarchical File System:** To support file organization and maintenance in a easier manner.
- **Tools for program development:** Support wide range of support tools (debuggers, compliers).

How UNIX is organized

The UNIX system is functionally organized at three levels:

- The kernel, which schedules tasks and manages storage;
- The shell, which connects and interprets users' commands, calls programs from memory, and executes them; and
- The tools and applications that offer additional functionality to the operating system



The kernel

- The heart of the operating system.
- The kernel controls the hardware and turns part of the system on and off at the programer's command.
- If you ask the computer to list (*ls*) all the files in a directory, the kernel tells the computer to read all the files in that directory from the disk and display them on your screen.

The shell

- There are several types of shell, most notably the command driven Bourne Shell and the C Shell
- (no pun intended), and menu-driven shells that make it easier for beginners to use.
- Whatever shell is used, its purpose remains the same -- to act as an interpreter between the user and the computer.
- The shell also provides the functionality of "pipes," whereby a number of commands can be linked together by a user, permitting the output of one program to become the input to another program.

Tools and applications

- There are hundreds of tools available to UNIX users, although some have been written by third party vendors for specific applications.
- Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming

UNIX PROCESS MANAGEMENT

- Every process has a process identification number that is used as the index to the process table.
- The Unix Operating system provides a number of system calls for process management. The fork system call is used to create an identical copy of a process.

- The created process is called the child process and the original process is called as the parent-process.
- The exit( ) system call is called by a process that wants to terminate itself and leave a status to its parent.
- Inter-process synchronization can be achieved as follows.
- A parent process can get blocked using the wait system call until the termination of one of its child process.
- Signals are used to inform a process about the occurrence of an event.
- The kernel handles the signals in the context of the process that receive the signal by executing a routine called as the signal handler.
- The exec system call and its variation are used to execute another program by passing the arguments and environmental variables.
- The nice system call can be used by a process to control their scheduling priority.
- **Management of the processes by the Kernal:**
- For each new process created, the kernel sets up an address space in the memory. This address space consists of the following logical segments.
  - **Text:**- Contains the program instructions.
  - **Data:**- Contains the initialized program variables.
  - **Bss:**- Contains un-initialized program variables.
  - **Stack:**- A dynamically grow-able segment, it contains variables allocated locally and parameters passed to functions in the program.
  - Each process has two stacks i.e. User stack and the Kernal stack.

The UNIX File System

- Physical disks are partitioned into different file systems. Each file system has a maximum size, and a maximum number of files and directories that it can contain.
- ***The file system can be seen with the df/dfspace command.*** Different system will have their file systems laid our differently.
- The / directory is called the root of the file system.

- The Unix file system store all the information that relates to the long term state of the system.
- This state includes the operating system kernel, executable files, configuration information, temporary work files, user data, and various special files that are used to give controlled access to system hardware and operating system.
- The functions usually performed on a file are as follows:
- Opening a file for processing .
- Reading data from a file for processing.
- Writing data to a file after processing
- Closing a file after all the necessary processing has been done.

<u>The constituents of Unix file system can be one of the following types</u>

1. **<u>Ordinary Files or Regular Files</u>:** Ordinary files can contain text, data, or program information.

- Files cannot contain other files or directories.
- Unix file names not broken into a name part and an extension part, instead they can contain any keyboard character except for **'/'** and be up 256 characters long.
- However characters such as *,?,# and & have special meaning in most shells and should not therefore be used in file names.
- Putting spaces in file names also makes them difficult to manipulate, so it is always preferred to use the underscore. The is not end of file concept in Unix file.

2. **<u>Directory Files:</u>**

- A *directory file* is a special file that contains information about the various files stored in the directory, such as file locations, file sizes, times of file creation, and file modifications.
- This special file can be read only by the UNIX operating system or programs expressly written to do directory processing.
- You may not view the content of the directory file, but you may use UNIX commands to inquire about these attributes of the directory.
- A file directory is like a telephone directory that contains address information about the files in it.

- When you ask UNIX to process a filename, UNIX looks up the specified directory to obtain information about the file. In each directory, you will always find two files:

<u>In each directory, you will always find two files</u>

1. . (single period)
2. .. (two consecutive periods)

  - The single period (.) refers to the current directory, and the two consecutive periods (..) refer to the directory one level up (sometimes referred to as parent directory).
  - An example of the directory attributes of a testdir are presented here:
  - ***drwxr-xr-x 2 raja Group1 512 Oct 30 18:39 testdir***rwxr-xr-x defines the permissions of testdir created by a user called raja belonging to a group called Group. The size of the directory entry testdir is 512 bytes. The directory was last modified on October 30 at 6:39 p.m.

3 <u>Character and Block Device Files</u>

- The character special files are used for unbuffered I/O to and from a device, and the block special files are used when data is transferred in fixed-size packets.
- The character special files do I/O on one character at a time mode while the block special file use buffer caching mechanism to increase the efficiency of data transfer by keeping in-memory copy of the data.
- Some examples of these files are
  - Floppy disk device--character or block special file
  - Tape device--character special file
  - Terminal--character special file
- UNIX treats the keyboard and the monitor (terminal) as files.
- The keyboard is considered an input file, also referred to as a *standard input file* (*stdin* in UNIX terminology).
- The terminal is considered an output file, also referred to as the *standard output file* (*stdout* in UNIX terminology).
- An important corollary of the standard input and output is referred to as *I/O redirection*.

- In UNIX, using I/O redirection makes it possible to change the standard input file from keyboard to a regular file, and change the standard output file from terminal to a new or existing regular file.
- All UNIX commands, by default, accept input from standard input, display output on standard output, and send error messages to standard error output.
- By using I/O redirection, it is possible to control the source and destination of the command input and output, respectively.
- It is possible to direct the output of a command to a different file than the standard output. Similarly, it is possible to accept the input from a file rather than standard input.
- It is also possible to direct error messages to a file rather than the standard output.
- This gives you the flexibility to run commands in background, where these special files, that is, standard input, standard output, and standard error output, are not available.
- You can use regular files to redirect these input, output, or error messages when you are running commands in the background.

List of standard UNIX directories.

Directory name	**Details about the directory**
/	Root directory. This is the parent of all the directories and files in the UNIX file system.
/bin	Command-line executable directory. This directory contains all the UNIX native command executables.
/dev	Device directory containing special files for character- and block-oriented devices such as printers and keyboards. A file called null existing in this directory is called the bit bucket and can be used to redirect output to nowhere.
/etc	System configuration files and executable directory. Most of the administrative, command-related files are stored here.
/lib	The library files for various programming languages such as C are stored in this directory.

/lost+found	This directory contains the in-process files if the system shuts down abnormally. The system uses this directory to recover these files. There is one lost+found directory in all disk partitions.
/u	Conventionally, all the user home directories are defined under this directory.
/usr	This directory has a number of subdirectories (such as adm, bin, etc, and include. For example, /usr/include has various header files for the C programming language.

Symbolic and Hard Links

- Links create pointers to the actual files, without duplicating the contents of the files.
- That is, a link is a way of providing another name to the same file.
- There are two types of links to a file:
  - Hard link
  - Symbolic (or soft) link; also referred to as *symlink*

Hard link

- With hard links, the original filename and the linked filename point to the same physical address and are absolutely identical.
- There are two important limitations of a hard link.
- A directory cannot have a hard link, and it cannot cross a file system. (A *file system* is a physical space within which a file must reside; a single file cannot span more than one file system, but a file system can have more than one file in it.)
- It is possible to delete the original filename without deleting the linked filename.
- Under such circumstances, the file is not deleted, but the directory entry of the original file is deleted and the link count is decremented by 1.
- The data blocks of the file are deleted when the link count becomes zero.

Symbolic Links

- A symbolic link (sometimes referred to as a *symlink*) differs from a hard link because it doesn't point to another inode but to another filename.
- This allows symbolic links to exist across file systems as well as be recognized as a special file to the operating system.
- You will find symbolic links to be crucial to the administration of your file systems, especially when trying to give the appearance of a seamless system when there isn't one.
- Symbolic links are created using the ln -s command. A common thing people do is create a symbolic link to a directory that has moved.
- For example, if you are accustomed to accessing the directory for your home page in the subdirectory www but at the new site you work at, home pages are kept in the public_html directory, you can create a symbolic link from www to public_html using the command ln -s public_html www.
- Performing an ls -l on the result shows the link.

-
  ```
  drwx------    2 sshah    admin        512 May 12 13:08 public_html
  ```

-
  ```
  lrwx------    1 sshah    admin         11 May 12 13:08 www -> public_html
  ```

<u>File and Directory Permissions</u>

- In UNIX, each user is identified with a unique *login id.*
- Additionally, multiple users can be grouped and associated with a *group.*
- A user can belong to one or more of these groups. However, a user belongs to one *primary group*. All other groups to which a user belongs are called *secondary groups.*
- The user login id is defined in the /etc/passwd file, and the user group is defined in /usr/group file.
- The file and directory permissions in UNIX are based on the user and group.
- All the permissions associated with a file or a directory have three types of permissions: ----
- **Permissions for the owner:** This identifies the operations the owner of the file or the directory can perform on the file or the directory
- **Permissions for the group:** This identifies the operations that can be performed by any user belonging to the same group as the owner of the file or the directory.

- **Permissions for Other Groups:** This identifies the operations everybody else (other than the owner and members of the group to which the owner belongs) can do.
- Using the permission attributes of a file or directory, a user can selectively provide access to users belonging to a particular group and users not belonging to a particular group.
- UNIX checks on the permissions in the order of owner, group, and other group and the first permission that is applicable to the current user is used.

The Permission Bits

- You know that files and directories have owners and groups associated with them.
- The following are three set of permissions associated with a file or directory:
  - Owner permission
  - Group permission
  - Others permission
- For each of these three types for permissions there are three permission bits associated.
- The following is a list of these permission bits and their meanings for files:
- Read (r): The file can be read.
- Write (w): The file can be modified, deleted, and renamed.
- Execute (x): The file can be executed.

The following is a list of these permissions and their meanings for directories: -

- Read (r): The directory can be read.
- Write (w): The directory can be updated, deleted, and renamed.
- Execute (x): Operations may be performed on the files in the directory. This bit is also called the *search bit,* because execute permission in a directory is not used to indicate whether a directory can be executed or not but to indicate whether you are permitted to search files under the directory.
- Permissions (for owners, groups, and others) are stored in the UNIX system in octal numbers. An octal number is stored in UNIX system using three bits so that each number can vary from 0 through 7. Following is how a octal number is stored:
- Bit 1, value 0 or 1 (defines read permission)

- Bit 2, value 0 or 1 (defines write permission)
- Bit 3, value 0 or 1 (defines execute permission)
- The first bit (read) has a weight of 4, the second bit (write) has a weight of 2, and the third bit (execute) has a weight of 1. For example, a value of 101 will be 5. (The value of binary 101 is (4 * 1) + (0 * 1) + (1 * 1) = 5.)
- Let us now examine how to use the octal number to define the permissions. For example, you might want to define the following permissions for the file testfile in the current directory:
- Owner read, write, and execute
- Group read and execute
- Others--no access at all
- This can be defined as (using binary arithmetic):
- Owner 111 = 7
- Group 101 = 5
- Others 000 = 0
- Thus, the permission of the file testfile is 750.
- Some versions of UNIX provide an additional bit called the *sticky bit* as part of a directory permission. The purpose of the sticky bit is to allow only the owner of the directory, owner of the file, or the root user to delete and rename files.
- The following is a convention for setting up permissions to directories and files. For private information, the permission should be set to 700. Only you will have read, write, and execute permissions on the directory or file.
- If you want to make information public but you want to be the only one who can publish the information, set the permission to 755. Nobody else will have write access, and nobody else will be able to update the file or directory.
- If you do not want the information to be accessed by anybody other than you or your group, set the permission for other 0. The permission may be 770 or 750.
- The following is an example of where you can set up permissions to deny permissions to a particular group.

- Assume that there is a directory called testdir in the current directory owned by a group called outsider.
- If you execute the following command in the current directory, the group outsider will not be able to perform any function on the directory testdir:
- chmod 705 testdir

Default Permissions: umask

- When a user logs into a UNIX system, she is provided with a default permission. All the files and directories the user creates will have the permissions defined in umask.
- You can find out what the default permissions you have by executing the following command:
- umask It might display the following result:
    - 022umask is stored and displayed as a number to be subtracted from 777. 022 means that the default permissions are777 - 022 = 755 That is, the owner can read, write, and execute; the group can read and execute; and all others can also read and execute.
    - The default umask, usually set for all users by the system administrator, may be modified to suit your needs. You can do that by executing the umask command with an argument, which is the mask you want. For example, if you want the default permissions to be owner with read, write, and execute (7); group with read and write (5); and others with only execute (1), umask must be set to 777 - 751 = 026. You would execute the command as follows:
    - umask 026

Changing Permissions: chmod

- You have just seen how the default permissions can be set for files and directories. There might be times when you will want to modify the existing permissions of a file or directory to suit your needs.
- The reason for changing permissions might be that you want to grant or deny access to one or more individuals. This can be done by using the chmod command.
- With the chmod command, you specify the new permissions you want on the file or directory. The new permissions can be specified using one the following two ways:
    - In a three-digit, numeric octal code

  - In symbolic mode

- You are already familiar with the octal mode. If you wanted the file testfile to allow the owner to read, write, and execute; the group to read; and others to execute, you would need to execute the following command:

- chmod 741 testfile

- When using symbolic mode, specify the following:

- Whose (owner, group, or others) permissions you want to change

- What (+ to add, - to subtract, = to equal) operation you want to perform on the permission

- The permission (r, w, x)

- Assuming that the current permission of testfile is 740 (the group has read-only permission), you can execute the following command to modify the permissions of testfile so that the group has write permissions also:

- chmod g+w testfile Another example of symbolic mode is when you want others to have the same permissions as the group for a file called testfile. You can execute the following command:

- chmod o=g testfile Another example of symbolic mode is when you want to modify the permissions of the group as well as the world. You can execute the following command to add write permission for the group and eliminate write permission for the world:

- chmod g+w, o-w testfile

<u>Changing Owner and Group: chown and chgrp</u>

- If you wanted to change the owner of a file or directory, you could use the chown command.

- If the file testfile is owned by user raja, to change the ownership of the file to a user friend, you would need to execute the following command:

  - chown friend testfile

- If you wanted to change the group to which file belongs, you may use the chgrp command.

- The group must be one of the groups to which the owner belongs. That is, the group must be either the primary group or one of the secondary groups of the owner.

- Let us assume that user raja owns the file testfile and the group of the file is staff.

- Also assume that raja belongs to the groups staff and devt. To change the owner of testfile from staff to devt, execute the following command:
  - chgrp devt testfile

Setuid and Setgid

- When you execute some programs, it becomes necessary to assume the identity of a different user or group. It is possible in UNIX to set the SET USER ID(setuid) bit of an executable so that when you execute it, you will assume the identity of the user who owns the executable.
- For example, if you are executing a file called testpgm, which is owned by specialuser, for the duration of the execution of the program you will assume the identity of specialuser. In a similar manner, if SET GROUP ID(setgid) of a executable file is set, executing that file will result in you assuming the identity of the group that owns the file during the duration of execution of the program.
- Here is an example of how the SET USER ID bit is used. Suppose you wanted a backup of all the files in the system to be done by a nightshift operator. T
- his usually is done by the root user. Create a copy of the backup program with the SET USER ID bit set. Then, the nightshift operator can execute this program and assume the identity of the root user during the duration of the backup

Types of Shell In UNIX

- Shell is an integral part of UNIX system. It is responsible for accepting commands from the user, interpreting them and then passing them to kernel for processing. Thus, it is the command processor of the UNIX system. Shell acts as user interface. The main characteristics of shell are under as.
  - Communication between user and UNIX system takes place through the shell.
  - Shell allows back ground processing of time consuming and non interactive tasks.
  - A frequently used sequence of commands can be stored in a file called "shell script". The name of the file can then be used to execute the sequence of commands, automatically.
  - Shell includes features (such as looping and conditional constructs) which allows it to act as a programming language.
  - A user can select a group file for processing with a single command.

- Input of one command can be taken from the output of another command or output of one command can be diverted to the input of a file or printer, using the input output redirection operators (> , < , >>). Pipes (|) can be used to connect simple commands in order to perform complex functions.

The three major shells for UNIX system are:

- Bourne shell.
- C Shell.
- Korn Shell.
- Another shell called Bash shell has become popular these days. Some UNIX versions may provide only one shell, but some may provide many and you can choose the one you like the best.
- **a)** **Bourne Shell:** Bourne shell is one of the oldest among all UNIX shells. It was created by Dr. Steven Bourne at AT&T Bell laboratory. It provides the following features.
- i) Use of wild cards.
- ii) Input and output redirection.
- Iii) A set of shell variables for customizing the shell environment.
- Iv) Back ground execution of commands.
- v) Command set, loop constructs and conditional statements for

  writing shell scripts.

SIMPLE UNIX COMMANDS

- Pwd command:-

  To find out in which directory you are currently working.

  ```
  $pwd     enter
  ```

- the pwd command prints the absolute pathname of your current directory.It takes no. of arguments.e.g the user raja got the result as

  ```
  /usr/raja.
  ```

- Mkdir command:-

To create a new directory, use the mkdir command followed by the directory name e.g. the user wants to create a directory named sales.

```
$mkdir sales
```

- To create a no. of directories with one mkdir command, separate the directory names with spaces. e.g;

  ```
  $mkdir sales/east sales/west

  sales/north sales/south
  ```

  will create four directories under the sales directory.

- Cd command:-

You can change the working directory using the cd command followed by the pathname to which you want to move.e.g; to move to the sales directory, type the following command.

```
$cd sales       enter
```

The cd command without any subsequent pathname always takes the user back to the user and home directory.

- Rmdir command:-

The command rmdir followed by the directory name which is to be renowed is used for deleting a directory.The directory to be deleted must be empty for this command to work.e.g; To remove the “exam” directory

```
$rmdir exam       enter
```

we can also delete more than one directory by separating the multiple directory names with spaces as separators.

- Ls command:-

To list the names of files and sub-directories of the current directory,ls command is used.its function is equivalent to the dir command in ms-dos.e.g. type the list contents of current directory,user “raja” should type the following.

```
$ls
```

he will get the following result.

```
East west north
```

- If you want to give more than one options along with the ls command, then you can use either of the following ways

```
$ls-l-a-t

$ls-lat
```

Both methods will work.

Option purpose

- -x Get multi-columner o/p.
- -r List files and sub-directories in reverse order.
- -t List files and sub-directories with time order.
- -a List all files ,including the normal hidden files.
- -A List all files excluding and …
- -c List files by mode modification time.
- -i List the inode for each file.
- -l Display permissions,owner size,modification time etc along with file and directory names.
- -h Display names of hidden files.
- -q Display file names having non-printable characters.

File commands

- Cat command:-To create a file, type the command cat at the shell prompt, followed by a > character and the filename e.g; To create a file called January under the current directory.

```
$cat>January          enter
```

After you press enter key, you will be prompted to enter the contents of the file type the required data and press cntl+D key to terminate the cmd line.

- Transferring contents from one file to another:-

You can also pass the contents of a file to another file.e.g; if you wish that6 the contents of January should also be there in another file called jan.

```
$cat January> jan
```

- If the file jan already exists,it will get overwritten. If you want to append the data to the file and not overwritten it,use double right chevron.

```
$cat January >> jan         enter.
```

- Listening the contents of files:-

To list the contents of a file,use the cat cmd followed by the file name.e.g; to list the contents of file January.

```
$cat January      enter
```

The cat cmd can also display the contents of more than one file by separating the different filenames with spaces.

```
$cat January febuary       enter
```

The above cmd will display the contents of two files in different rows.

- <u>Cp command</u>:-

The cmd cp is used to copy a file into another. It requires two arguments, a source filename,the contents of which are to be copied, and a target filename, to which the contents are to be copied. E.g. to copy the file January into another file, say march.

```
$cp January march       enter
```

Use cp-I option if you want the system to ask for configuration before copying the files.

- <u>Rm cmd</u>:-

To delete or remove a file, the rm cmd is used e.g; to remove the file march.

```
$ rm march
```

Use rm with –i option, if you want the system to ask for configuration, like in use of cp cmd.

To remove a directory, use rm with –r option. Here –r stands recursive deletion.e.g. the following cmd will remove the directory accounts.

```
$rm-r accounts       enter
```

Unlike rmdir,rm-r will remove directories even if they are not empty.

- <u>Mv(moving files)</u>:-

A file can be shifted from one directory to another by using the mv cmd.The following cmd will displace the directory north from “sales” directory to the directory “/usr/raja”.

```
$mv north/usr/raja/north    e   enter
```

The above cmd doesn’t create a copy of north but displays it.

- Changing permissions:-

Typically, users on a unix system are open with the files.The usual set of permissions given to files is rw-r—r--,which let other users read the file but not change it in any way.

In order to change these default permissions and set your own,chmod cmd is used.Only owner of the file can execute the chmod cmd system.

Filename(s)

The argument category specifies the user’s class. The argument operation specifies whether the permission is to granted or denied . The argument permission specifies the type of permission

- I/O redirection operator (<) wc. Cmd:-

The cmd wc counts the no. of characters, words and lines in a text. We can use i/p redirection operator to take i/p from a file.

```
$wc < myfile         enter
```

will count & display the no. of lines, words and characters in the file myfile.

- Kill command:-

In order to abort a process, kill cmd is used. Kill cmd is followed by the PID of the process which you want to abort.

e.g; to kill the process sleep.

```
$kill 1369      enter
```

1369 is the PID of the process sleep. You can also kill more than one process,by separating the PID’s with spaces.using a single kill cmd.

-Some p-rocesses are hard to kill.

- To forcibly kill such such processes,use the cmd kill-9 followed by the PID of the process to be killed.

- Cal command:-
  - The cmd cal displays the current date & time.
  - If we want to refer to a calendar i.e., the time when we use the cal cmd.
  - It is capable of printing calendar for any year in the range 1 to 9999. to invoke it all that we have to do is type cal.

```
$cal
```

- Touch command:-

➢ This command not only create empty files, but it also allows you to change the modifications and access times of a file.

➢ When you run a cat cmd on a file you are accessing it,when you are making chages in a file you are modifying it, whereas

➢ When you are preparing a new file afresh you are creating it. Unix keeps track of times at which each of these activities are performed for every single file on the file system.

➢ Touch comes into picture when you want to change these times without really accessing or modifying the file.e.g;

```
$touch-a myfile
```

➢ would change the access time of myfile to whatever is the current time.

➢ We can also set the access time for a file to a particular time instead of current time.

e.g;

```
$touch-a 0425120596 story
```

➢ This would set the access time of story to the specified time.

➢ The unintelligible looking series of digits is the supplied time. Read it as two digits each for month,day,hour,minutes and year.

- Sort command:-

➢ As the name suggests the sort command can be used for sorting the contents of the file.apart from sorting files,sort has another trick up its sleev.

- It can merge multiple sorted files and store the result in the specified o/p file.While sorting the sort command basis its comparisons on the first character in each line in the file.

- If the first character of two lines is same then the second character in each line is compared and so on.

- To put it in more technical terms, the sorting is done accordingly to the ASCII collection sequence i.e. it sorts the spaces and the tabs first, then the punctuation marks followed by numbers, uppercase letters and lowercase letters in that order.

- The simplest form of sort command will be:

```
$sort myfile
```

- This would sort the contents of myfile and displays the sorted o/p on the screen.

- If we want we can sort the contents of several files at one short as in:

```
$sort file1 file2 file3
```

- Grep command:-

- Grep is an acronym for 'globally search a regular expression & print it'.

- The cmd searches the specified i/p fully (globally) for a watch with the supplied pattern & displays it.

- While forming the patterns to be searched , we can use shell metacharacters, or regular expressions , as professional unix users call them.

- Knowing the versatility of the metacharacters , what powers they yield to grep can easily be imagined.

- Added to that is is its capability to search in more than one file, enhanced by the use of various options or switches.

- The simplest example of the grep cmd is as follows:

```
$ grep picture newsfile
```

- This would search the word 'picture' in the file newsfile & if found, the lines containing it would be displayed on the screen.

We can use grep to search a pattern in several files.e.g;

```
$ grep picture newsfile storyfile
```

- Here the word ‘picture’ would be searched in both the files, newsfile & storyfile & if found, the lines containing it would be displayed along with the name of the file where it occurred.

- **<u>Mail command</u>**:-

Unix supports electronic-mailing by providing several mail programs. One such mail program is mail.

- Syntax:

- mail [options] [users]

- When mail cmd is invoked along with a user-id, the system waits for some message to be typed in & sends this message to the specified user-id.

- **<u>Positional parameters</u>**:-

We can provide argument to a simple shell program by specifying them on a command line.

When you execute a shell program, shell variables are automatically set to match the command line argument given to your program.

These variables are referred as positional parameters & allow your program to access & manipulate command line information.

- e.g;

cmd	arg 1	arg 2	arg n
$0	$1	$2	$n

- The parameters $# is used to count the number of arguments given by the user except the the $0 i.e., program name.

- S* the S* is used to display the rest of arguments in the command name.

- $? It is used to check the status of the last executed command. If the $0 value is 0 then last command is successfully executed else not executed.

- **<u>Shifting positional parameters</u>**:-

- All command line arguments are assigned by the shell to the position parameters.

- The value of the first command parameters line argument is contained into $1.

- The value of the 2nd in $2, the 3rd in $3 and so forth.
- We can reorder positional parameters by assigning them to variables and then using the built in command shift which renames the parameters.
- The value of $2 is shifted to $1 and $3 is shifted to $2 and so on. The original value of $1 is lost.
- Clear
- Set “who”
- Echo 1
- Echo 2
- Echo 3
- Echo 4
- Echo 5
- Shift
- Echo 1
- Echo 2
- Echo 3
- Echo 4
- Echo 5
- Echo command:-

➢ Is the simplest command that allows you to write o/p from a shell program.

➢ Echo writes its arguments to a standard o/p.

➢ you can use echo directly, as a regular command or as a component of a shell script.

➢ The following examples shows how vit works.

```
$ echo “hello”

o/p        hello
```

or x=10

$ echo $x

it displays the value of x

o/p 10

- **Read command**:-

➢ The read command lets you insert the user i/p into your script.

➢ Read reads one line from more shell variables. You can also use the read command to assign a several shell variables at once.

➢ When you use read with several variable names, the first field typed by the user is assigned by the first variable, the second field to the second variable & so on.

- The case command:-

➢ If you wish to make a comparison of some variables against the entire series of possible values, you can use nested if…..then or if…..then……elif…..else statements. However, the case cmd provides a simpler & more readable way to make the same comparison.

➢ It also provides a convenient & powerful way to compare a variable to a pattern, rather than to a single specific value.

➢ Case command

```
case string

in

pattern-list)

            command line

            command line

                  -     -

               ;       ;
```

- **case**

Case operators as follows:

- The value of string is compared in turn each of the pattern.
- If a match is found, the commands follows the patterns is executed up until the double semicolon(; ;) at which point the case statement terminates.
- If the value of string doesn't match any of the pattern, the program goes through the entire case statements.

```
  Clear

echo “enter your choice”

read num

   case $ num in

1)     echo “1 Hello”; ;

2)     echo “2 Bye”; ;
```

- **case**

Looping:-

The loop or an iteration constructs directly program to perform set of operations again and again, until a specified condition is achieved.

The condition causes the termination of loop.

Shell provides the following two commonly used loop constructs:

- for
- while
- **For**:-
- For is most frequently used loop constructs in shell programming. The shell for construct
- is different from its c-programming language counterpart. It takes the following form:

```
for variable in [list]

do

        commands

done
```

- The [list] specifies a list of values which the variable can take. The loop is then executed for each value mentioned in the list e.g; the following will display three planets of solar system.

```
For planet in mercury venus earth

do

        echo $ planet

done
```

- All the instructions between do and done are executed using the first value in the list.
- These instructions are then repeated for the 2nd value.
- The process is repeated until all the values in the list gets executed.
- **While**:-
- While loop constructs contains the condition first.
- If the condition is satisfied, the control executes the statements following the while loop, else it ignores these statements.
- The general form of while loop is

```
while[condition]

        do

                command

        done
```

- The commands between do and done are executed as long as the condition holds true.e.g. the following will print the the values 1 to 10 in reverse order.

```
i=0

while[$i-gt0]

do

echo $i

i='expr $i-1'

done
```

- i is initialized to 10. The commands within the do and done block will print a values of i and then decrement it as long as value in I is greater then 0.
- **Break statements**:-
- Normally, when you setup a loop using the for, while and until commands and the select command in SEL.
- Execution of the commands enclosed in the loop continuous until the logical loop is met.
- The shell provides two ways to alter the operation of command in a loop.
- Break exists from immediately enclosing loop. If you give break a numerical argument , the program breaks out of that number of loops.
- In a set of nested loops, break breaks out of the immediately enclosing loop and the next enclosing loop. The command immediately following the loop are executed.
- e.g;

```
i=5

while[$i-le5]

do

echo $i

if[$i-le0]

then

echo"0 or negative value found "

break

fi

i='expr $i-1'

done
```

- **Continue statement**:-
- Is the opposite of break control goes back to the top of the smallest enclosing loop.
- If an argument is given e.g; continue 2, control goes to the top of the nth 2nd,enclosing loop.