SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

Common to: CSE, IT

UNIT - I

UNIT 1 INTRODUCTION **8 Hrs.**

Introduction - Operating system structures - System components - OS services - System calls - System structure - Resources Processes - Threads - Objects - Device management - Different approaches - Buffering device drivers.

UNIT – I- SCS1301- OPERATING SYSTEM

I. Introduction

A computer system has many resources (hardware and software), which may be require to complete a task. Operating System is a system software that acts as an intermediary between a user and Computer Hardware to enable convenient usage of the system and efficient utilization of resources. The commonly required resources are input/output devices, memory, file storage space, CPU etc. Also an operating system is a program designed to run other programs on a computer.

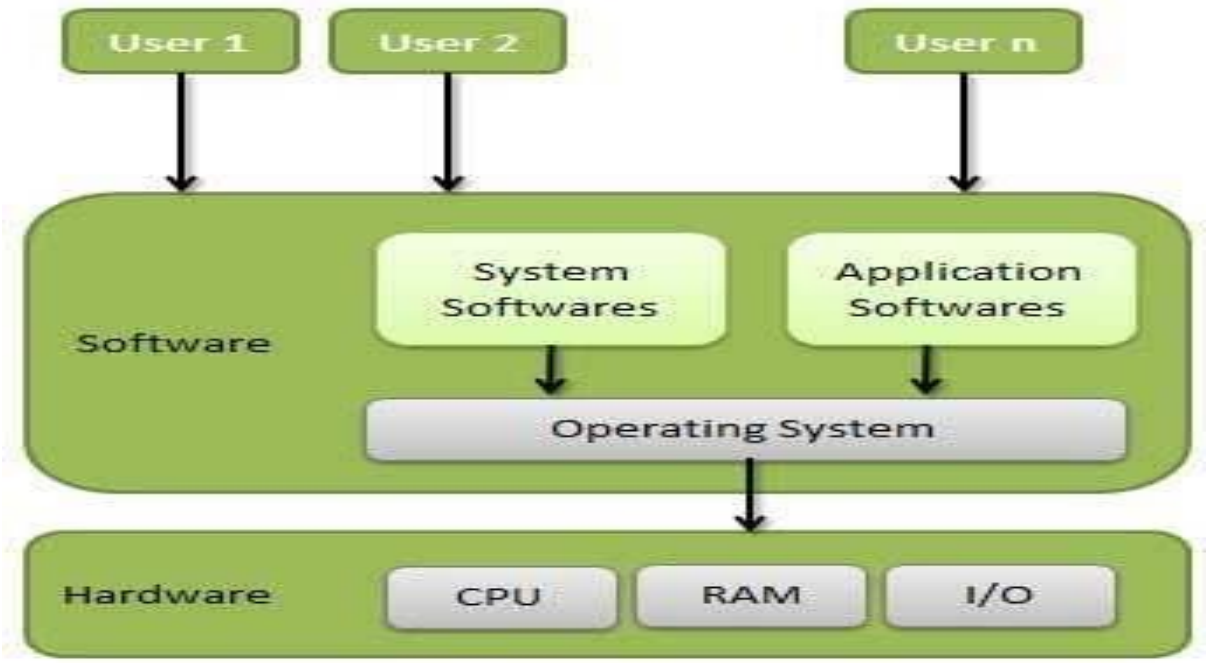


Fig. 1.1 Abstract View of the Components of a Computer System.

OS is considered as the backbone of a computer, managing both software and hardware resources. They are responsible for everything from the control and allocation of memory to recognizing input from external devices and transmitting output to computer displays. They also manage files on computer hard drives and control peripherals, like printers and scanners. Operating systems monitor different programs and users, making sure everything runs smoothly, without interference, despite the fact that numerous devices and programs are used simultaneously. An operating system also has a vital role to play in security. Its job includes preventing unauthorized users from accessing the computer system.

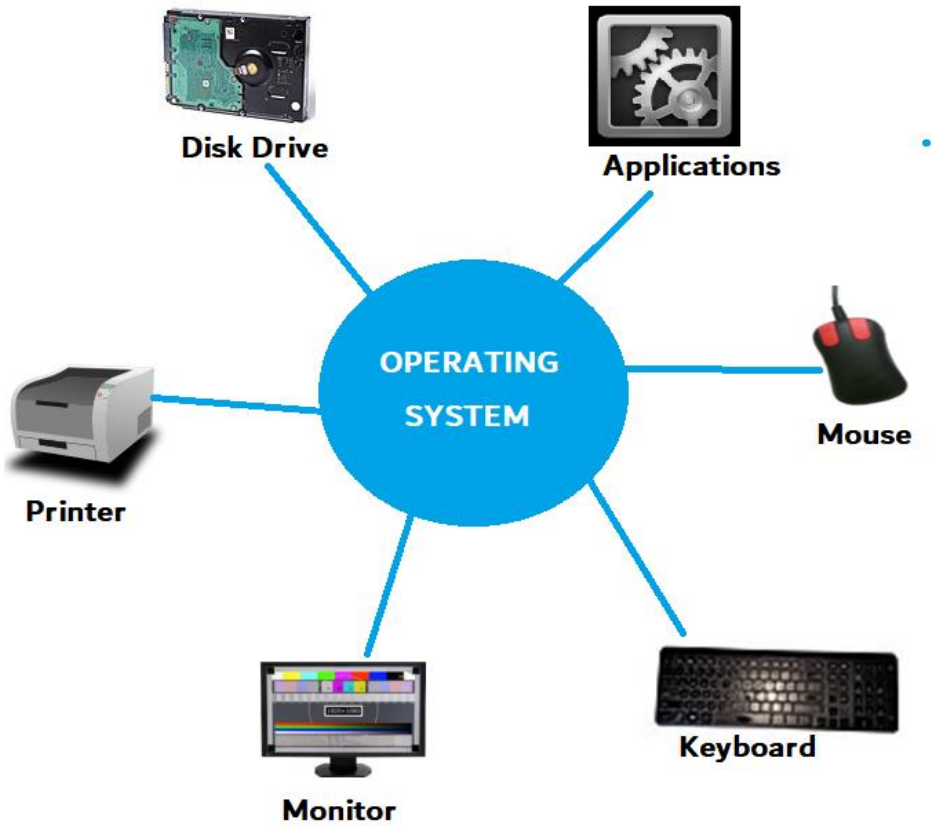


Fig. 1.2 Interaction of Operating System

Goals of the Operating System

The following are the goals of the operating system:

1. To execute user programs
2. Make solving user problems easier.
3. Make the computer system convenient to use.
4. Use the computer hardware in an efficient manner.

Classification of Operating Systems

1. Multi-user OS

Allows two or more users to run programs at the same time. This type of operating system may be used for just a few people or hundreds of them. In fact, there are some operating systems that permit hundreds or even thousands of concurrent users.

2. Multiprocessing OS

Support a program to run on more than one central processing unit (CPU) at a time. This can come in very handy in some work environments, at schools, and even for some home-computing situations.

3. Multitasking OS

It allows to run more than one program at a time.

4. Multithreading OS

It allows different parts of a single program to run concurrently (simultaneously or at the same time).

5. Real time OS

These are designed to allow computers to process and respond to input instantly. Usually, general purpose operating systems, such as disk operating system (DOS), are not considered real time, as they may require seconds or minutes to respond to input. Real-time operating systems are typically used when computers must react to the consistent input of information without delay. For example, real-time operating systems may be used in navigation. General-purpose operating systems, such as DOS and UNIX, are not real-time. Today's operating systems tend to have graphical user interfaces (GUIs) that employ pointing devices for input. A mouse is an example of such a pointing device, as is a stylus. Commonly used operating systems for IBM-compatible personal computers include Microsoft Windows, Linux, and Unix variations. For Macintosh computers, Mac OS X, Linux, BSD, and some Windows variants are commonly used.

2. Operating System Structures

An OS provides the environment within which programs are executed. Internally, Operating Systems vary greatly in their makeup, being organized along many different lines. The design of a new OS is a major task. The goals of the system must be well defined before the design begins. The type of system desired is the basis for choices among various algorithms and strategies.

An OS may be viewed from several vantage ways:

- By examining the services that it provides.
- By looking at the interface that it makes available to users and programmers.
- By disassembling the system into its components and their interconnections.

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. As modern operating systems are large and complex careful engineering is required. There are four different structures that have shown in this document in order to get some idea of he spectrum of possibilities. These are by no means exhaustive, but they give an idea of some

designs that have been tried in practice.

A SIMPLE STRUCTURE:

Many commercial systems do not have well-defined structures. Frequently,such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. These started as small systems and rapidly expanded much further than their scope. A common example of this is MS-DOS.

MS – DOS SYSTEM STRUCTURE:

MS-DOS is written to provide the most functionality in the least space. It is divided into modules. Although MS-DOS has some structure,its interfaces and levels of functionality are not well-separated.

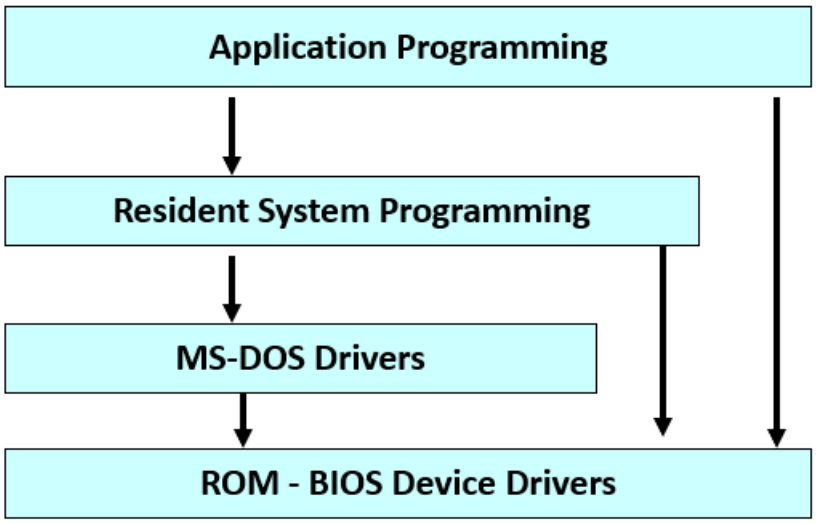


Fig. 1.3 MS-DOS Structure

LAYERED STRUCTURE:

A system can be made modular in many ways. One way to achieve modularity in the operating system is the layered approach in which the operating system is broken up into a number of layers(or levels), each built on top of lower layers. In this, the bottom layer is the hardware and the topmost layer is the user interface. The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface as shown in below. With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

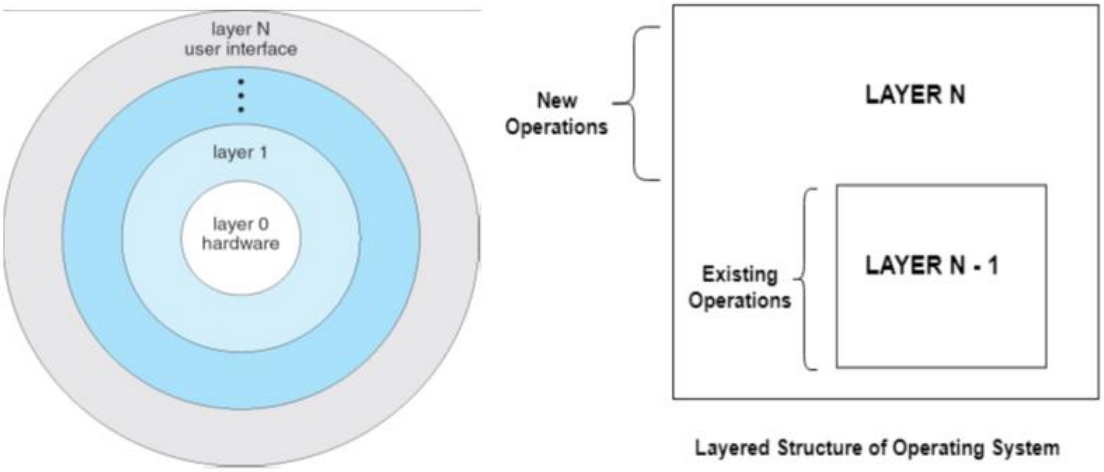


Fig. 1.4 Layered Structure

A layered design was first used in the operating system. Its six layers are as follows:

layer 5: user programs

layer 4: buffering for input and output

layer 3: operator-console device driver

layer 2: memory management

layer 1: CPU scheduling

layer 0: Hardware

An operating-system layer is an implementation of an abstract object made up of data, and of the operations that can manipulate those data. A typical operating-system layer—say, layer M consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. The main advantage of the layered approach is modularity (simplicity of construction and debugging). The layers are selected so that each uses functions (operations) and services of only lower-level layers.

This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.

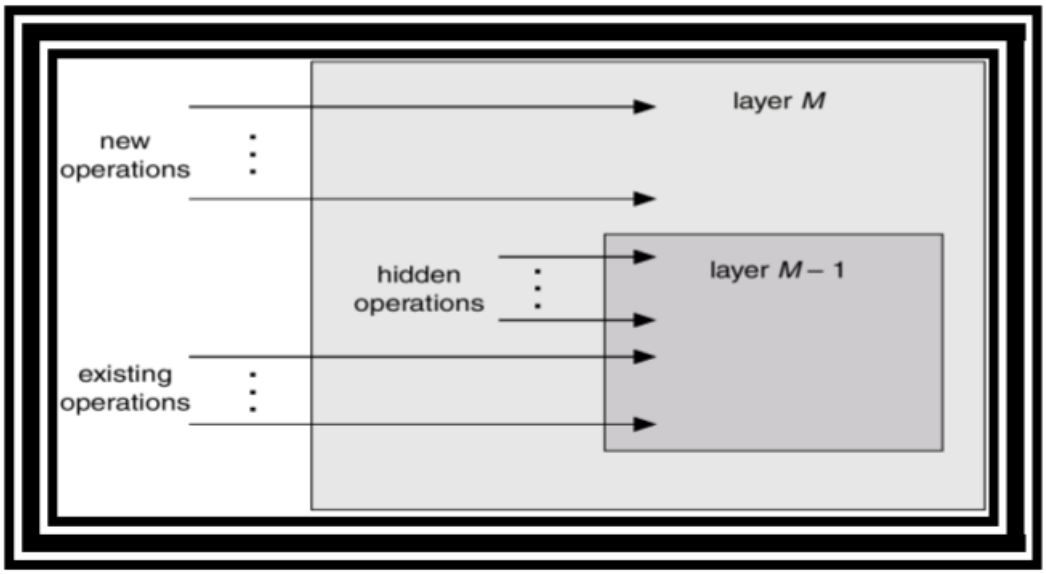


Fig. 1.5 An Operating System Layer

Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified when the system is broken down into layers.

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

Drawbacks

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the disk space used by virtual-memory algorithms must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the disk space.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may

need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non-layered system. These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of laver definition and interaction. For instance, OS/2 shown in fig(f) is a descendent of MS-DOS that adds multitasking and dual mode operation, as well as other new features.

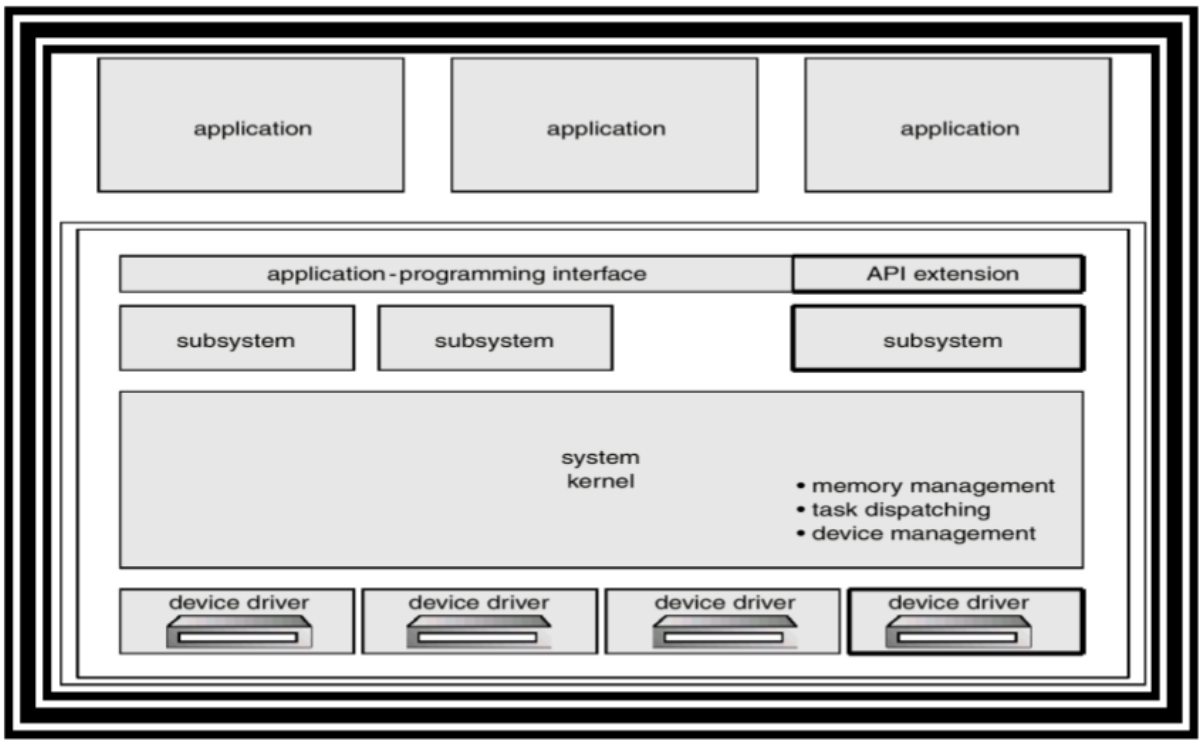


Fig. 1.6 OS/2 Layer Structure

Microkernels

We have already seen, as the UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs i.e., moves as much from the kernel into "user" space which results is a smaller kernel.

There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. However, microkernels typically provide minimal process and memory management, in addition to a communication facility. The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.

Communication takes place between user modules using message passing. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched. Several contemporary operating systems have used the microkernel approach.

Examples

• Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel.

• QNX is a real-time operating system that is also based on the microkernel design. The QNX microkernel provides services for message passing and process scheduling.

Benefits

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Drawback

Microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. However, this version delivered low performance compared with that of Windows 95. Windows NT 4.0 partially redressed the performance problem by moving

layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, its architecture was more monolithic than microkernel.

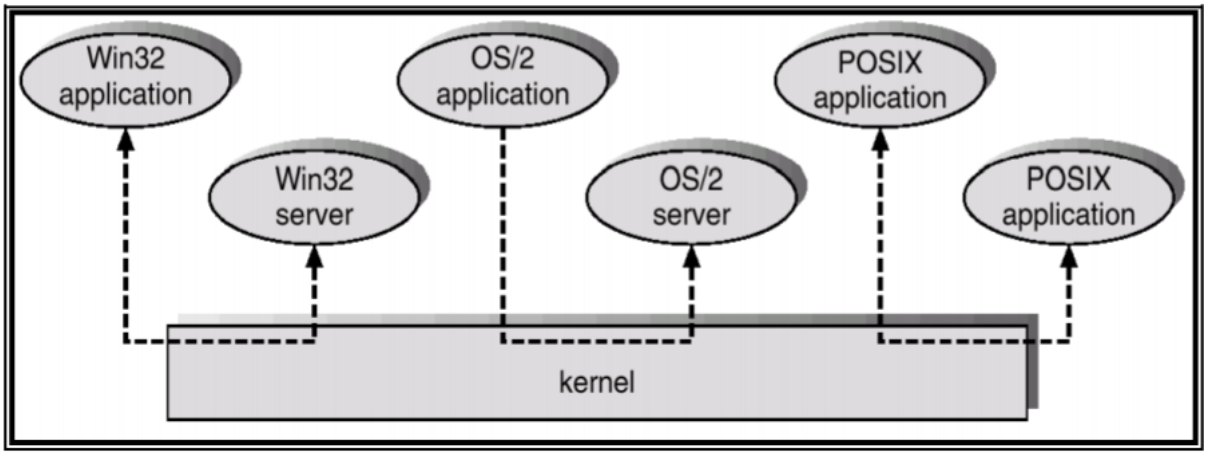


Fig. 1.7 Windows NT Client-Server Structure

UNIX STRUCTURE:

UNIX is limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS Consists of two separable parts.

1. Systems Programs
2. The kernel

• The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.

• Everything below the system call interface and above the physical hardware is the kernel.

• The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

• Taken in sum, that is an enormous amount of functionality to be combined into one level.

This monolithic structure of UNIX was difficult to implement and maintain i.e., hanges in one system could adversely effect other areas.

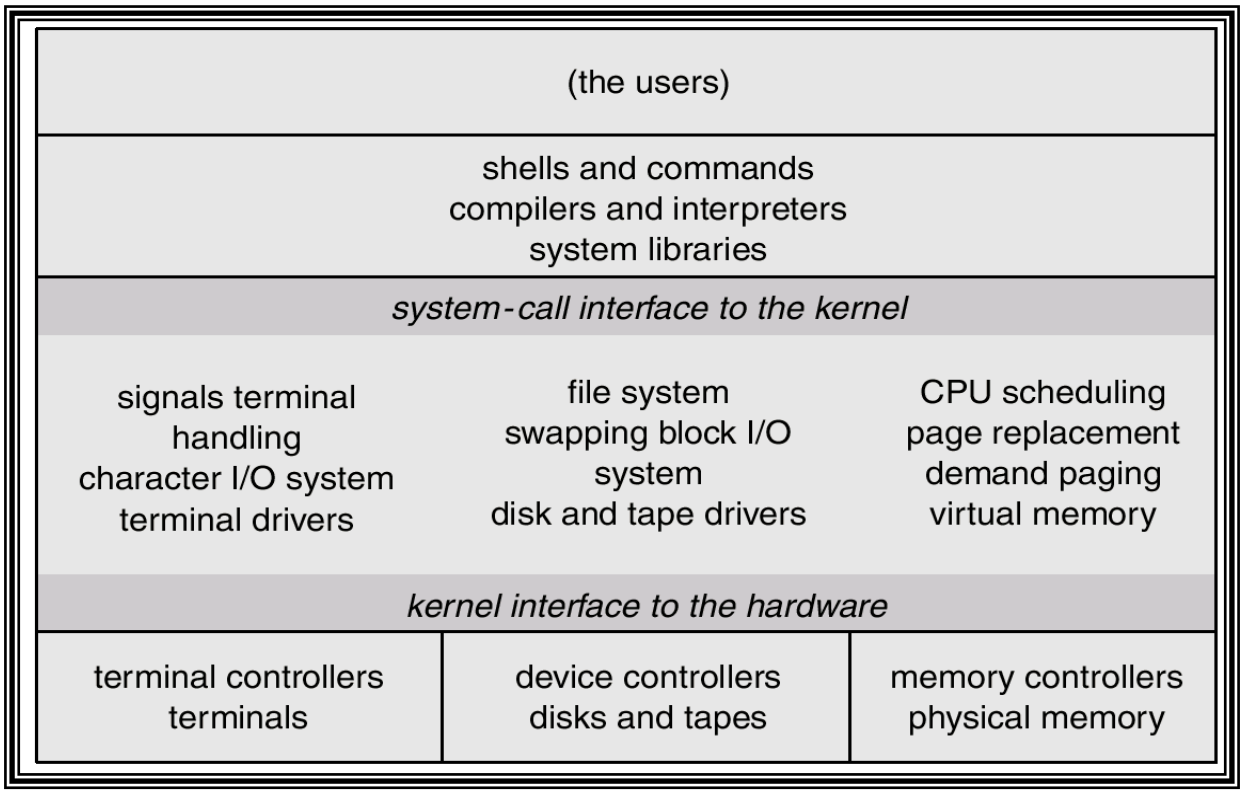


Fig. 1.8 UNIX Structure

New versions of UNIX are designed to use more advanced hardware. With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and are separated into components. This seperation allows programmers to hide information, they are therefore free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

VIRTUAL MACHINE:

In a Virtual Machine - each process "seems" to execute on its own processor with its own memory, devices, etc. The resources of the physical machine are shared. Virtual devices are sliced out of the physical ones. Virtual disks are subsets of physical ones.

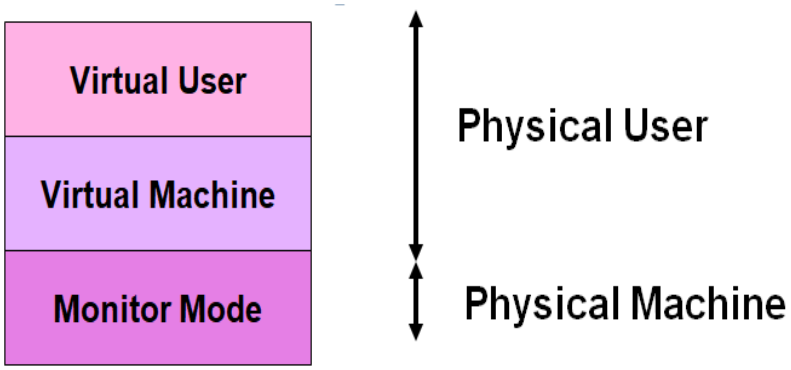


Fig. 1.9 Demarcation as Physical User and Physical Machine

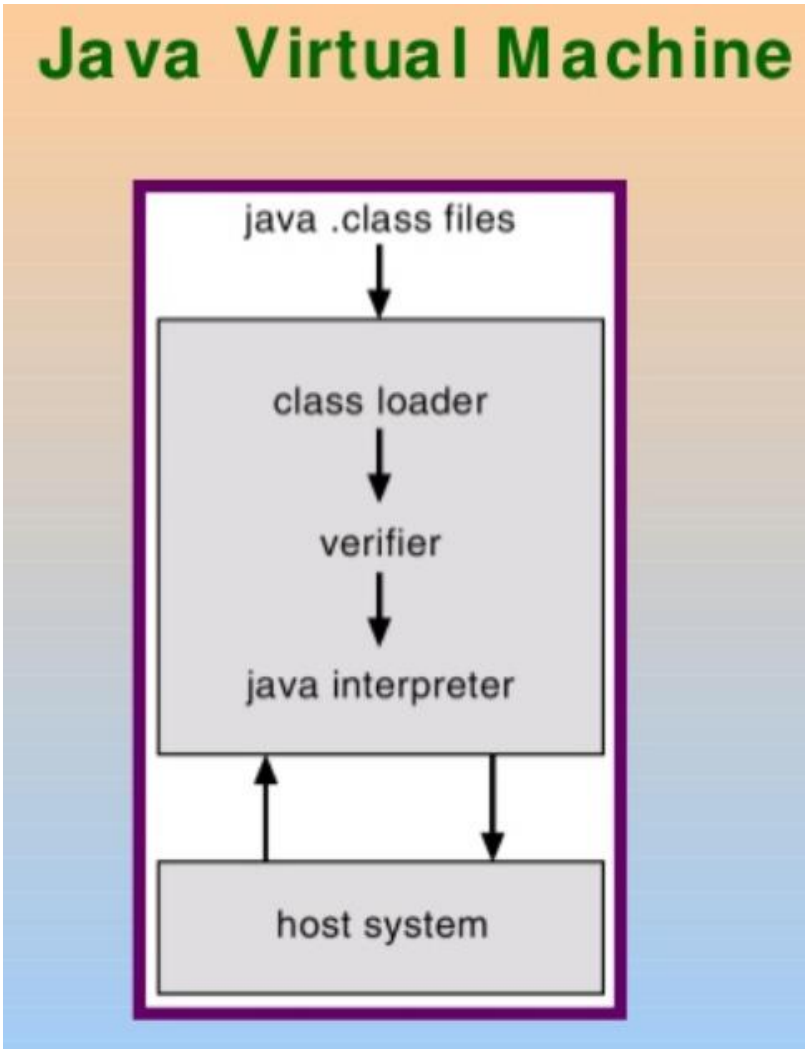


Fig. 1.10 Java Virtual Machine

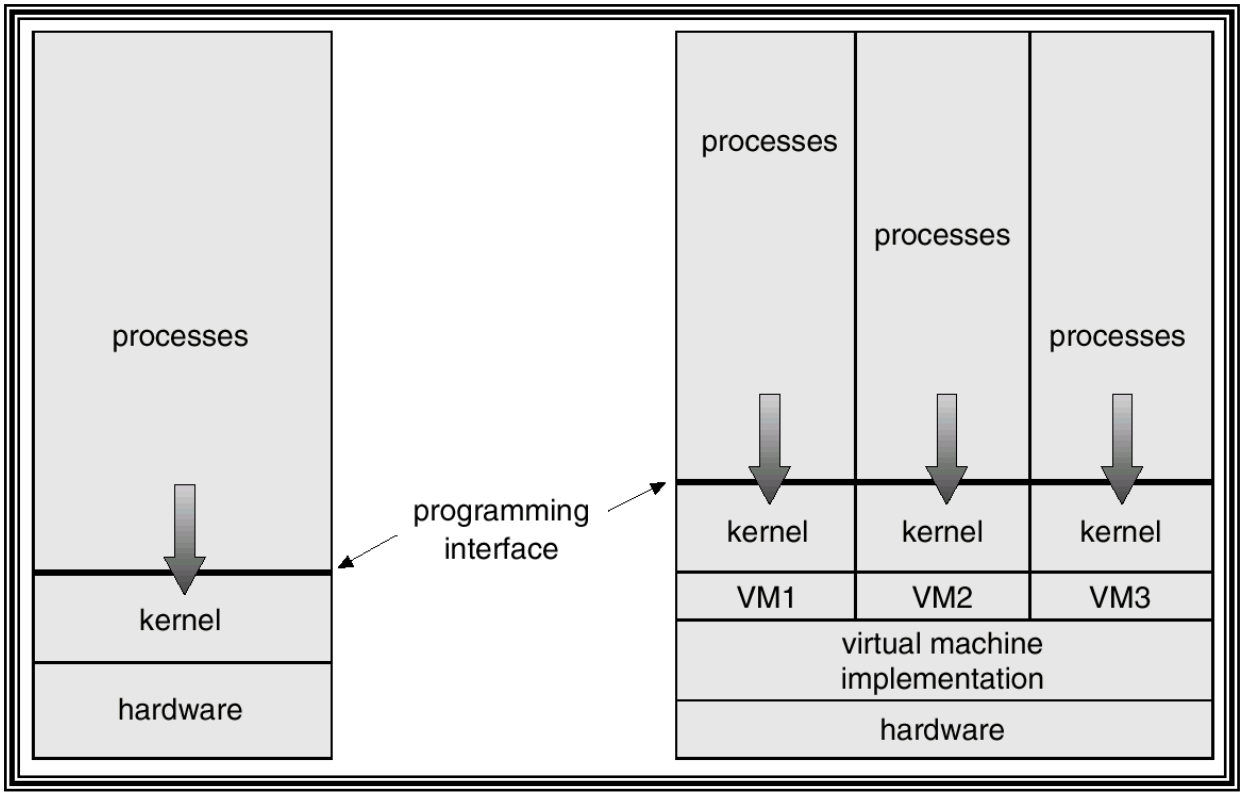


Fig. 1.11 Process-Level View of Virtual Machine

3. System Components

We can create a system as large and complex as an operating system by partitioning it into smaller pieces. Each piece should be a well-delineated (represented accurately or precisely) portion of the system with carefully defined inputs, outputs and functions. Even though, not all systems have the same structure. However, many modern operating systems share the same goal of supporting the following types of system components:

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

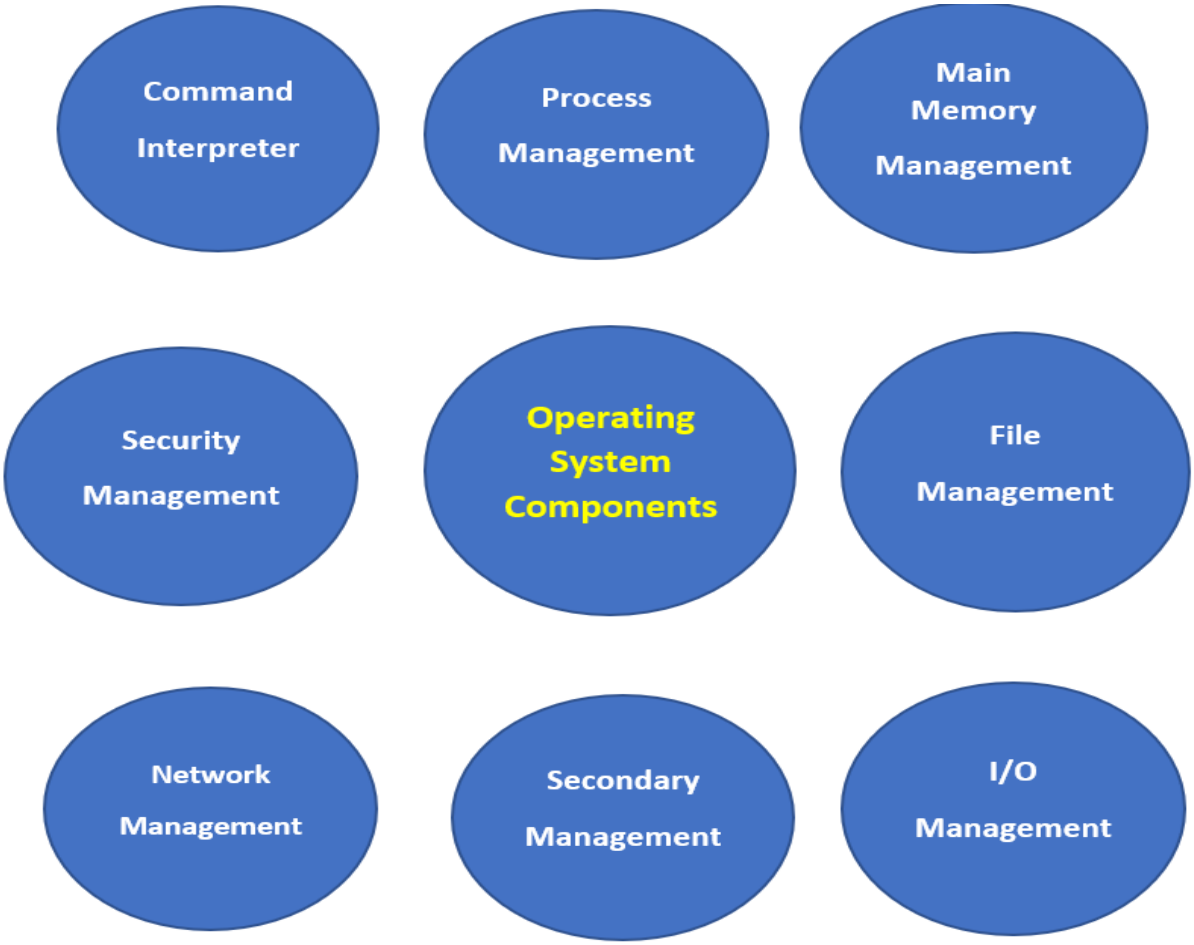


Fig. 1.12 Components of Operating System

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A program by itself is not a process but a program in execution. For example

- A batch job is a process
- A time-shared user program is a process
- A system task (e.g. spooling output to printer) is a process.

There can be many processes running the same program. A program does nothing unless its instructions are executed by a CPU. The execution of a process must be sequential. The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.
- Suspension and resumption (Block/Unblock) of processes.
- Providing mechanism for process Synchronization.
- Providing mechanism for process Communication.
- Providing mechanism for process deadlock handling.

Main Memory Management

Introduction

Main memory is central to the operation of a modern computer system. Primary-Memory or MainMemory is a large array of words or bytes ranging in size from hundreds of thousands to billion. Each word or byte has its own address. Main-memory provides storage that can be accessed directly by the CPU. The main memory is only large storage device that the CPU is able to address and access directly for a program to be executed, that must in the main memory. To improve both the utilisation of the CPU and the speed of the computer's response to its users, we must keep several programs in memory.

The major activities of an operating system in regard to memory-management are:

- Monitoring which part of memory are currently being used and by whom.
- Deciding which process are loaded into memory when memory space becomes available.
- Allocating and deallocating memory space as needed.

File Management

File management is one of the most visible components of an OS. Computers can store information on several different types of physical media (e.g. magnetic tap, magnetic disk, CD etc). Each of these media has its own properties like speed, capacity, data transfer rate and access methods. For convenient use of the computer system, the OS provides a uniform logical view of information storage.

A file is a logical storage unit, which abstracts away the physical properties of its storage device. A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. The operating system is responsible for the following activities in connection with file management:

- Creation and deletion of files.
- Creation and deletion of directions.
- Support of primitives for manipulating files and directions.
- Mapping of files onto secondary storage.
- Backing up of files on stable (non volatile) storage media.

I/O System Management

OS hides the peculiarities of specific hardware devices from the user. I/O subsystem consists of:

- A memory management component that includes buffering, caching and spooling.
- A general device-driver interface
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of the specific device to which it is assigned.

Secondary-Storage Management

The main purpose of a computer system is to execute programs. These programs, with the data they access, must be in main memory, or primary storage. Systems have several levels of storage, including primary storage, secondary storage and cache storage. Since main memory (primary storage) is volatile and too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the principle on-line storage medium, for both programs and data. The operating system is responsible for the following activities in connection with disk management:

- Free-space management (paging/swapping)
- Storage allocation (what data goes where on the disk)
- Disk scheduling (Scheduling the requests for memory access).

Networking

A distributed systems is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines such as network or high-speed buses. The processors in a distributed system vary in size and function. They may include small processors, workstations, minicomputers and large, general-purpose computer systems. The processors in the system are connected through a communication-network, which are configured in a number of different ways i.e. Communication takes place using a protocol. The network may be fully or partially connected . The communication-network design must consider routing and connection strategies, and the problems of contention and security. A distributed system provides user access to various system resources. Access to a shared resource allows:

- Computation Speed-up
- Increased functionality
- Increased data availability
- Enhanced reliability

Protection System

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, files, memory segments, processes(CPU) only by the users who have gained proper authorization from the OS.

The protection mechanism must:

- Distinguish between authorized and unauthorized usage.
- Specify the controls to be imposed.
- Provide a means of enforcement.

Command Interpreter System:

A command interpreter is one of the important system programs for an OS. It is an interface of the operating system with the user. The user gives commands, which are executed by Operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Many commands are given to the operating system by control statements which deal with:

- process creation and management
- I/O handling
- secondary-storage management
- main-memory management
- file-system access
- protection
- networking

4. Operating system services

An OS provides environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided differs from one OS to another, but we can identify common classes. These OS services are provided for the convenience of the programmer, to make the programming task easier. One set of operating-system services provides functions that are helpful to the user are as follows:

1. Program Execution
2. I/O Operations
3. File System Manipulation
4. Communication
5. Error Detection
6. Resource Allocation
7. Accounting
8. Protection & Security

Program Execution:

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O Operations:

A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (rewind a tape drive, or to blank a CRT). For efficiency and protection, users usually cannot execute I/O operations directly. Therefore Operating system must provide some means to perform I/O.

File System Manipulation

The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Communication:

One process needs to exchange information with another process. Such communication can occur in two ways: The first takes place between processes that are executing on the same computer. The second takes place between processes that are executing on different computers over a network. • Communications may be implemented via shared memory or through message passing, in which packets of information moved between the processes by the OS.

Error Detection:

OS needs to be constantly aware of possible errors. Errors may occur

1. In the CPU and memory hardware (such as a memory error or power failure)
2. In I/O devices (such as a parity error on tape, a connection failure on a network or lack of power in the printer).
3. In user program (such as arithmetic overflow, an attempt to access illegal memory location, or a too-great use of CPU time).

For each type of error, OS should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system. Another set of OS functions exists not for helping user, but for ensuring

the efficient operation of the system itself via resource sharing. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource Allocation

When multiple users logged on the system or multiple jobs running at the same time, resources must be allocated to each of them. Many types of resources are managed by OS. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have general request and release code.

Accounting

To keep track of which users use how much and what kinds of computer resources. This record may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

Protection & Security:

The owners of information stored in a multi-user or networked computer system may want to control the use of that information. When several disjointed processes execute concurrently, processes should not interfere with each other or with the OS itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts. If a system is to be protected and secure, precautions must be instituted throughout. A chain is only as strong as its weakest link.

5. System Calls

System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions. Some systems also allow to make system calls from a high level language, such as C, C++ and Perl (have been defined to replace assembly language for systems programming). As an example of how system calls are used,consider writing a simple program to read data from one file and to copy them to another file. The first input that the program will need is the names of the two files:

- The input file and
- The output file

These names can be specified in many ways,depending on the OS design:

One approach is for the program to ask the user for the names of the two files.

I. In an **interactive system**, this approach will require a sequence of system calls, first to write a prompting message on the screen, and then to read from the keyboard the charaters that define the two files.

II. **On mouse-based window-and-menu systems**, a menu to select the source name and a similar window can be opened to select the destination name.

Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call and may encounter possible error conditions. When the program tries to open the file, it may find that no file of that name exists or that the file is protected against access. In these cases, the program should print a message on the console (anotther sequence of system calls), and then terminate abnormally(another system call).

• If the input file exists, then we must create a new output file.

o We may find an output file with the same name.

This situation may cause the program to abort (a system call), or

we may delete the existing file (another system call).

o In an interactive system another option is to ask the user (a sequence of system calls to output the prompting message and to read response from the keyboard) whether to replace the existing file or to abort the program.

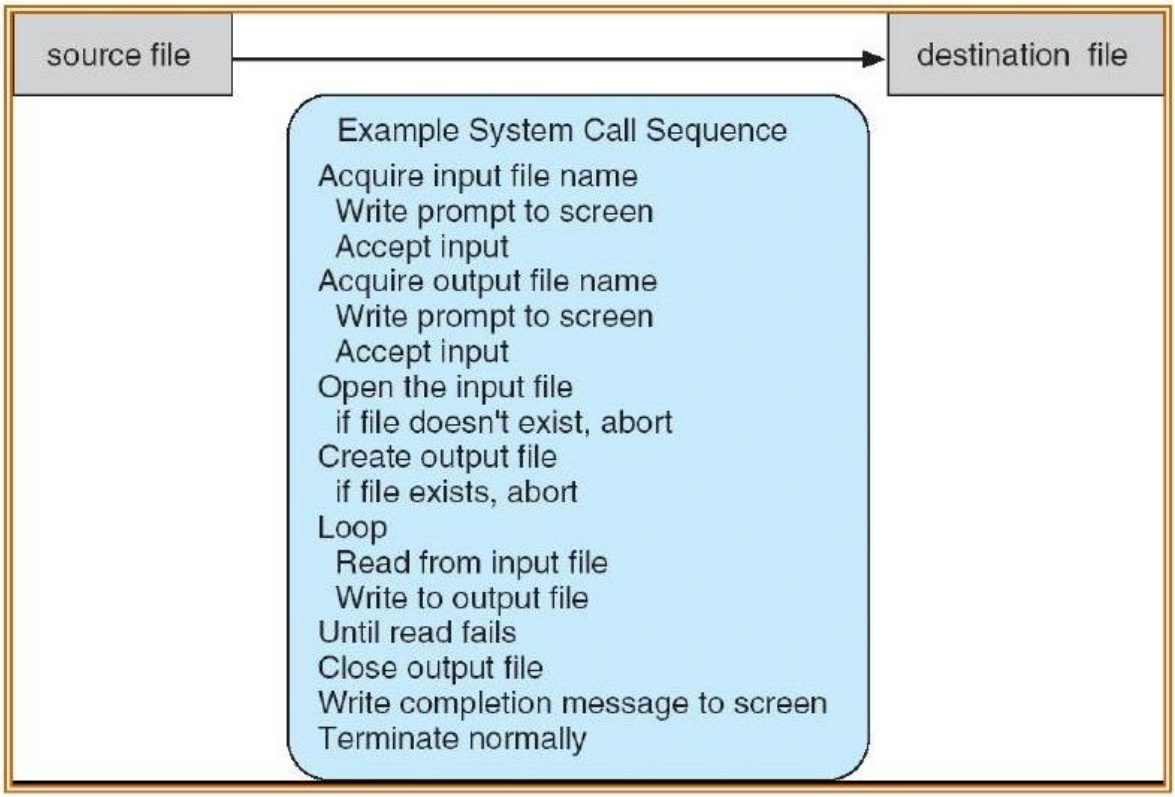


Fig. 1.13 System Call for Copying a File

Now that both the files are setup, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).

• Each read and write must return status information regarding various possible error conditions.

o On input,

the program may find that the end of file has been reached, or

that a hardware failure occured in the read (such as a parity error).

o On output,

Various errors may occur, depending on the output device (such as no more disk space, physical end of tape, printer out of paper).

Finally, after the entire file is copied,

o The program may close both files (another system call), writes a message to the console(more sytem calls),and finally terminates normall (the final system call).

As we can see, even simple programs may make heavy use of the OS.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular OS. Three general methods are used to pass parameters between a running program and the operating system.

− Simplest approach is to pass parameters in registers.

− Store the parameters in a table in memory, and the table address is passed as a parameter in a register (in the cases where parameters are more than registers).

− Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.

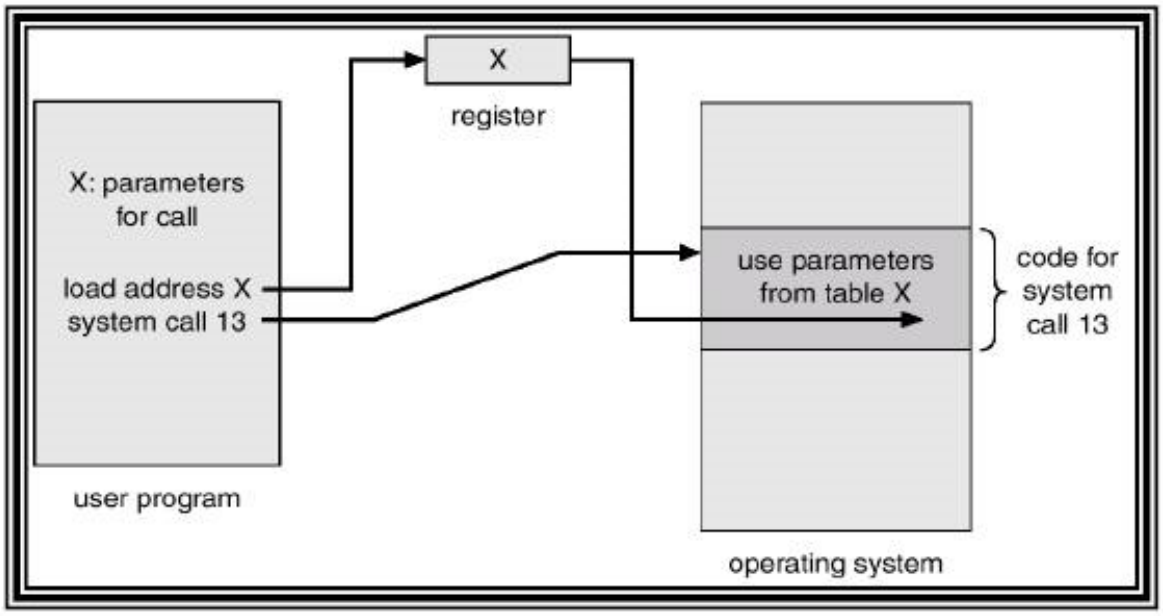


Fig. 1.14 Passing of Parameters

Types of System Calls

System calls can be grouped roughly into five categories:

i. Process Control

ii. File Management

iii. Device Management

iv. Information Maintenance

v. Communication

Process Control:

Load, execute, abort, end, create process, terminate process, get process attributes, set process attributes, allocate and free memory, wait event, signal event.

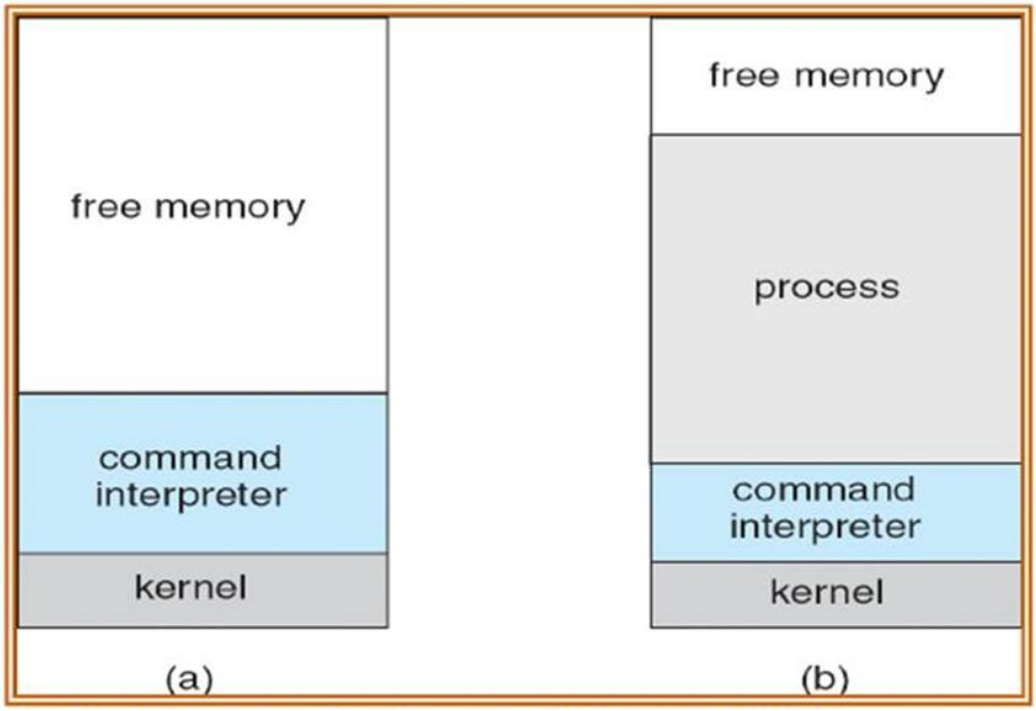


Fig. 1.15 MS-DOS execution. (a) At system startup (b) running a program

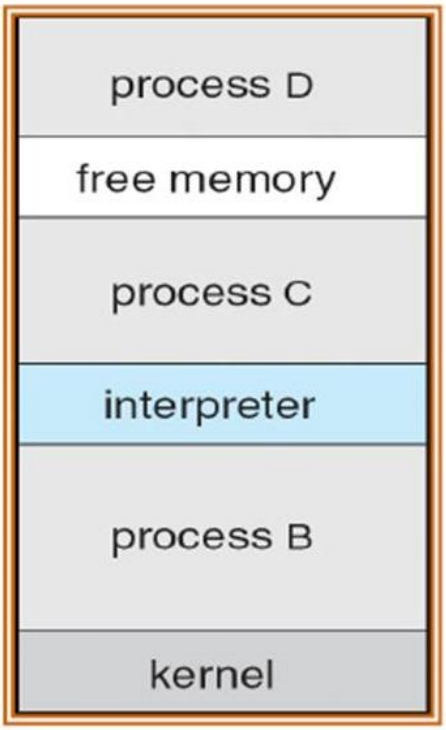


Fig. 1.16 UNIX Running Multiple Program s

File Management:

Create file, delete file, open, close, read, write,reposition, get file attribute,set file

attributes.

Device management:

Request device, release device, read, reposition, write, get device attributes, set device attributes, logically attach or detach device.

Information Maintenance:

Get time and date, set time and date, get system data, set system data, get process file or device attributes,set process file or device attributes.

Communication:

Create, close communication connection, send, receive messages, transfer status information, attach or detach remote devices. Communication may take place using:

i. Message Passing Model or

ii. Shared Memory Model

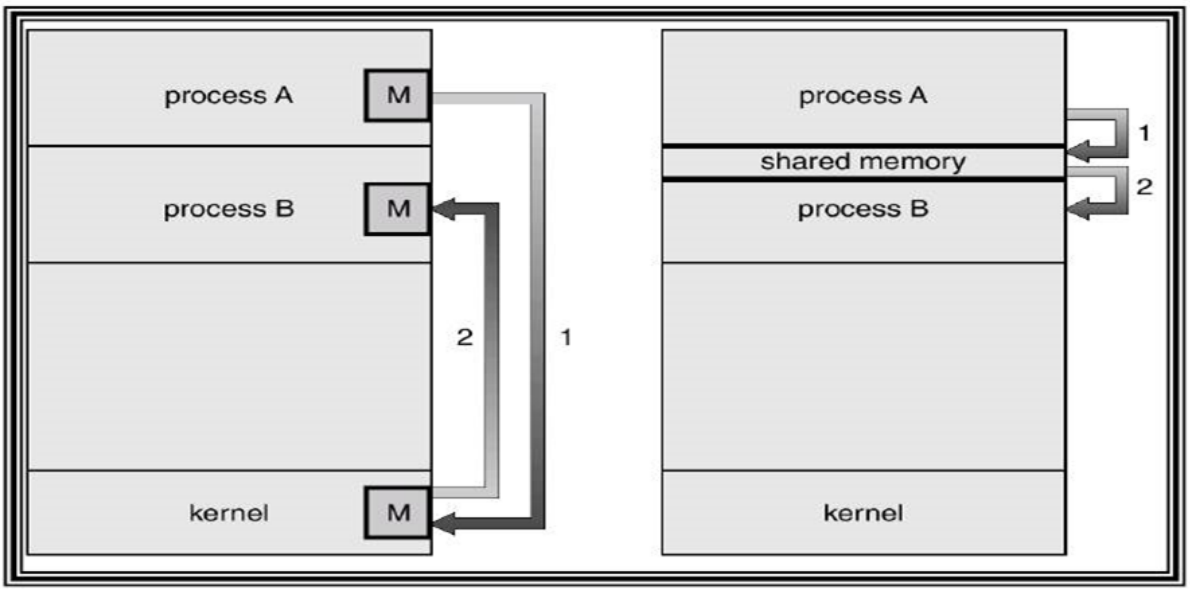


Fig. 1.17 Communication Models: (a) Message Passing Model (b) Shared Memory Model

Resources

The OS treats an entity as a resource if it satisfies the below characteristics:

- A process must request it from the OS.
- A process must suspend its operation until the entity is allocated to it.

The most common source is a file. A process must request a file before it can read it or write it. Further, if the file is unavailable, the process must wait until it becomes available. This abstract description of a resource is crucial to the way various entities(such as files, memory and devices) are managed.

Files

A sequential file is a named,linear stream bytes of memory. You can store information by opeening a file

Process

An operating system executes a variety of programs:

- Batch system – jobs
- Time-shared systems – user programs or tasks

Process – a program in execution; process execution must progress in sequential fashion. A process is a program in execution. A process is more than the program code, which is sometimes known as the text section. A process includes:

1. Program counter
2. Stack
3. Data section

The current activity, as represented by the value of the Program counter and the contents of the processor's registers. The process Stack contains temporary data (such as function parameters, return addresses, and local variables), Data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

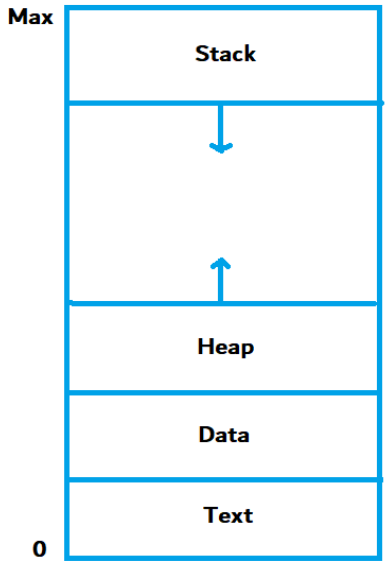


Fig. 1.18 Process in Memory

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

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.) Although two processes may be associated with the same program, they are nevertheless

considered two separate execution sequences. Several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program.

Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

Threads

A thread is a basic unit of CPU utilization. A thread,sometimes called as light weight process whereas a process is a heavyweight process.

Thread comprises:

- o A thread ID
- o A program counter
- o A register set
- o A stack.

A process is a program that performs a single thread of execution i.e., a process is a executing program with a single thread of control. For example, when a process is running a word processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. For example, the user cannot simultaneously type in characters and run the spell checker within the same process.

Traditionally a process contained only a single thread of control as it ran, many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

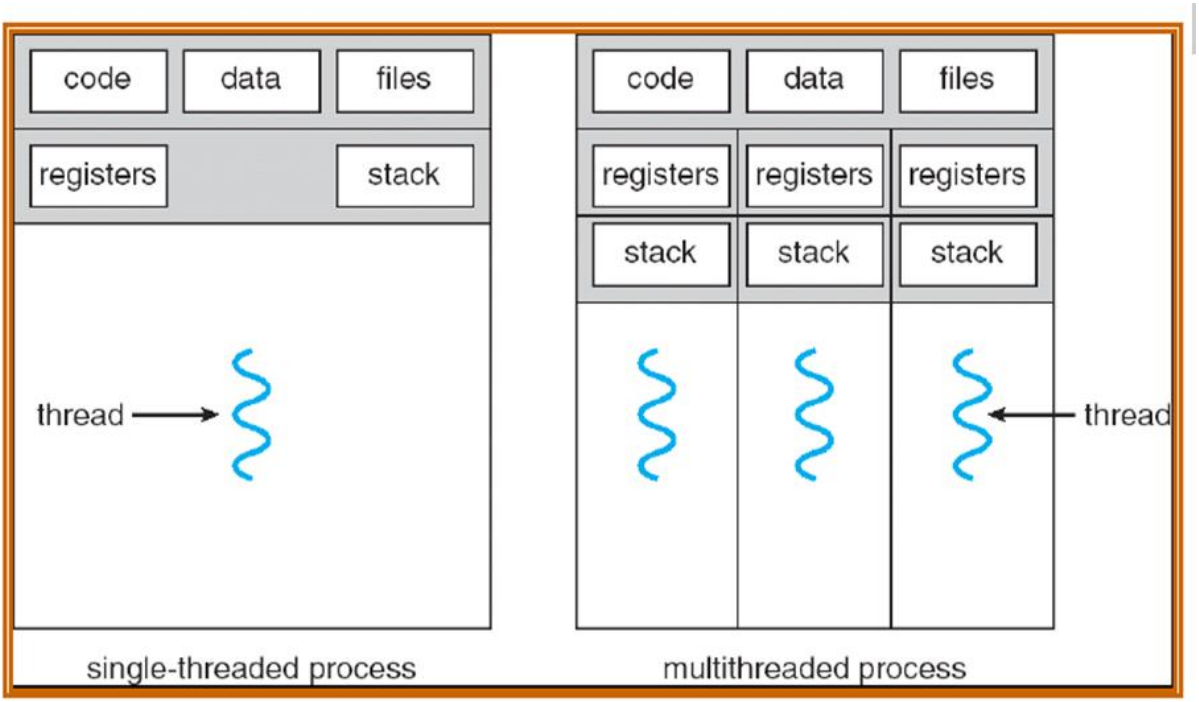


Fig. 1.19 Single-Threaded and Multi-Threaded Processes

The operating system is responsible for the following activities in connection with process and thread management:

- o The creation and deletion of both user and system processes;
- o The scheduling of processes;
- o The provision of mechanisms for synchronization,
- o Communication,
- o Deadlock handling for processes.

Benefits

The benefits of multi threaded programming can be broken down into four major categories:

Responsiveness

Multi threading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. A multi threaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

Resource sharing

By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy

Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.

• Empirically gauging the difference in overhead can be difficult, but in general it is muc more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

Utilization of multiprocessor architectures

• The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

• A singlethreaded process can only run on one CPU, no matter how many are available.

• Multithreading on a multi-CPU machine increases concurrency.

User and Kernel Threads

Threads may be provided either at the user level, for user threads, or by the kernel, for kernel

threads.

User Threads:

• User threads are supported above the kernel and are managed without kernel support i.e., they are implemented by thread library at the user level. The library porvides support for thread creation,scheduling,and management with no support from the kernel. Because the kernel is unaware of user-level threads,all thread creation and scheduling are done in user space without the need for

kernel intervention. Therefore, user-level threads are generally fast to create and manage;they have drawbacks however. If the kernel is single-threaded,then any user-level thread perfroming a blocking system call will cause the entire process to block,even if other threads ae available to run within the application. User-thread libraries include POSIX Pthreads,Mach C-threads, and Solaris 2 UI-threads.

Kernel Threads:

Kernel threads are supported and managed directly by the operating system. The kernel performs thread creation,scheduling,and management in kernel space. Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads. However, since kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Also, in a multiprocessor environment, the kernel can schedule threads on different processors. Most contemporary operating systems—including Windows NT,Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

Multithreading Models

Many systems provide support for both user and kernel threads,resulting in different multi threading models. Three common ways of establishing this relationship are:

Many-to-One Model

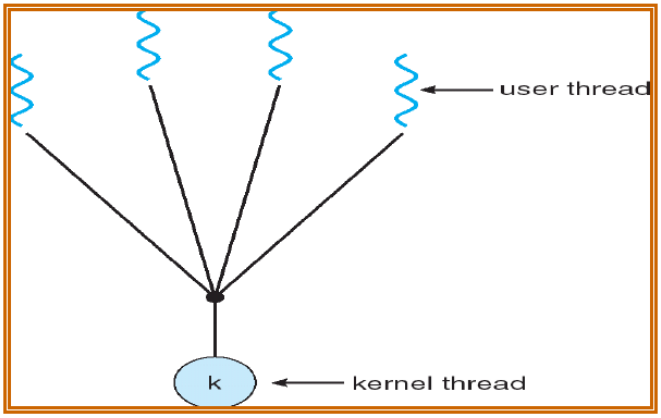


Fig. 1.20 Many-To-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. Green threads—a thread library available for Solaris 2—uses this model. In addition, user level thread libraries implemented on Operating systems that do not support kernel threads use the many-to-one model.

One-to-One Model

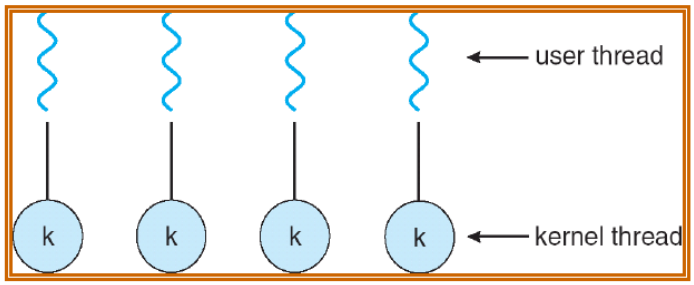


Fig. 1.21 One-To-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems—including Windows 95, 98, NT, 2000, and OS/2— implement the one-to-one model.

Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.

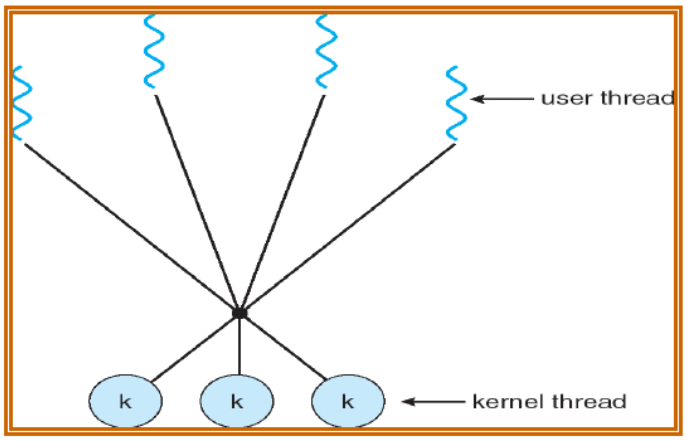


Fig. 1.22 Many-To-Many Model

The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).

• The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris 2, IRIX, HP-UX and Tru64 UNIX support this model.

Processes Vs Threads

How threads differ from processes

• Threads differ from traditional multitasking operating system processes:

- Processes are typically independent, while threads exist as subsets of a process. processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources

- Processes have separate address spaces, whereas threads share their address space. Processes interact only through system-provided inter-process communication mechanisms
- Context switching between threads in the same process is typically faster than context switching between processes. As we mentioned earlier that in many respect threads operate in the same way as that of processes.

Some of the similarities and differences are:

Similarities

• Like processes threads share CPU and only one thread active (running) at a time.

• Like processes, threads within a processes, threads within a processes execute sequentially.

• Like processes, thread can create children.

• And like process, if one thread is blocked, another thread can run.

Differences

• Unlike processes, threads are not independent of one another.

• Unlike processes, all threads can access every address in the task .

• Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account,a table of data or any item that the program has to handle. Programming problem is analysed in terms of objects and the nature of communication between them. When a program is executed,the objects interact by sending messages to one another . For example,if 'customer' and 'account' are the two objects in a program,then the customer object may send a message to the account object. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each others data or code. It is sufficient to know the type of message accepted,and the type of response returned by the objects.

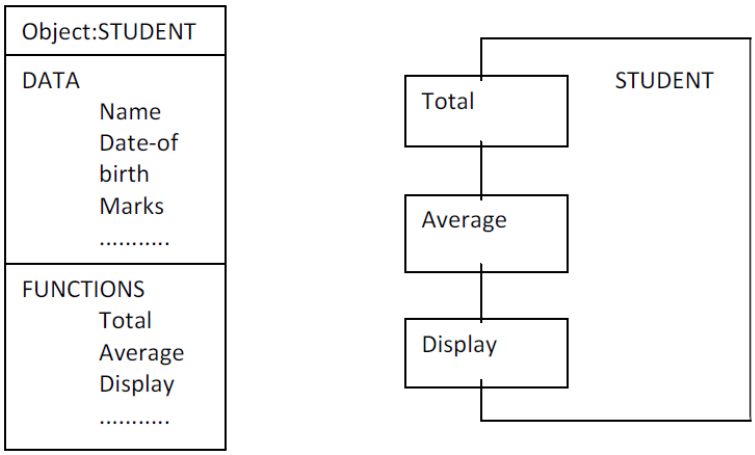


Fig. 1.23 Two ways of representing an object

Real-world objects share two characteristics: They all have state and behavior.

• Desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune).

• You may also notice that some objects, in turn, will also contain other objects.

• These real-world observations all translate into the world of object-oriented programming.

Device Management

This component of operating system manages hardware devices via their respective drivers. The operating system performs the following activities for device management.

• It keeps track of all the devices. The program responsible for keeping track of all the devices is known as I/O controller.

• It provides a uniform interface to access devices with different physical characteristics.

• It allocates the devices in an efficient manner.

• It deallocates the devices after usage.

• It decides which process gets the device and how much time it should be used.

• It optimizes the performance of each individual device.

Approaches

• **Direct I/O** – CPU software explicitly transfer data to and from the controller's data registers

– **Direct I/O with polling** – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the device driver, if interrupts are not used

– **Interrupt driven direct I/O** – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a device driver and interrupt routine

• **Memory mapped I/O** – device addressing simplifies the interface (device seen as a range of memory locations)

– **Memory mapped I/O with polling** – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the device driver.

– **Interrupt driven I/O** – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a device driver and interrupt routine Direct memory access – involves designing of hardware to avoid the CPU perform the transfer of information between the device (controller's data registers) and the memory.

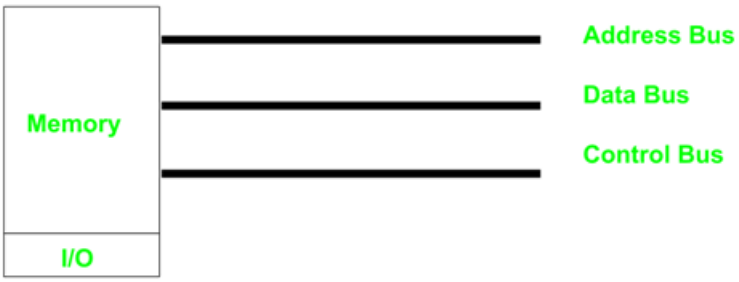


Fig. 1.24 Address Bus, Data Bus and Control Bus

I/O System Organization

An application process uses a device by issuing commands and exchanging data with the device management (device driver).

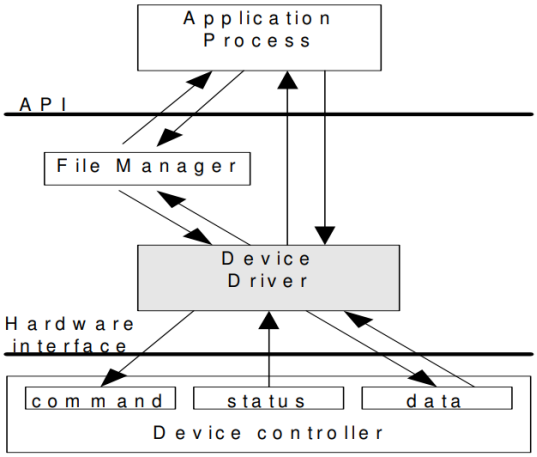


Fig. 1.25 I/O System Organization

Device driver responsibilities:

– Implement communication APIs that abstract the functionality of the device

– Provide device dependent operations to implement functions defined by the API

API should be similar across different device drivers, reducing the amount of info an application programmer needs to know to use the device. Since each device controller is specific to a particular device, the device driver implementation will be device specific, to

– Provide correct commands to the controller

– Interpret the controller status register (CSR) correctly

– Transfer data to and from device controller data registers as required for correct device operation

Direct I/O versus memory mapped I/O

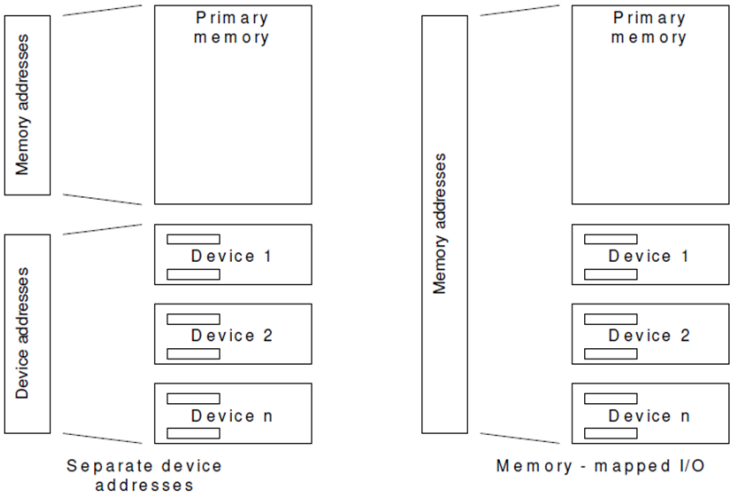


Fig. 1.26 Direct I/O vs. Memory Mapped I/O

I/O with polling

Each I/O operation requires that the software and hardware coordinate their operations to accomplish desired effect. In direct I/O pooling this coordination is done in the device driver; While managing the I/O, the device manager will poll the busy/done flags to detect the operation's completion; thus, the CPU starts the device, then polls the CSR to determine when the operation has completed. With this approach is difficult to achieve high CPU utilization, since the CPU must constantly check the controller status.

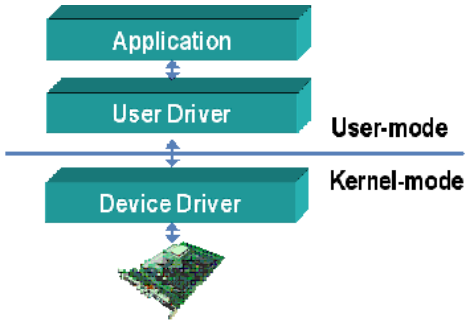


Fig. 1.27 User Mode and Kernel Mode

I/O with polling – read

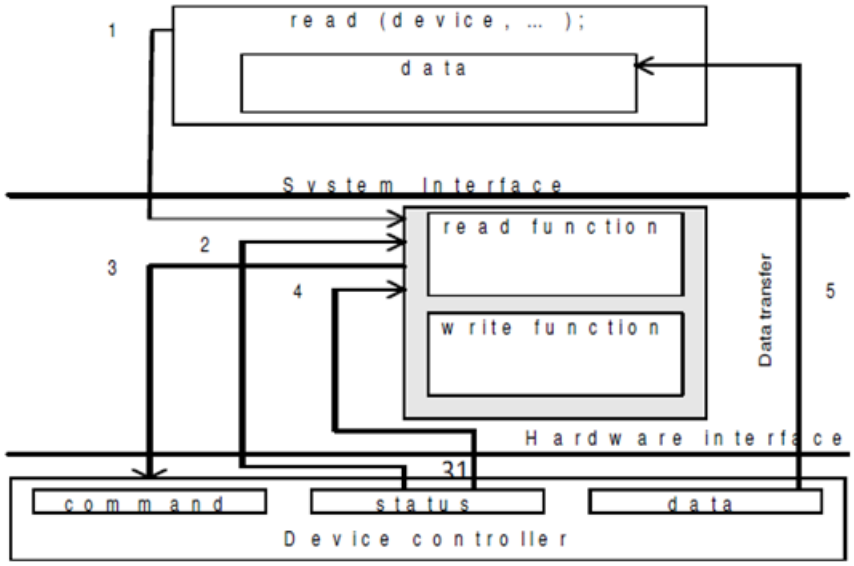


Fig. 1.28 I/O with Polling - Read

1. Application process requests a read operation
2. The device driver queries the CSR to determine whether de device is idle; if device is busy, the driver waits for it to become idle
3. The driver stores an input command into the controller's command register, thus starting the device
4. The driver repeatedly reads the content of CSR to detect the completion of the read operation
5. The driver copies the content of the controller's data register(s) into the main memory user's process's space.

I/O with Polling – Write

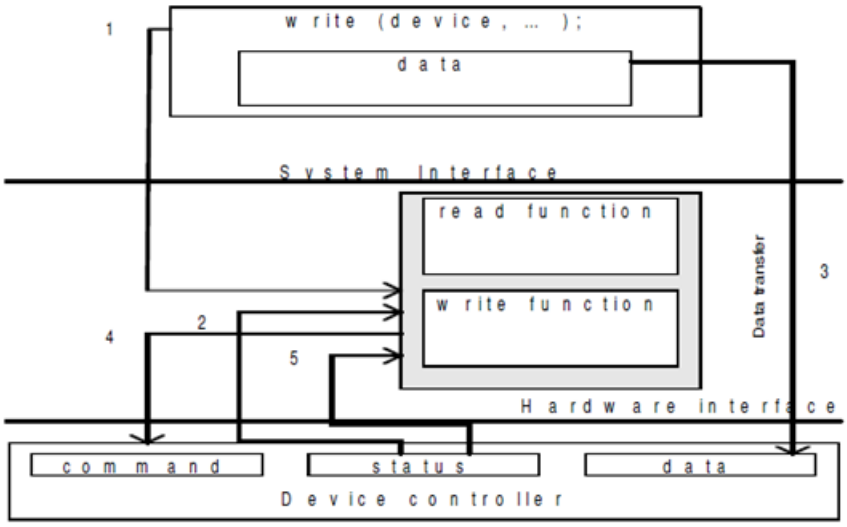


Fig. 1.29 I/O with Polling - Write

1. The application process requests a write operation

2. The device driver queries the CSR to determine if the device is idle; if busy, it will wait to become idle

3. The device driver copies data from user space memory to the controller's data register(s)

4. The driver stores an output command into the command register, thus starting the device

5. The driver repeatedly reads the CSR to determine when the device completed its operation.

Interrupt driven I/O

In a multiprogramming system the wasted CPU time (in polled I/O) could be used by another process; because the CPU is used by other processes in addition to the one waiting for the I/O operation completion, in multiprogramming system may result a sporadic detection of I/O completion; this may be remedied by use of interrupts. The reason for incorporating the interrupts into a computer hardware is to eliminate the need for a device driver to constantly poll the CSR. Instead polling, the device controller "automatically" notifies the device driver when the operation has completed.

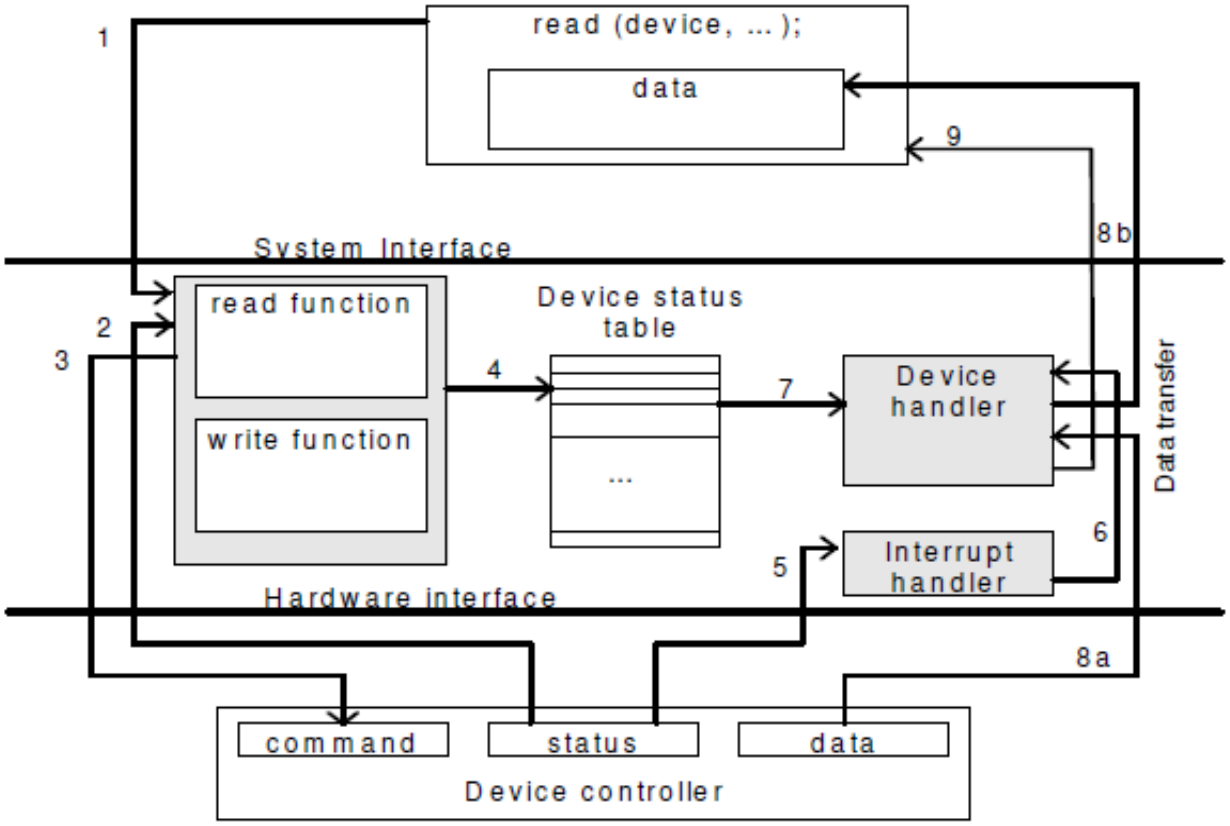


Fig. 1.29 Interrupt-driven I/O

1. The application process requests a read operation

2. The device driver queries the CSR to find out if the device is idle; if busy, then it waits until the device becomes idle

3. The driver stores an input command into the controller's command register, thus starting the device

4. When this part of the device driver completes its work, it saves information regarding the operation it began in the device status table; this table contains an entry for each device in system; the information written into this table contains the return address of the original call and any special parameters for the I/O operation; the CPU, after is doing this, can be used by other program, so the device manager invokes the scheduler part of the process manager. It

then terminates

5. The device completes the operation and interrupts the CPU, therefore causing an interrupt handler to run

6. The interrupt handler determines which device caused the interrupt; it then branches to the device handler for that device

7. The device driver retrieves the pending I/O status information from the device status table

8. (a,b) The device driver copies the content of the controller's data register(s) into the user process's space

9. the device handler returns the control to the application process (knowing the return ddress from the device status table). Same sequence (or similar) of operations will be accomplished for an output operation.

Overlapping CPU execution with device operation

The software interface to an I/O device usually enables the operating system to execute alternative processes when any specific process is waiting for I/O to complete, while preserving serial execution semantics for an individual process. That means that whenever the programmers will use read statement in a program, they will know that the next instruction will not execute until the read instruction has completed.

Consider the following code:

…

read (device, "%d", x);

y=f(x);

….

Direct Memory Access

• Traditional I/O

– Polling approach:

• CPU transfer data between the controller data registers and the primary memory

• Output operations - device driver copies data from the application process data area to the controller; vice versa for input operations

– Interrupt driven I/O approach - the interrupt handler is responsible for the transfer task

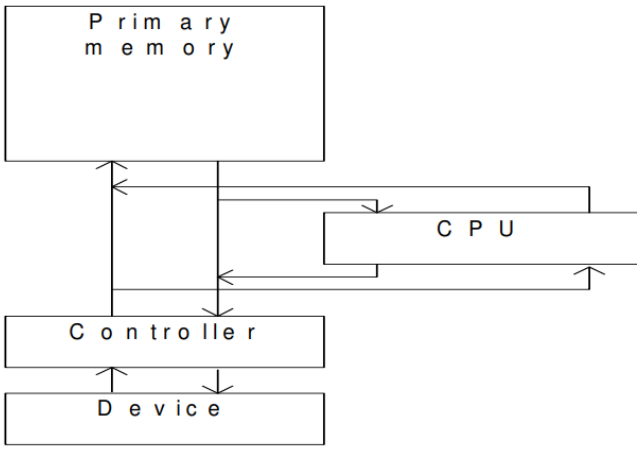


Fig. 1.30 Traditional I/O

DMA controllers are able to read and write information directly from /to primary memory, with no software intervention. The I/O operation has to be initiated by the driver. DMA hardware enables the data transfer to be accomplished without using the CPU at all. The DMA controller must include an address register (and address generation hardware) loaded by the driver with a pointer to the relevant memory block; this pointer is used by the DMA hardware to locate the target block in primary memory.

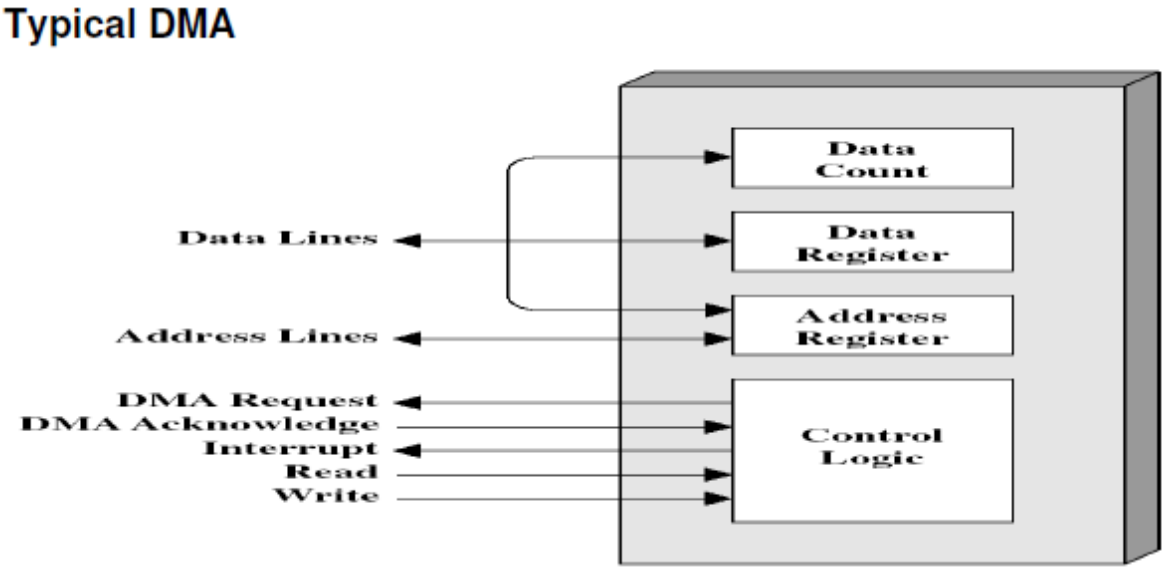


Fig. 1.31 Typical DMA

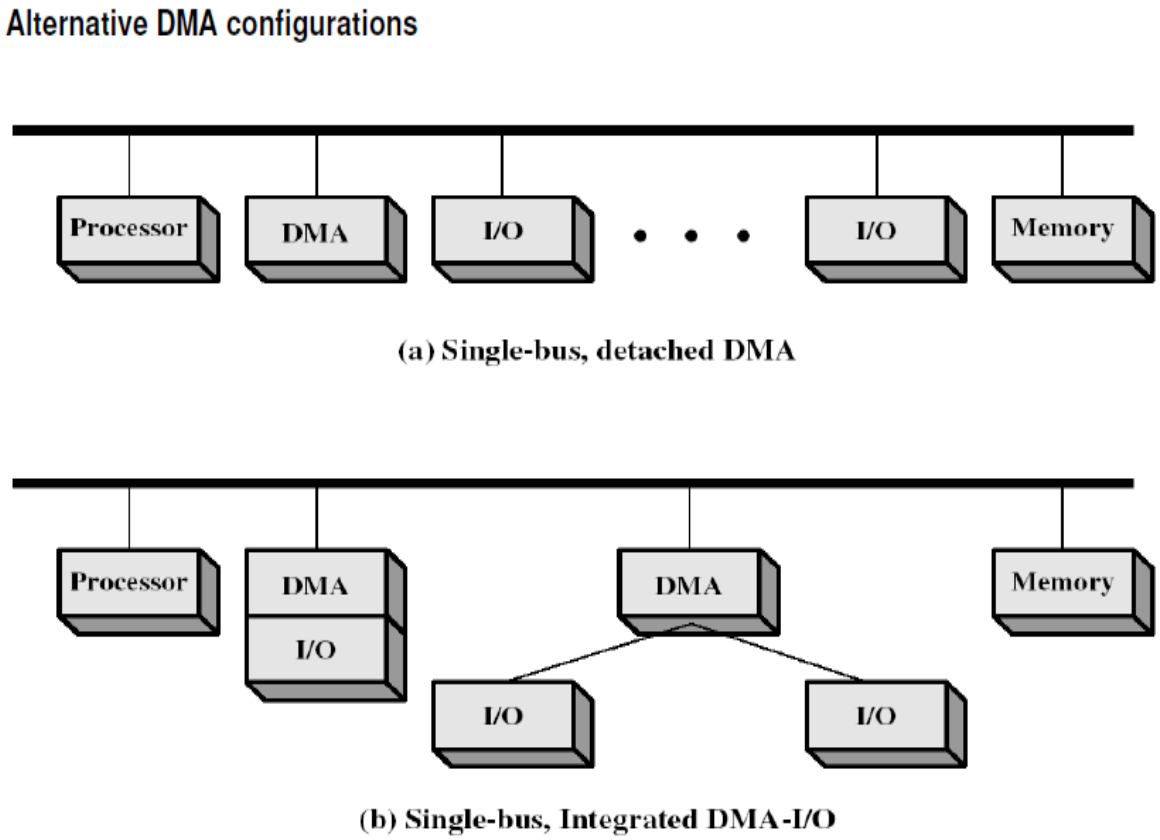


Fig. 1.32 Alternative DMA Configurations

Buffering

Buffering is a technique by which a device manager keeps the slower I/O devices busy when a process is not requiring I/O operations. **Input buffering** is the process of reading the data into the primary memory before the process requests it. **Output buffering** is the process of saving the data in the memory and then writing it to the device while the process continues its execution.

Hardware level buffering

Consider a simple character device controller that reads a single byte form a modem for each input operation.

- Normal operation: read occurs, the driver passes a read command to the controller; the controller instructs the device to put the next character into one-byte data controller's register; the process calling for byte waits for the operation to complete and the retrieves the character from the data register

Add a hardware buffer to the controller to decrease the amount of time the process has to wait

- Buffered operation: the next character to be read by the process has already been placed into the data register, even the process has not yet called for the read operation

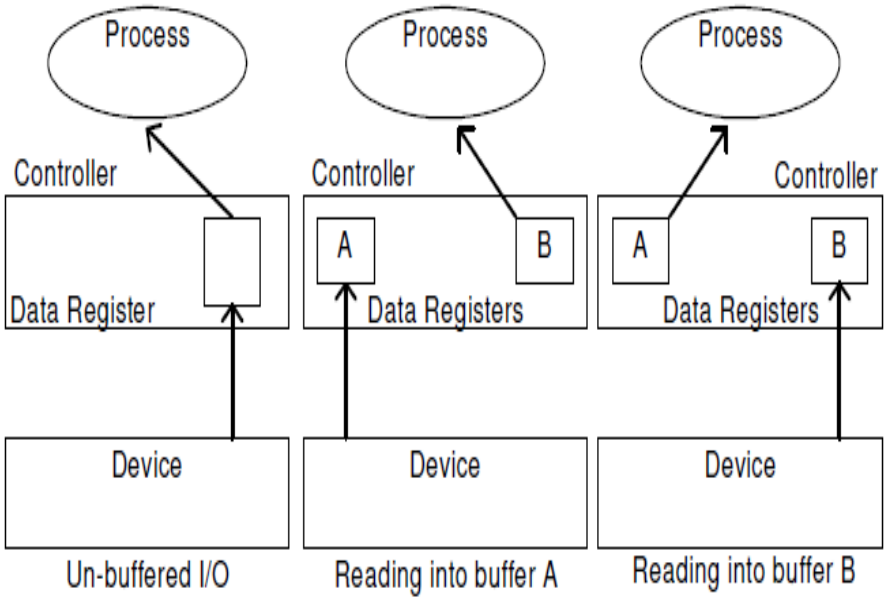


Fig. 1.33 Hardware-Level Buffering

Driver level buffering

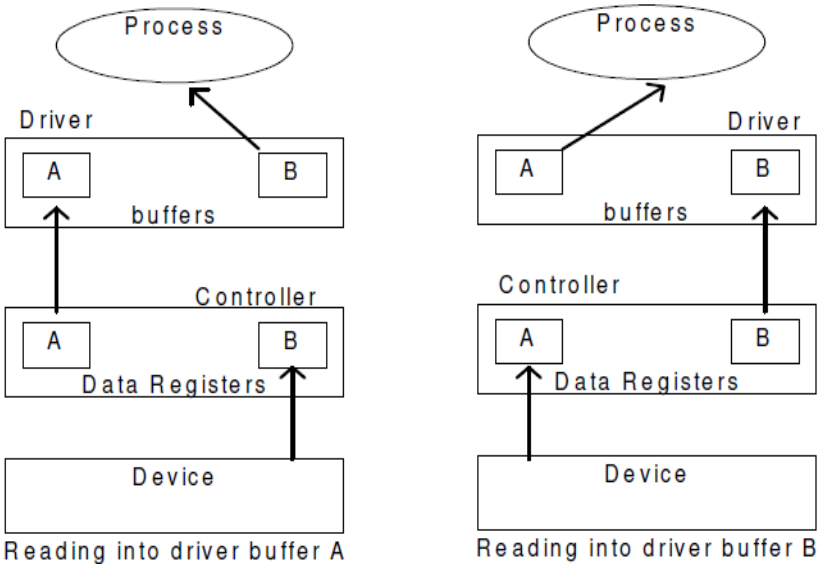


Fig 1.34 Driver-Level Buffering

This is generally called **double buffering**. One buffer is for the driver to store the data while waiting for the higher layers to read it. The other buffer is to store data from the lower level module. This technique can be used for the block-oriented devices (buffers must be large enough to accommodate a block of data).

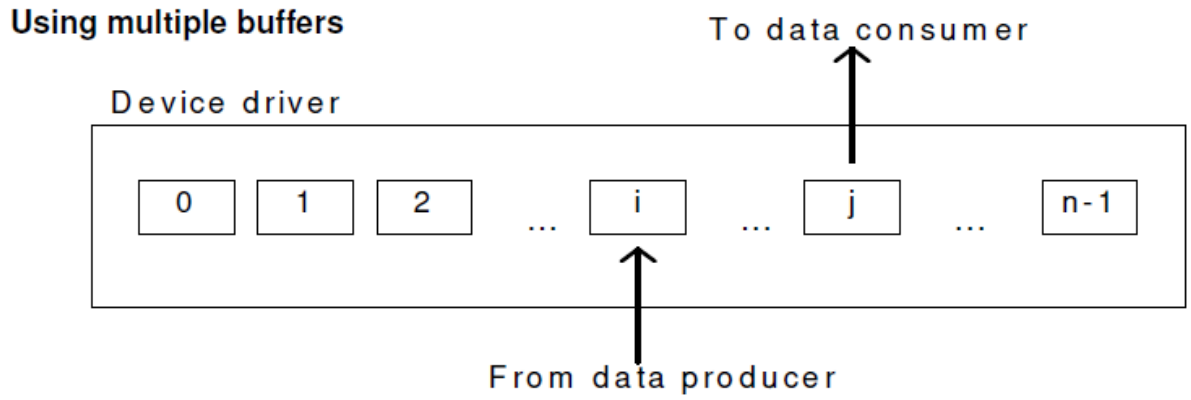


Fig. 1.35 Using Multiple Buffers

The number of buffers is extended from two to n. The data producer (controller in read operations, CPU in write operations) is writing into buffer i while the data consumer (the CPU in read operations, the controller in write operations) is reading form buffer j. In this configuration buffers j+1 to n-1 and 0 to i-1 are full. This is known as circular buffering technique.

Device Driver

It is a software program that controls a particular type of device attached to the computer. It provides an interface to the hardware devices without the requirement to know the precise information about the hardware. A device driver communicates with the device through a bus or communication sub system.

Responsibilities

1. Initialize devices

2. Interpreting the commands from the operating system

3. Manage data transfers

4. Accept and process interrupts

5. Maintain the integrity of driver and kernel data structures

Device Driver Interface

Each operating system defines an architecture for its device management system. The designs are different from operating system to operating system; there is no universal organization. Each operating system has two major interfaces to the device manager:

– The driver API

– The interface between a driver and the operating system kernel.

Driver API

• Provides a set of functions that an programmer can call to manage a device (usually comms or storage). The device manager

– Must track the state of the device: when it is idle, when is being used and by which

process

– Must maintain the information in the device status table

– May maintain a device descriptor to store other information about the device

• open/close functions to allow initiate/terminate of the device's use

– open – allocates the device and initializes the tables and the device for use

– close – releases dynamic tables entries and releases the device

• read/write functions to allow the programmer to read/write from/to the device

– A consistent way to do these operations across all the devices is not possible; so a concession by dividing the devices into classes is made:

o such as character devices or block devices

o Sequential devices or randomly accessed devices

• **ioctl** function to allow programmers to implement device specific functions.

The driver - kernel interface

The device driver must execute privileged instructions when it starts the device; this means that the device driver must be executed as part of the operating system rather than part of a user program. The driver must also be able to read/write info from/to the address spaces of different processes, since same device driver can be used by different processes

• Two ways of dealing with the device drivers

– Old way: driver is part of the operating system, to add a new device driver, the whole OS must have been complied

– Modern way: drivers installation is allowed without re-compilation of the OS by using reconfigurable device drivers; the OS dynamically binds the OS code to the driver functions.

o A reconfigurable device driver has to have a fixed, standardized API

o The kernel has to provide an interface to the device driver to allocate/dealocate space for buffers, manipulate tables in the kernel, etc.

Reconfigurable Device Drivers

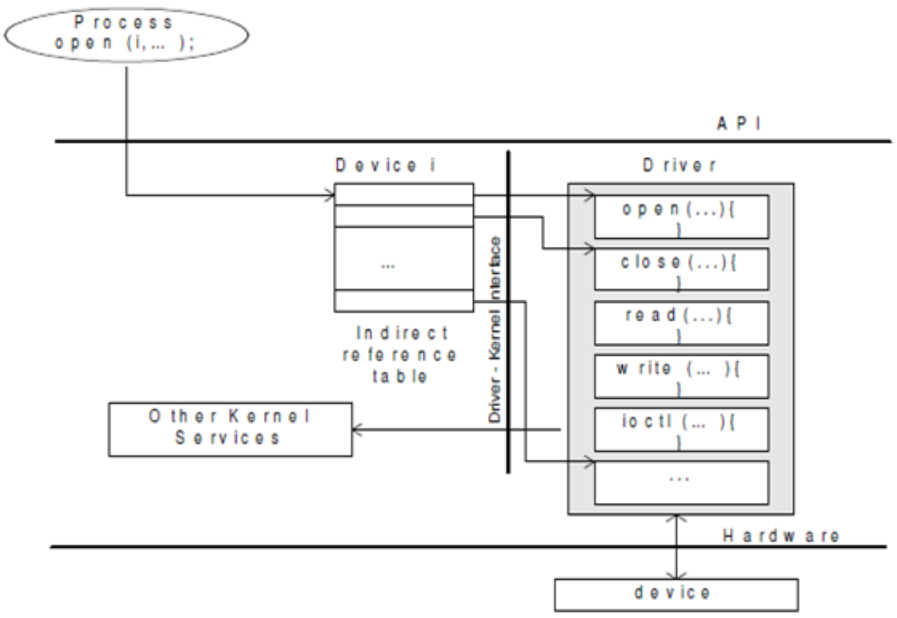


Fig. 1.36 Reconfigurable Device Drivers

The OS uses an indirect reference table to access the different driver entry points, based on the device identifier and function name. The indirect reference table is filled with appropriate values whenever the device driver loads (because the detection of a PnP device or at boot time). When a process performs a system call, the kernel passes the call onto the device driver via the indirect reference table.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

Common to : CSE , IT

UNIT II

UNIT 2 PROCESS MANAGEMENT 9 Hrs.

Processes - Process concepts - Process scheduling - Operations on processes - Cooperating processes - CPU scheduling - Basic concepts - Scheduling criteria - Scheduling algorithms - Preemptive strategies - Non-preemptive strategies

UNIT – II- SCS1301- OPERATING SYSTEM

UNIT 2

PROCESS MANAGEMENT

INTRODUCTION TO PROCESSES:

- Early computer systems allowed only one program to be executed at a time.
- This program had complete control of the system and had access to all the system's resources.
- In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.
- This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution.
- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.
- **Definition:** A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes:
    - Operating-system processes execute system code, and
    - User processes execute user code.
- Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
- A process is the unit of work in a modern time-sharing system.
- The more complex the operating system is, the more it is expected to do on behalf of its users.
- Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself.
- A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code.

PROCESSES:

- A process is mainly a program in execution where the execution of a process must progress in a sequential order or based on some priority or algorithms.
- In other words, it is an entity that represents the fundamental working that has been assigned to a system.
- When a program gets loaded into the memory, it is said to as process. This processing can be categorized into 4 sections. These are:
    - Heap
    - Stack
    - Data
    - Text

PROCESS CONCEPTS:

- A question that arises in discussing operating systems involves what to call all the CPU activities.
- A batch system executes jobs, whereas a time-shared system has user programs, or tasks.
- Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package.
- Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management.
- In many respects, all these activities are similar, so we call all of them processes.
- The terms job and process are used almost interchangeably used.
- Although we personally prefer the term process, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing.
- It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job.

THE PROCESS:

- An operating system executes a variety of programs:
    - Batch system – jobs
    - Time-shared systems – user programs or tasks
- We use the terms job and process almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.

- A process is a program in execution. A process is more than the program code, which is sometimes known as the text section.
- A process includes:
    - Counter
    - Program stack
    - Data section
        - The current activity, as represented by the value of the program counter and the contents of the processor's registers.
        - The process stack contains temporary data (such as function parameters, return addresses, and local variables)
        - A data section, which contains global variables.
- Process may also include a heap, which is memory that is dynamically allocated during process run time.

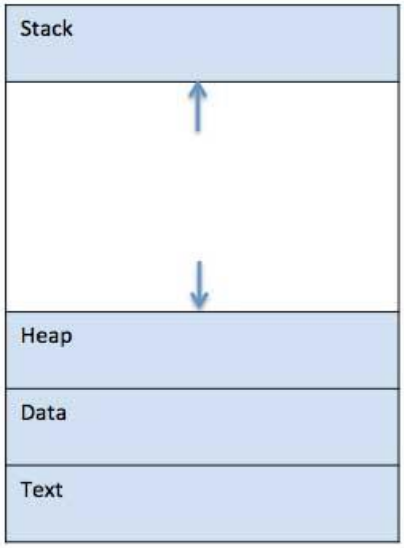


Fig 1 : The Process

- **STACK** - The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- **HEAP** - This is dynamically allocated memory to a process during its run time.
- **TEXT** - This includes the current activity represented by the value of Program Counter and the contents of the processor's registers**.**
- **DATA -** This section contains the global and static variables.

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

PROCESS STATE / PROCESS LIFE CYCLE:

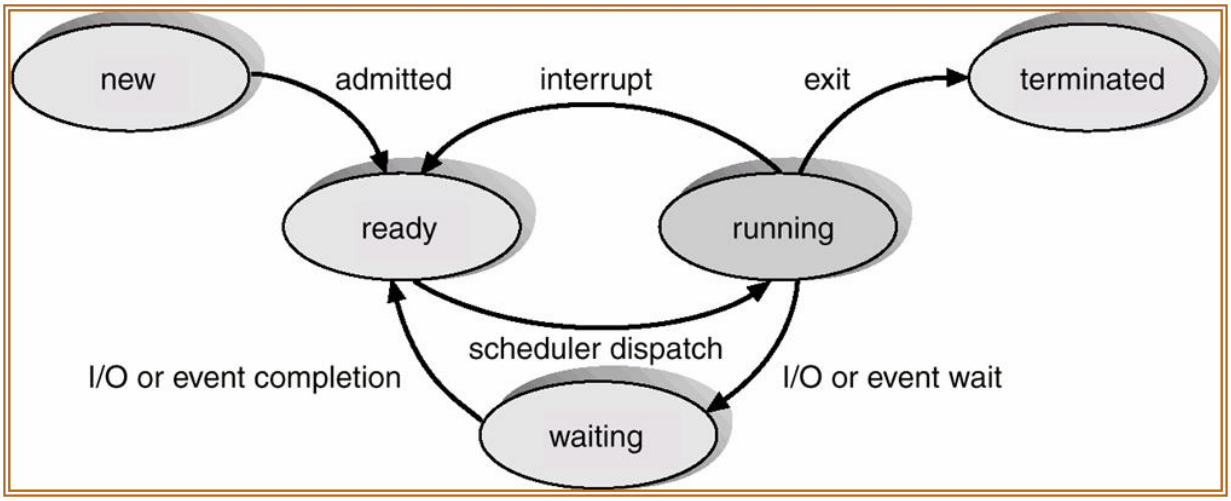


Fig 2 : Process life cycle

- When a process executes, it passes through different states.
- These stages may differ in different operating systems, and the names of these states are also not standardized.
- In general, a process can have one of the following five states at a time.
    - New/Start
    - Running
    - Waiting
    - Ready
    - Terminated
- **START / NEW –**
    - This is the initial state when a process is first started / created.
- **READY –**
    - The process is waiting to be assigned to a processor.

  - Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
  - Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
- **RUNNING –**
  - Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **WAITING –**
  - Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **TERMINATED OR EXIT –**
  - Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.
- These names are arbitrary, and they vary across operating systems.

PROCESS CONTROL BLOCK (PCB):

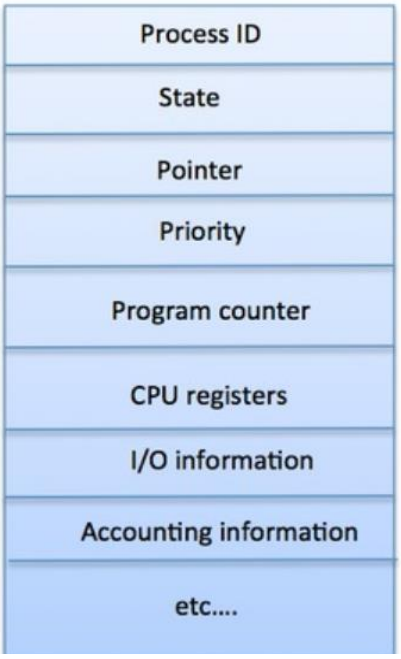


Fig 3 : PCB

- A Process Control Block is a data structure maintained by the Operating System for every process.
- The PCB is identified by an integer process ID (PID).

- A PCB keeps all the information needed to keep track of a process as listed below ,
- **PROCESS STATE**
    - The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **PROCESS PRIVILEGES**
    - This is required to allow/disallow access to system resources.
- **PROCESS ID**
    - Unique identification for each of the process in the operating system**.**
- **POINTER**
    - A pointer to parent process.
- **PROGRAM COUNTER**
    - Program Counter is a pointer to the address of the next instruction to be executed for this process**.**
- **CPU REGISTERS**
    - Various CPU registers where process need to be stored for execution for running state.
- **CPU SCHEDULING INFORMATION**
    - Process priority and other scheduling information which is required to schedule the process.
- **MEMORY MANAGEMENT INFORMATION**
    - This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **ACCOUNTING INFORMATION**
    - This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO STATUS INFORMATION**
    - This includes a list of I/O devices allocated to the process.
- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. (shown in above diagram)
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

DIAGRAM SHOWING CPU SWITCH PROCESS TO PROCESS

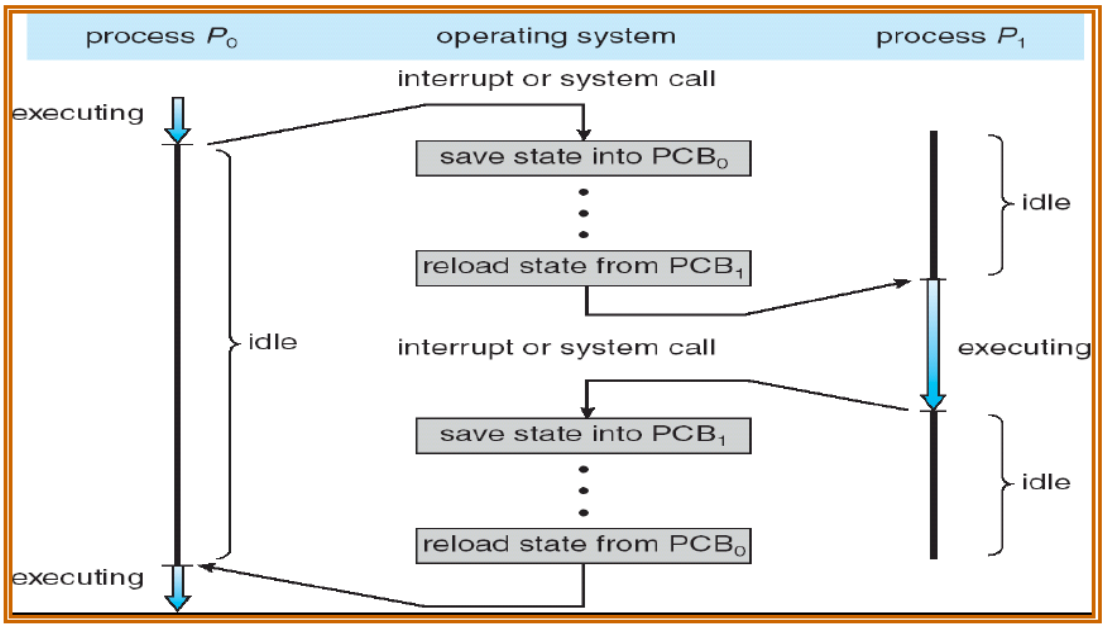


Fig 4 : CPU switch to process

PROCESS SCHEDULING:

- A uniprocessor system can have only one running process. If more process exist, the rest must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

PROCESS SCHEDULING QUEUES:

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

- Process migration between the various queues:
    - Job queue
    - Ready queue
    - Device queues

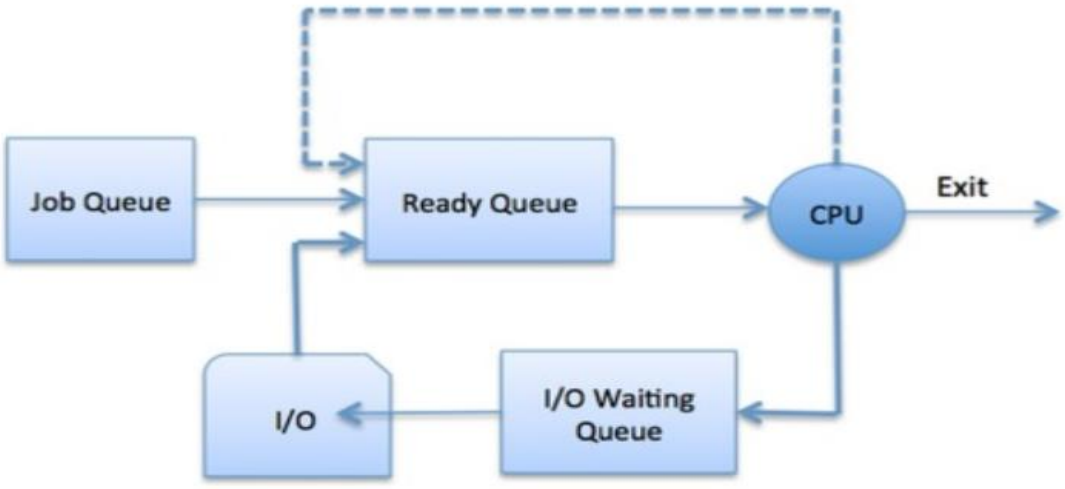


Fig 5 : Process scheduling queues

- **JOB QUEUE** − This queue keeps all the processes in the system.
- **READY QUEUE** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **DEVICE QUEUES** − The processes which are blocked due to unavailability of an I/O device constitute this queue.
- The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.).
- The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

TWO STATE PROCESS MODEL:

- Two-state process model refers to running and non-running states which are described below

- **RUNNING**
    - When a new process is created, it enters into the system as in the running state.
- **NOT RUNNING**

- Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list.

QUEUING DIAGRAM REPRESENTATION FOR PROCESS SCHEDULING

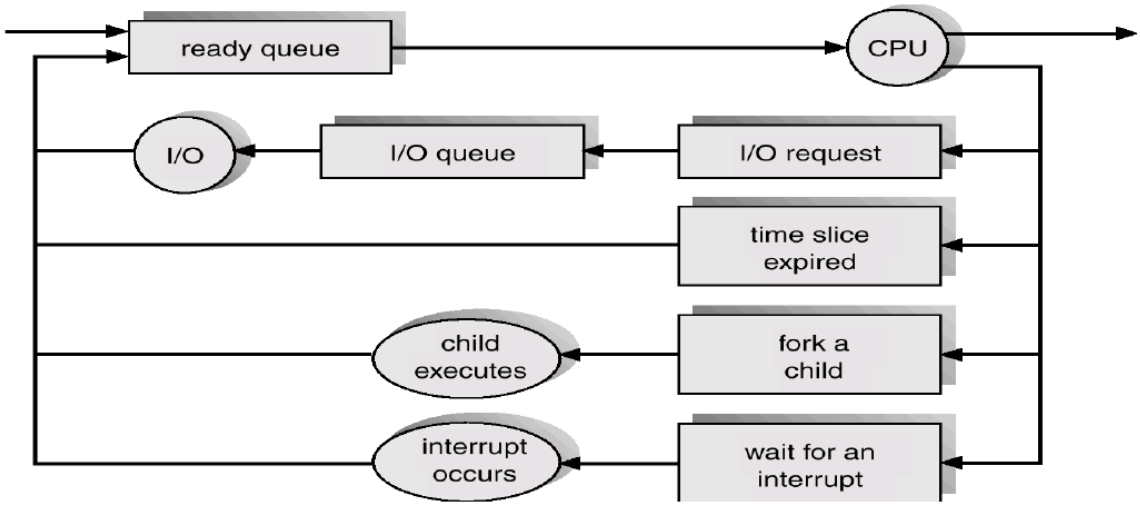


Fig 6 : Queuing diagram

- Each rectangular box represents a queue.
- Two types of queues are present:
  - the ready queue and
  - a set of device queues
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or is dispatched.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new sub process and wait for the sub process's termination.
  - The process could be removed forcibly from the CPU, as a result of an
  - Interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.

- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources de-allocated.

SCHEDULERS:

- Schedulers are special system software which handles the process scheduling in various ways.
- Their main task is to select the jobs to be submitted into the system and to decide which process to run.
- Schedulers are of three types –
    - Long-Term Scheduler
    - Short-Term Scheduler
    - Medium-Term Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

- **LONG TERM SCHEDULER:**
    - It is also called a job scheduler.

- A long-term scheduler determines which programs are admitted to the system for processing.
- It selects processes from the queue and loads them into memory for execution.
- Process loads into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.
- It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

- **SHORT TERM SCHEDULER:**
  - It is also called as CPU scheduler.
  - Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process.
  - CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
  - Short-term schedulers, also known as dispatchers, make the decision of which process to execute next.
  - Short-term schedulers are faster than long-term schedulers.
- **MEDIUM TERM SCHEDULER:**
  - Medium-term scheduling is a part of swapping.
  - It removes the processes from the memory.
  - It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.
  - A running process may become suspended if it makes an I/O request.
  - A suspended process cannot make any progress towards completion.
  - In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

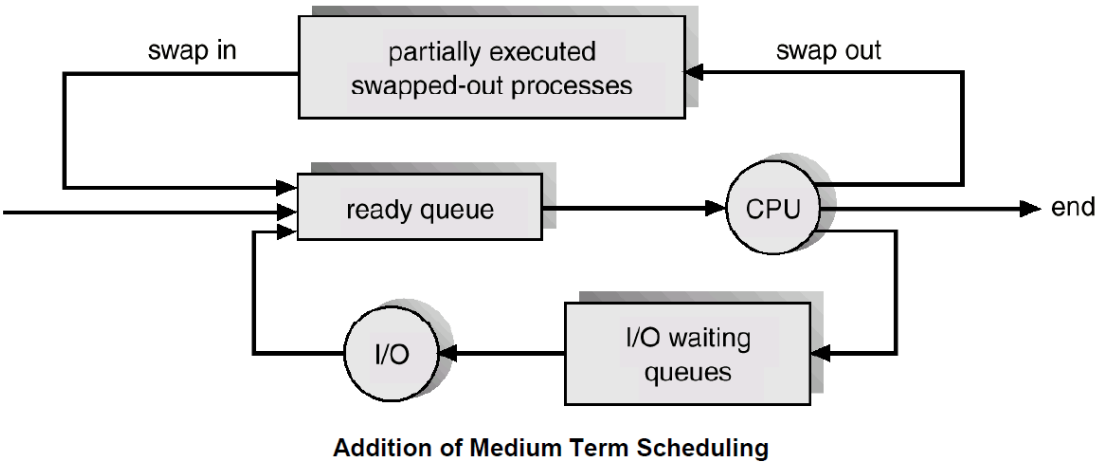


Fig 7 : Scheduling process

CONTEXT SWITCH:

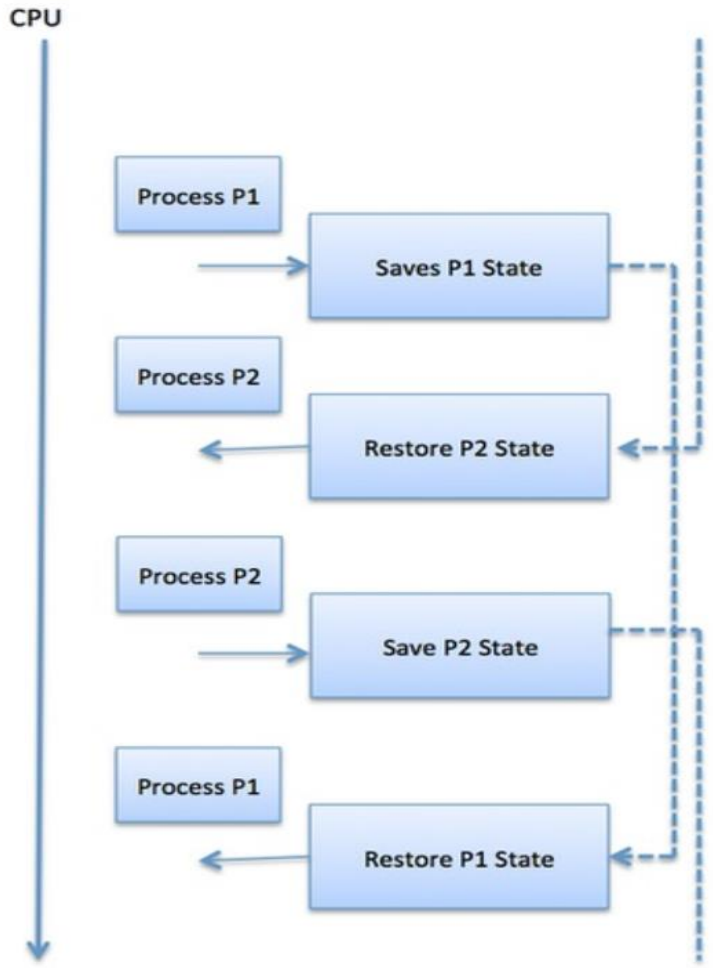


Fig 8 : Context switch

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.
- Using this technique, a context switcher enables multiple processes to share a single CPU.
- Context switching is an essential part of a multitasking operating system features.
- Context switches are computationally intensive since register and memory state must be saved and restored.
- To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.
- When the process is switched, the following information is stored for later use.
    - Program Counter
    - Scheduling information

- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

OPERATIONS ON PROCESSES:

- The processes in the system can execute concurrently, and they must be created and deleted dynamically.
- Thus, the operating system must provide a mechanism (or facility) for process creation and termination.
- In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

PROCESS CREATION:

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

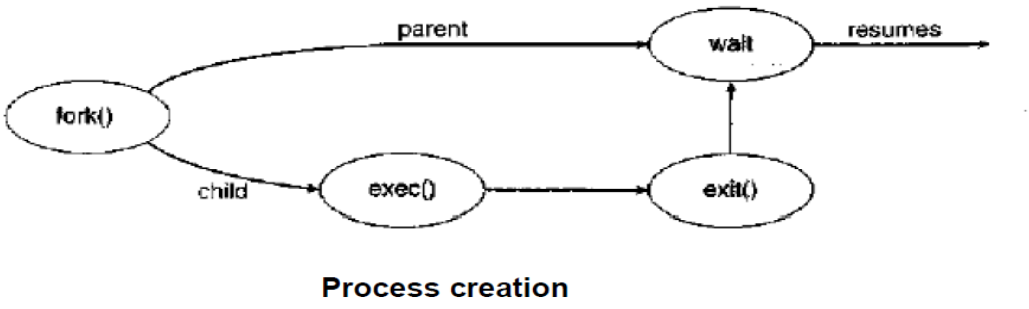
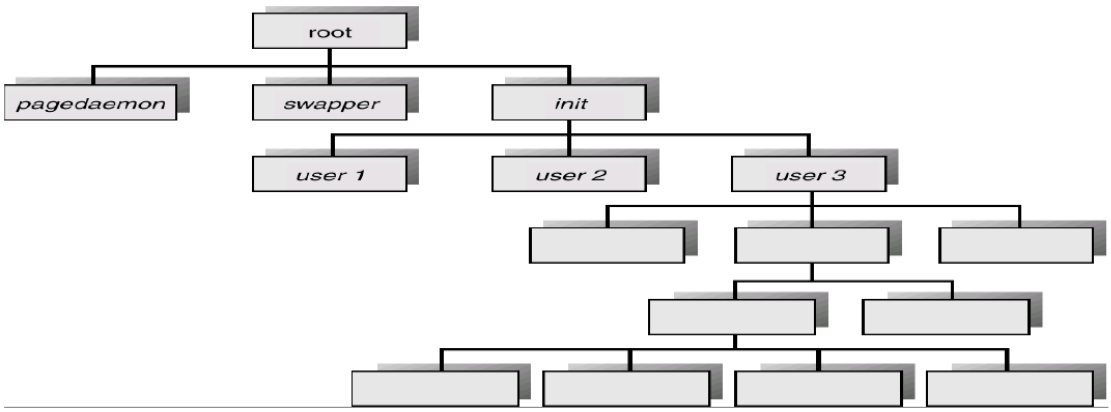


Fig 9 : Process creation

- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, and the new processes are called the children of that process.



A Tree of Processes On A Typical UNIX System

Fig 10 : tree process in Unix system

- Each of these new processes may in turn create other processes, forming a tree of processes.
- When a process creates a sub process, that sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- When a process creates a new process, two possibilities exist in terms of execution:
    - The parent continues to execute concurrently with its children.
    - The parent waits until some or all of its children have terminated.
- There are also two possibilities in terms of the address space of the new process:
    - The child process is a duplicate of the parent process (it has the same program and data as the parent).

COOPERATING PROCESSES:

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.

- Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system.
- There are several reasons for providing an environment that allows process cooperation:
- **INFORMATION SHARING:**
    - Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **COMPUTATION SPEEDUP:**
    - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
- **MODULARITY:**
    - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **CONVENIENCE:**
    - Even an individual user may work on many tasks at the same time.
- Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.
- To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process.
- To allow producer and consumer processes to run concurrently, we must have available **A BUFFER** of items that can be filled by the producer and emptied by the consumer.
- This **BUFFER** will reside in a region of memory that is shared by the producer and consumer processes.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used:
    - Unbounded buffer
    - Bounded buffer
- **UNBOUNDED BUFFER** –
    - 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.

- **BOUNDED BUFFER –**
    - It assumes a fixed buffer size.
    - In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

CPU SCHEDULING:

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU
- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).

CPU SCHEDULING: DISPATCHER

- Another component involved in the CPU scheduling function is the Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program from where it left last time.
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
    - The time it takes for the dispatcher to stop one process and start another running is known as the **DISPATCH LATENCY.**

TYPES OF CPU SCHEDULING:

- CPU scheduling decisions may take place under the following four circumstances:
- When a process switches from the **running state** to the **waiting** state (for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs).
- When a process switches from the **waiting state** to the **ready state** (for example, completion of I/O).
- When a process terminates.

CPU SCHEDULING: SCHEDULING CRITERIA

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.
- The criteria include the following
- **CPU UTILIZATION:**
    - We want to keep the CPU as busy as possible.
    - Conceptually, CPU utilization can range from 0 to 100 percent.
    - In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **THROUGHPUT**
    - It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes**.**
- **TURNAROUND TIME**
    - It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process (Wall clock time).
- **WAITING TIME**
    - The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- **LOAD AVERAGE**
    - It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- **RESPONSE TIME**
    - Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

SCHEDULING ALGORITHMS:

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU scheduling algorithms:
    - First Come First Serve (FCFS) Scheduling
    - Shortest-Job-First (SJF) Scheduling
    - Priority Scheduling
    - Shortest Remaining Time (SRT) Scheduling
    - Round Robin (RR) Scheduling
    - Multilevel Queue Scheduling
    - Multilevel Feedback Queue Scheduling

FCFS - FIRST COME FIRST SERVE SCHEDULING:

- In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.
- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

COMMON DEFINITIONS INVOLVED IN CALCULATION PROCESS

- **ARRIVAL TIME:** Time taken for the arrival of each process in the CPU Scheduling Queue.
- **COMPLETION TIME:** Time taken for the execution to complete, starting from arrival time.
- **TURN AROUND TIME:** Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.
- **WAITING TIME:** Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process**.**

CPU SCHEDULING ALGORITHMS:

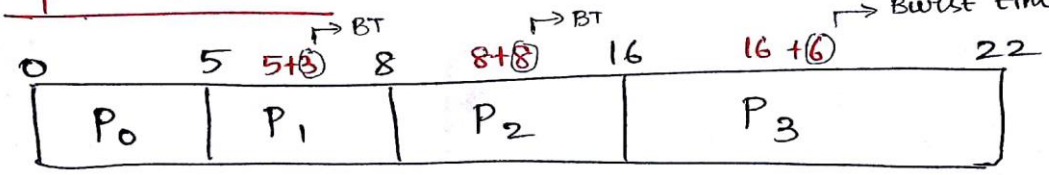
* FCFS → first come first serve
* SJF → shortest job first.
* SRT → shortest Remaining Time.
* Priority based scheduling
* Round robin based scheduling.

FCFS:

CT − AT TAT − BT

Process	Arrival time (AT)	Burst time (BT)	completion time (CT)	Turnaround time (TAT)	waiting time (WT)
P_0	0	5	5	5	0
P_1	1	3	8	7	4
P_2	2	8	16	14	6
P_3	3	6	22	19	13

Gantt chart:



* completion time values are found by using Gantt chart.

$$\text{Avg (TAT)} = \frac{\text{tot (TAT)}}{\text{no. of process}} = \frac{5+7+14+19}{4}$$

$$= \frac{45}{4} = 11.25 \text{ msec}$$

$$\text{Avg (WT)} = \frac{0+4+6+13}{4} = \frac{23}{4} = 5.75 \text{ msec}$$

SJF: (check notes for Explanation)

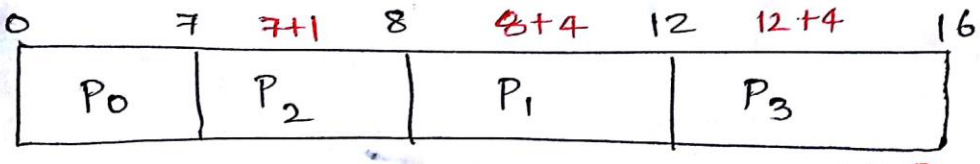
* Shortest Job First

* Non - preemptive

→ AT → Arrival time
BT → Burst time
CT → Completion time
TAT → Turnaround time
WT → Waiting time

Process	AT	BT	CT	TAT	WT
P_0	0	7	7	7	0
P_1	2	4	12	10	6
P_2	4	1	8	4	3
P_3	5	4	16	11	7
				tot: 32	tot: 16

Gantt chart:



$$\text{Avg (TAT)} = \frac{32}{4} = \boxed{8 \text{ msec.}}$$

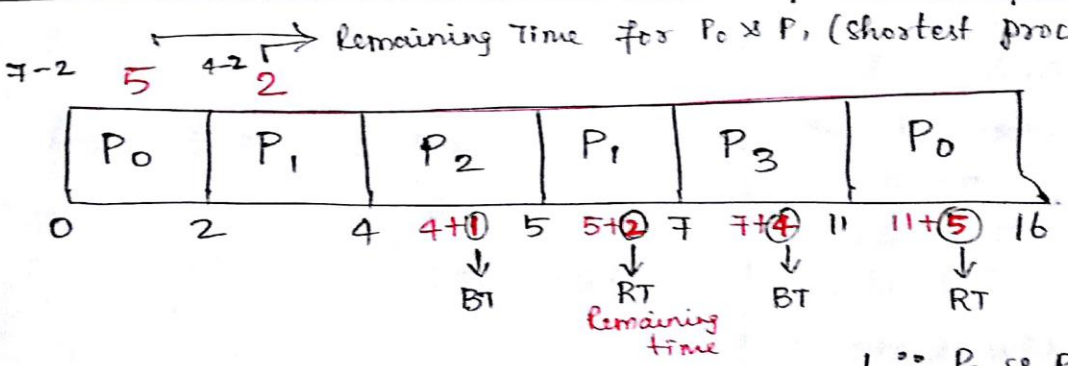
$$\text{Avg (WT)} = \frac{16}{4} = \boxed{4 \text{ msec.}}$$

∵ P_0 & P_1 are shortest processes.
∵ P_0 completed.
∵ P_1 P_2 P_3 are remaining in Queue
∵ P_1 P_2 P_3 → (4 1 4 ⇒ BT values)
∵ In the above processes check BT values.
∵ P_2 is small.
∵ remaining P_1 P_3 with same BT values
∵ Take small process as P_1 → first, P_3 → Last.

SRT:

* Shortest Remaining Time
* preemptive. (A process is pause the process & switching over to another process depending upon some criteria)

Process	AT	BT	CT	TAT (CT − AT)	WT (TAT − BT)
P_0	0	7	16	16	9
P_1	2	4	7	5	1
P_2	4	1	5	1	0
P_3	5	4	11	6	2



$$\text{Avg (TAT)} = \frac{28}{4} = \boxed{7 \text{ msec.}}$$

$$\text{Avg (WT)} = \frac{12}{4} = \boxed{3 \text{ msec.}}$$

∵ P_0 & P_1 are the two shortest processes.

∵ As given in the name we need to find remaining time (RT) for P_0 & P_1

⇒ P_0 = BT → 7, AT → 2 } 7 − 2 = 5

P_1 = BT → 4, AT → 2 } 4 − 2 = 2

∵ other P_2 & P_3 are calculated with BT.

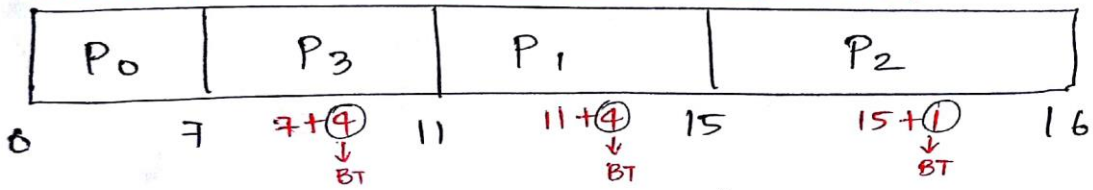
Priority Based Scheduling:

Process	Priority	AT	BT	CT	TAT (CT−AT)	WT (TAT−BT)
P_0	3	0	7	7	7	0
P_1	2	2	4	15	13	9
P_2	4	4	1	16	12	11
P_3	1	5	4	11	6	2

P_0 & P_3 ⇒ Low & High process.
1 → High
3 → Low
} check with process & priority values.

Gnatt chart:

Low | High



Avg (TAT) = 38/4 = 9.5 msec.

Avg (WT) = 22/4 = 5.5 msec.

∵ first finish off the P_0 process. since it is Low priority.

∵ Next High priority with P_1 P_2 P_3 processes.

∵ start with P_3 with BT value 4
↳ P_3 : 1 → High

∵ Next P_1 with BT value 4
↳ P_1 : 2 → High

∵ Next P_0 → we completed already.

∵ Next P_2 with BT value 4
↳ P_2 : 4 → High

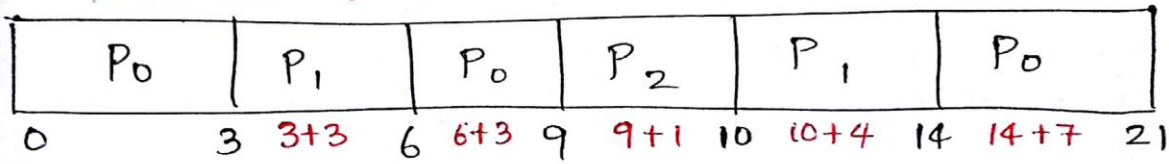
Round Robin based Scheduling:

Quantum value = 3 msec.
↓
Also called as time slice.

Process	AT	BT	CT	CT − AT = TAT	TAT − BT = WT
P_0	0	7	21	21	14
P_1	2	4	14	12	8
P_2	4	1	10	6	5

Quantum: Maximum time CPU given to the process at a single point of time.

Short process



$$\text{Avg (TAT)} = \frac{39}{4} = 9.75 \text{ msec.}$$

$$\text{Avg (WT)} = \frac{27}{4} = 6.75 \text{ msec.}$$

∵ P_0 & P_1 are short process add with the Quantum values.
∵ and after that all the three processes are added with the Burst time (BT) values.

DIFFERENCE BETWEEN PREEMPTIVE AND NON – PREEMPTIVVE

Preemptive Scheduling	Non-Preemptive Scheduling
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more compared to Non-Preemptive Scheduling.	CPU utilization is less compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.	When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Ex:- SRTF, Priority, Round Robin, etc.	Ex:- FCFS, SJF, Priority, etc.

SJF – SHORTEST JOB FIRST SCHEDULING:

- This is also known as shortest job next, or SJN
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

PRIORITY BASED SCHEDULING:

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement**.**

SRT – SHORTEST REMAINING TIME SCHEDULING:

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

ROUND ROBIN SCHEDULING:

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

MULTIPLE-LEVEL QUEUES SCHEDULING:

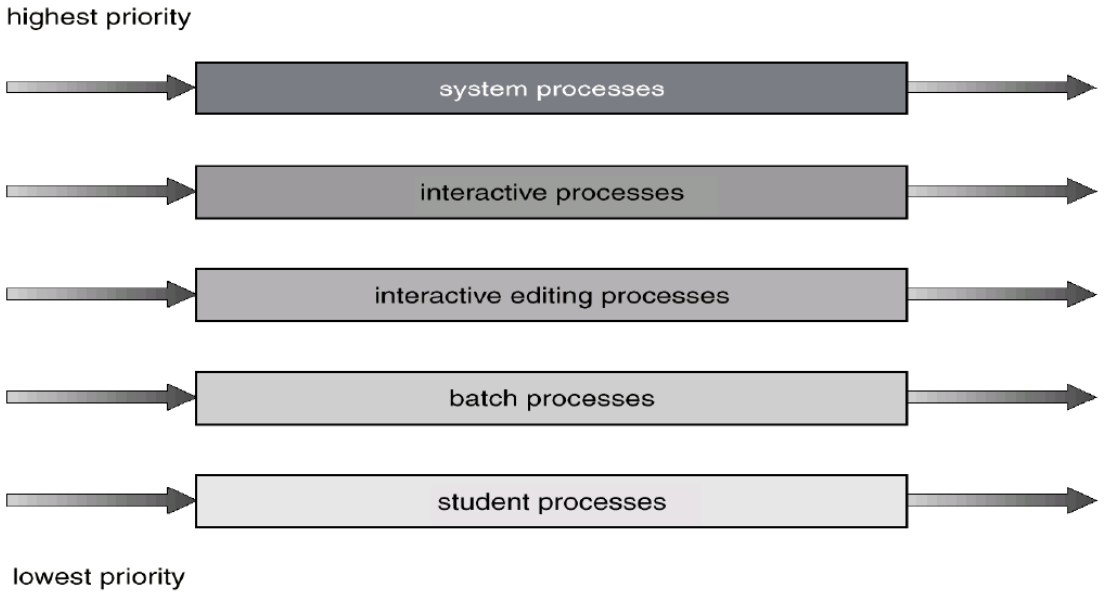


Fig 11 : Multiple level queues scheduling

- Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue**.**

MULTILEVEL FEEDBACK QUEUE SCHEDULING:

- Normally, in the multilevel queue scheduling algorithm, processes are permanently assigned to a queue when they enter to the system.
- The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of **AGING** prevents **STARVATION.**

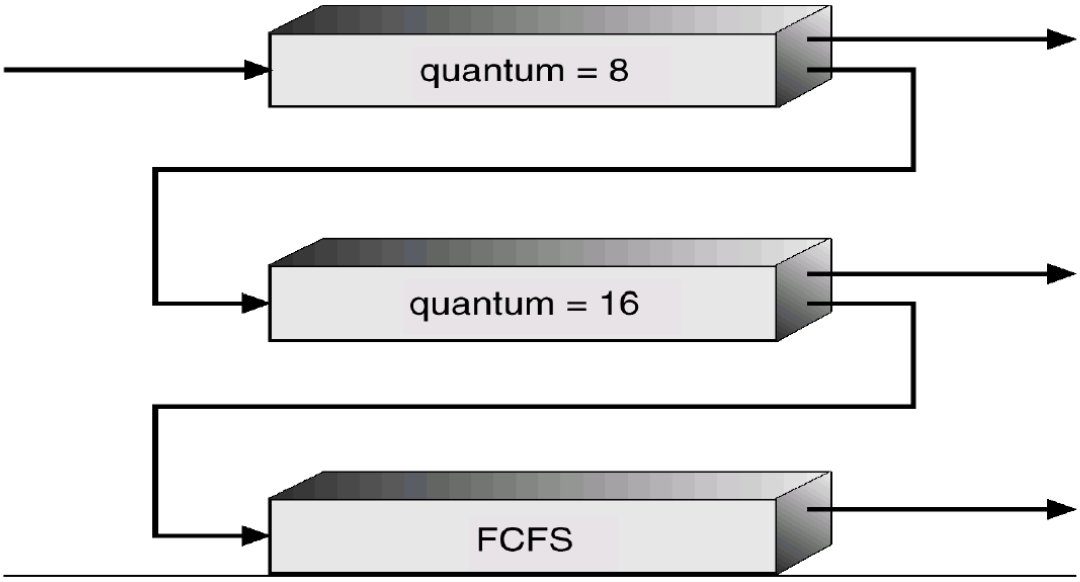


Fig 12 : Multilevel feedback queues

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING
Common to : CSE , IT

UNIT III

UNIT 3 SYNCHRONIZATION AND DEADLOCKS 9 Hrs.

The critical section problem - Semaphores - Classic problems of synchronization - Critical regions - Monitors-Dead locks - Deadlock characterization - Prevention - Avoidance - Detection - Recovery.

UNIT – III – SCS1301- OPERATING SYSTEM

INTRODUCTION

A cooperating process is one that can affect or be affected by other processes executing in the system. Co- operating processes can either directly share a logical address space or be allowed to share data only through files or messages. The former is achieved through the use of light weight processes or threads. Concurrent access to shared data may result in data inconsistency.

A bounded buffer could be used to enable processes to share memory. While considering the solution for producer – consumer problem, it allows at most BUFFER_SIZE – 1 items in the buffer. If the algorithm can be modified to remedy this deficiency, one possibility is to add an integer variable counter initialized to 0. Counter is incremented every time a new item is added to the buffer and is decremented every time, an item is removed from the buffer. Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

The code for the producer process can be modified as follows:

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

The code for the consumer process can be modified as follows:

```
while (true)
{
while (counter == 0)
; /* do nothing */ nextConsumed =
buffer [out] ;
out = (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in nextConsumed */
}
```

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter—" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately. We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as

*register*1 = counter
register1 = *register1* + 1
counter = *register1*
where *register1* is a local CPU register.

Similarly, the statement "counter—" is
implemented as follows:
register2 = counter
register2 = *register2* — 1
counter = *register2*
where again *register2* is a local CPU register.

Synchronization and Deadlocks

The concurrent execution of "counter++" and "counter—" is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each
high-level statement is preserved). One such interleaving is

T0: producer execute	*register1* = counter *{registeri = 5}*
T1: producer execute	*register 1* = *register1* + 1 *{registeri* = 6}
T2: consumer execute	*register2.* = counter *{register2* — 5}
T3: consumer execute	*register2* = *register2* — 1 *{register 2* — 4}
T4: producer execute	counter = *register1 {counter* = 6}
T5: consumer execute	counter = *register2 {counter* = 4}

When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called **race condition**. To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. Hence processes must be synchronized.

The Critical Section Problem

Each process in a system has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing a file etc. The important feature of the system is that when one process is executing in its critical section, no other process is to be allowed to execute in its critical section that is no two processes are executing in their critical sections at the same time. The critical section problem is to design a protocol that the processes can use to co- operate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

```
do
{
Entry Section
```

Critical

Section

Exit Section

Remainder Section

}while(TRUE);

General Structure of a typical process P_i.

A critical section is a piece of code that only one thread can execute at a time. If multiple threads try to enter a critical section, only one can run and the others will sleep. Imagine you have three threads that all want to enter a critical section. Only one thread can enter the critical section; the other two have to sleep. When a thread sleeps, its execution is paused and the OS will run some other thread. Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.

A solution to the critical section problem must satisfy the following three requirements:

Mutual exclusion: If a process is executing in critical section, then no other process can be executing in their critical section.

Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next and this selection cannot be postponed indefinitely.

Bounded waiting: There exists a bound or limit on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that request is granted.

Two Process Solutions

Algorithm 1:

do {

while (turn !=1); critical

section turn = j;

remainder section

}while (1);

This Algorithm satisfies Mutual exclusion whereas it fails to satisfy progress requirement since it requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and p1 is ready to enter its critical section,p1 cannot do so, even though p0 may be in its remainder section.

Algorithm 2:

do{

flag[i] =true; while

(flag[j]); critical section

flag[i]= false;

remainder section

}while(1);

In this solution, the mutual exclusion is met. But the progress is not met. To illustrate this problem, we consider the following execution sequence:

To: P0 sets Flag[0] = true T1: P1

sets Flag[1] = true

Now P0 and P1 are looping forever in their respective while statements.

Mutual exclusion

P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag [0] is true and either flag [1] is false (meaning P1 has left its critical section) or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting). In both cases, P1 cannot be in critical section when P0 is in critical section.

Progress

A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

Bounded waiting

A process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

Algorithm 3:

By combining the key ideas of algorithm 1 and 2, we obtain a correct solution.

```
do{
Flag[i] =true; Turn =
j;
While(flag[j] && turn==j);
Critical section
Flag[i] = false;
Remainder section
}while(1);
```

The algorithm does satisfy the three essential criteria to solve the critical section problem. The three criteria are mutual exclusion, progress, and bounded waiting.

Synchronization Hardware

Any solution to the critical section problem requires a simple tool called a lock. Race conditions are prevented by requiring that critical regions are protected by locks that is a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

```
do
{
```

```
Acquire Lock
        Critical Section Release Lock
Remainder Section
}while(TRUE);
```

Solution to the critical-section problem using locks.

Hardware features can make any programming task easier and improve system efficiency. The critical section problem can be solved simply in a uni processor environment if we could prevent interrupts from occurring while a shared variable was being modified. Then we can ensure that the current sequence of instructions would be allowed to execute in order without pre emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This solution is not feasible in a multi processor environment. Disabling interrupts on a multi processor can be time consuming, as the message is passed to all the processors.

```
boolean TestAndSet(boolean *target)
{
boolean rv=target;
*target = TRUE; return rv;
}
```

The definition of the TestAndSet() instruction.

```
do
{
while(TestAndSetLock(&lock))
;  // do nothing
// critical section lock = FALSE;
```

```
//remainder section
}while(TRUE);
```

Mutual-Exclusion Implementation with TestAndSet()

This message passing delays entry into each critical section and system efficiency decreases. Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** that is as one uninterruptable unit.

The TestAndSet () instruction can be defined as above. This instruction is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously each on a different CPU, they will be executed sequentially in some order.

The Swap () instruction operates on the contents of two words – Void

```
swap(boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp;
}
```

Definition of Swap Instruction

```
do
{
key=TRUE; while(key==TRUE)

swap(&lock, &key);
```

// critical section lock = FALSE;

// remainder section

}while(TRUE);

Mutual Exclusion Implementation with the Swap() function.

This is also executed atomically. If the machine supports Swap() instruction, then mutual exclusion can be provided by using a global Boolean variable lock initialized to false. Each process has a local Boolean variable key.

But these algorithms do not satisfy the bounded – waiting requirement.

The below algorithm satisfies all the critical section problems: Common data structures used in this algorithm are: Boolean waiting[n];

Boolean lock;

Both these data structures are initialized to false. For proving that the mutual exclusion requirement is met, we must make sure that process Pi can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the TestAndSet() is executed.

Semaphores

The various hardware based solutions to the critical section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that is accessed only through standard atomic operations: wait() and signal().

Wait:	Signal:
Wait(S)	signal(S)
{	{
While S<=0;	S++; S--; }

}

Modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

```
do{
        waiting(mutex);
                //critical section
        Signal(mutex);
                //remainder section
        }while(true);
```

Usage

OS's distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Binary semaphores are known as mutex locks as they are locks that provide mutual exclusion.

Binary semaphores are used to deal with the critical section problem for multiple processes. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore. When a process releases a resource, it performs a signal() operation.

Semaphores can be used to solve various synchronization problems. Simple binary semaphore example with two processes: Two processes are both vying for the single semaphore. Process A performs the acquire first and therefore is provided with the semaphore. The period in which the semaphore is owned by the process is commonly called a *critical section*. The critical section can be performed by only one process, therefore the need for the coordination provided by the semaphore. While Process A has the semaphore, Process B is not permitted to perform its critical section.

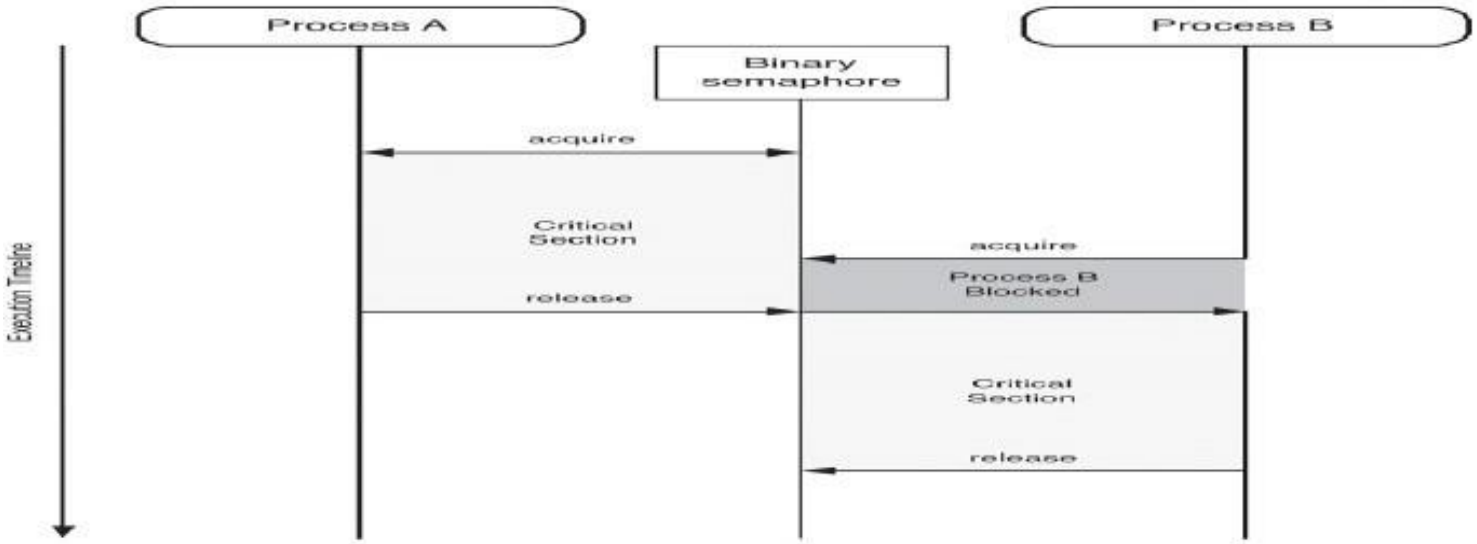


Fig.3.1. Binary Semaphore

Note that while Process A is in its critical section, Process B attempts to acquire the semaphore. As the semaphore has already been acquired by Process A, Process B is placed into a blocked state. When Process A finally releases the semaphore, it is then granted to Process B, which is allowed to enter its critical section. Process B at a later time releases the semaphore, making it available for acquisition.

In the counting semaphore example, each process requires two resources before being able to perform its desired activities. In this example, the value of the counting semaphore is 3, which means that only one process will be permitted to fully operate at a time. Process A acquires its two resources first, which means that Process B blocks until Process A releases at least one of its resources.

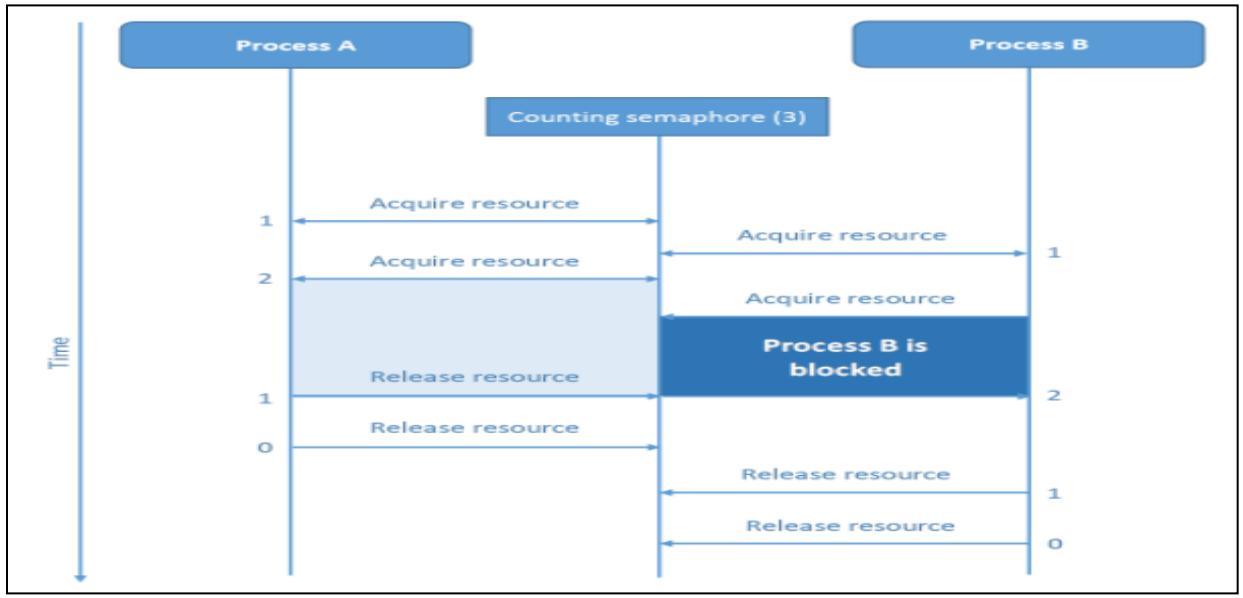


Fig.3.2. **Counting Semaphore**

Implementation

The main disadvantage of the semaphore is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is called a **spinlock** because the process spins while waiting for the lock. To overcome, the need for busy waiting the definition of wait () and signal() semaphore operations can be modified. When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. Then control is transferred to CPU scheduler which selects another process to execute.

A process that is blocked waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation which changes the process from the waiting state to the ready state. Process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct { int
value;
struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal () operation removes one process from the list of waiting processes and awakens that process. A signal() operation removes one process from the list of waiting processes and awakens that process.

Wait() semaphore operation can be defined as

```
wait(semaphore *S) { S-
>value—;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

Signal() semaphore operation can be defined as

```
signal(semaphore *S) {
```

```
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

The block operation suspends the process that invokes it. 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. Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation. The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list in a way that ensures bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists. The critical aspect of semaphores is that they be executed atomically- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the wait () and signal () operations are executing. This scheme works in a single processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques— such as spinlocks—to ensure that wait() and signal() are performed atomically. It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have removed busy waiting from the entry section to the critical sections of application programs.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked.**

To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that *P0* executes wait (S) and then P1 executes wait (Q). When Po executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until Po executes signal(S). Since these signal () operations cannot be executed, Po and P1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release.* Another problem related to deadlocks is **indefinite blocking,** or **starvation,** a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic problems of synchronization

These synchronization problems are examples of large class of concurrency control problems. In solutions to these problems, we use semaphores for synchronization.

The Bounded Buffer problem

Here the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n, the semaphore full is initialized to value 0.

The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

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

The structure of the Producer Process

```
do
{
wait(full); wait(mutex);
// remove an item from buffer into nextc signal(mutex);
```

signal(empty);
// consume the item in nextc
}while(TRUE);
The structure of the Consumer Process

The Readers – Writers Problem

A data base is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers) whereas others may want to update the database (writers). If two readers access the shared data simultaneously, no adverse effects will result. If a writer and some other thread, access the database simultaneously, problems occur.

To prevent these problems, writers have exclusive access to the shared database. This synchronization problem is referred to as readers – writers problem.

```
do {
    wait(wrt);
     ----------
    //Writing is Performed
   ----------
    signal(wrt);
     }while(True);
```

The Structure of a Writer Process

The simplest readers writers problem requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. No reader should wait for other readers to finish simply because a writer is waiting.

```
do {
   wait(mutex);
    readcount++;
    if(readcount==1)
       wait(wrt);
    signal(mutex);
     ------------
   // reading is performed
     -------------
    wait(mutex);
    readcount--;
    if(readcount==0)
         signal(wrt);
         signal(mutex);
} while(True);
```

The Structure of a Reader Process

The second readers writers problem requires that once a writer is ready, that writer performs its write as soon as possible, that is if a writer is waiting to access the object, no new readers may start reading.

In the solution to the first readers – writers problem, the reader processes share the following data structures: Semaphore
mutex, wrt;
int readcount;

Semaphores mutex and wrt are initialized to 1, readcount is initialized to 0. Semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical section.

The readers – writers problem and its solutions has been generalized to provide reader – writer locks on some systems. Acquiring the reader – writer lock requires specifying the mode of the lock, either read or write access. When a process only wishes to read shared data, it requests the reader – writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader – writer lock in read mode, only one process may acquire the lock for writing as exclusive access is required for writers.

Reader – writer locks are useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
- In applications that have more readers than writers.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chop sticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. A philosopher may pick up only one chopstick at a time. She cannot pick up a chopstick that is already in the hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chop sticks. When she is finished eating, she puts down both of her chop sticks and starts thinkingagain.

The dining philosopher's problem is considered a classic synchronization problem as it an example of a large class of concurrency control problems. One simple solution is to represent each chop stick with a semaphore.

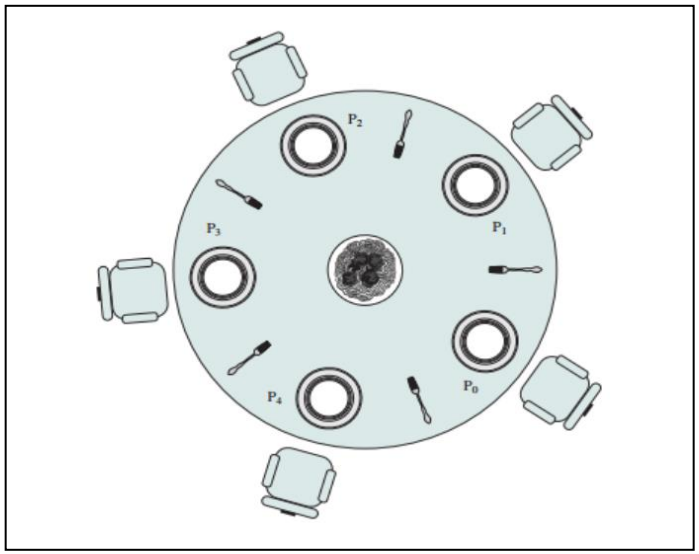


Fig.3.3 Dining Philosopher

A philosopher tries to grab a chop stick by executing a wait () operation on that semaphore; she releases her chop sticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are Semaphore chopstick [5]; where all elements of chopstick are initialized to 1.This solution is rejected as it could create a dead lock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chop stick. All the elements of chop stick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

```
do
{
wait(chopstick[i]); wait(chopstick[(i+ 1)
% 5] );
//eat signal(chopstick[i]);
signal(chopstick[(i+ 1) % 5] );
//think
}while(TRUE);
```

The structure of philosopher i.

Solution to dining philosophers problem:

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick whereas an even philosopher picks up her right chopstick and then her left chopstick.

Critical Regions

- A **critical region** is a section of code that is always executed under mutual exclusion.

- Critical regions shift the responsibility for enforcing mutual exclusion from theprogrammer (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
  1. Variables that must be accessed under mutual exclusion.
  2. A new language statement that identifies a critical regionin which thevariables are accessed.

Example:

```
var
v : shared T;
...
region v do
begin
...
end;
```

All critical regions that are 'tagged' with the same variable have compiler-enforced mutual exclusion so that only one of them can be executed at a time:

```
Process A:
region V1 do
begin
{ Do some stuff. } end;
region V2 do
begin
{ Do more stuff. }
end;

Process B:
region V1 do
begin
```

{ Do other stuff. }
end;
Here process A can be executing inside its V2 region while process Bis executing inside its V1 region, but if they both want to execute inside their respective V1 regions only one will be permitted to proceed.
Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

Conditional Critical Regions
Critical regions aren't equivalent to semaphores. They lack condition synchronization. We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can't do this with critical regions.

Conditional critical regions provide condition synchronization for critical regions:
region v when B do
begin
...
end;
where B is a boolean expression (usually B will refer to v).

Conditional critical regions work as follows:

1. A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.
2. Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest B.

Implementation
Each shared variable now has two queues associated with it. The **main queue** is for processes that want to enter a critical region but find it locked. The **event queue** is for the processes that have blocked because they found the condition to be false. When a process leaves the conditional critical

region the processes on the event queue join those in the main queue. Because these processes must retest their condition they are doing something akin to busy-waiting, although the frequency with which they will retest the condition is much less. Note also that the condition is only retested when there is reason to believe that it may have changed (another process has finished accessing the shared variable, potentially altering the condition). Though this is more controlled than busy-waiting, it may still be sufficiently close to it to be unattractive.

Limitations

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation. Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect since these errors happen only if some particular execution sequences take place and these sequences do not always occur. Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed.

Signal (mutex);

…….

Critical section

……..

Wait (mutex);

Here several processes may be executing in their critical sections simultaneously, violating the mutual exclusion requirement.

* Suppose that a process replaces signal (mutex) with wait (mutex) that is it executes

Wait(mutex);

……

Critical section

……. Wait(mutex);

Here a deadlock will occur.

* Suppose that a process omits the wait(mutex), or the signal(mutex) or both. Here, either mutual exclusion is violated or a dead lock will occur.

To deal with such errors, a fundamental high level synchronization construct called **monitor** type is used.

Usage

A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of the procedures or functions that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. The local variables of a monitor can be accessed by only the local procedures.

Name:monitor

…..local declarations

…..initialize local data

Proc1(….parameters)

……statement list

Proc2(…parameters)

…….statement list

Proc3(…parameters)

…….statement list

The monitor construct ensures that only one process at a time can be active within the monitor. But this monitor construct is not powerful for modeling some synchronization schemes. For this we need additional synchronization mechanisms. These mechanisms are provided by condition construct. The only operations that can be invoked on a condition variable are wait() and signal(). The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal(). The x.signal() operation resumes exactly one suspended process.

When x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x. If suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor

However, both processes can conceptually continue with their execution. Two possibilities exist:

- Signal and wait – P either waits until Q leaves the monitor or waits for another condition.
- Signal and condition – Q either waits until P leaves the monitor or waits for another condition.

Dining Philosophers Solution using Monitors

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we use this data structure enum {thinking, hungry, eating } state[5];

Philosopher i can set the variable state[i] = eating only if her two neighbors are not eating: (state [(i+4) % 5]! = eating) and (state [(i+1)%5]!=eating) Also declare condition self [5]; where philosopher i can delay herself when she is hungry but is unable to obtain the chop sticks she needs. The distribution of the chopsticks is controlled by the monitor dp. Each philosopher before starting to eat, must invoke the operation pickup().

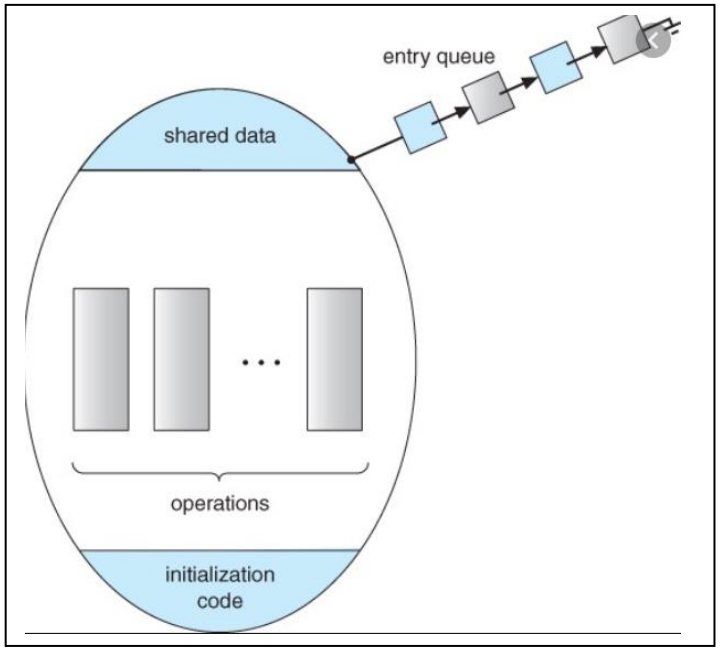


Fig.3.4. Monitors

This may result in the suspension of the philosopher process. After the successful completion of the operation the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus philosopher i must invoke the operations pickup() and putdown() in the following sequence:

dp.pickup(i);
……. Eat
…….
dp.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor using Semaphores

For each monitor, a semaphore mutex initialized to 1 is provided. A process must execute wait(mutex) before entering the monitor and must execute(signal) after leaving the monitor. Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore next initialized to 0, on which the signaling processes may suspend themselves. An integer variable next_count is used to count the number of processes suspended on next. Thus, each external procedure P is replaced by wait(mutex);

body of F

if (next_count > 0) signal(next);

else

signal(mutex);

Mutual exclusion within a monitor is thus ensured.

We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x. wait () can now be implemented as x_count++;

if (next_count > 0)

signal(next);

else signal(mutex);

wait(x_sem);

x_count—;

The operation x. signal () can be implemented as if

(x_count > 0) {

next_count++;

signal(x_sem); wait(next) ;

next_count—;

}

Resuming Processes Within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then for determining which suspended process should be resumed next, we use FCFS ordering so that the process waiting the longest is resumed first. Or conditional wait construct() can be used as x.wait(c); where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number,** is then stored with the name of the process that is suspended. When x. signal () is executed, the process with the smallest associated priority number is resumed next.

```
monitor ResourceAllocator
boolean busy;
condition x;
void acquire(int time)
 if (busy)
x.wait(time);
busy = TRUE;
void release() {
busy = FALSE;
 x.signal();
initialization_code busy = FALSE;
```

A monitor to allocate a single resource

we consider the Resource Allocator monitor shown in which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest timeallocation request. A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

access the resource;

R. release () ;

where R is an instance of type ResourceAllocator. Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- Synchronization Example
- A process might attempt to release a resource that it never request.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores.Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource access operations within the ResourceAllocator monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time- dependent errors will occur and that the scheduling algorithm will not be defeated. Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only by additional mechanisms.

Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. For Example

System has 2 tape drives.

P_0 and P_1 each hold one tape drive and each needs another one, semaphores *A* and *B*, initialized to 1

P_0	*P1*
wait (A);	*wait(B);*
wait (B);	*wait(A);*

Bridge Crossing Example

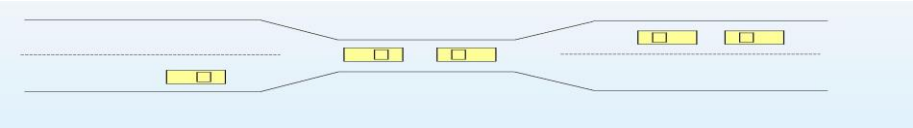


Fig.3.5. Bride Crossing Problem

Traffic only in one direction. Each section of a bridge can be viewed as a resource. If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).Several cars may have to be backed up if a deadlock occurs. Starvation is possible.

System Model

1. Resource types $R_1, R_2, \ldots, R_m$
2. *CPU cycles, memory space, I/O devices*

3. Each resource type R_i has W_i instances. Each process utilizes a resource as follows:

- request
- use
- release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

A set of vertices V and a set of edges E. V

is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

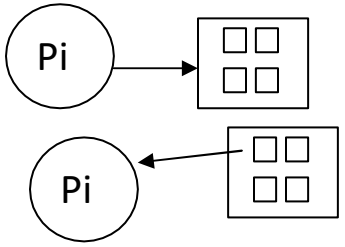
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

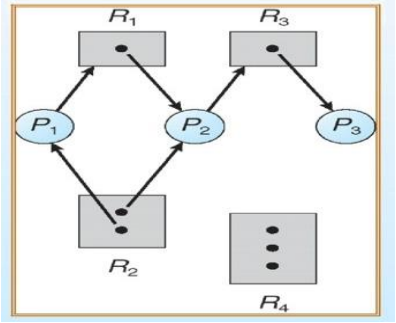
Request edge – directed edge $P_1 \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

- Process

- Resource Type with 4 instances
- Pi Requests Instance Of Rj
- Pi is Holding An Instance Of Rj





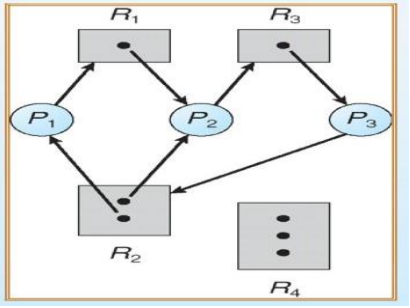


Fig.3.6. Resource Allocation Graph

Basic Facts

Synchronization and Deadlocks

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle if only one instance per resource type, then deadlock. if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Deadlock Prevention or Avoidance- Ensure that the system will *never* enter a deadlock state
- Deadlock Detection- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources. Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none. Low resource utilization; starvation possible.
- **No Preemption** –If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. Preempted resources are added to the list of resources for which the process is waiting. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available. Simplest and most useful model requires that each process

declare the *maximum number* of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.System is in safe state if there exists a safe sequence of all processes.
- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j<I.
- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

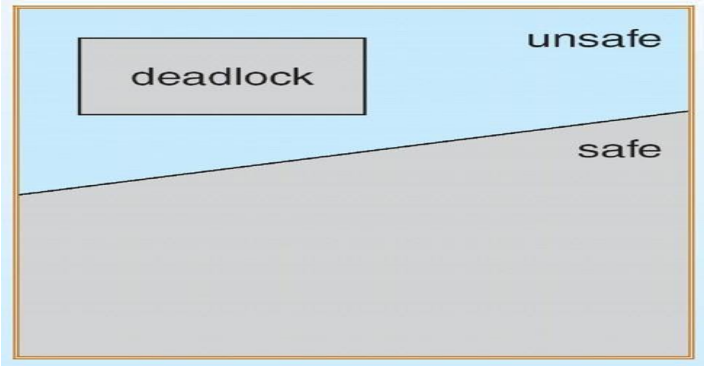


Fig.3.7. Safe and Unsafe State

Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

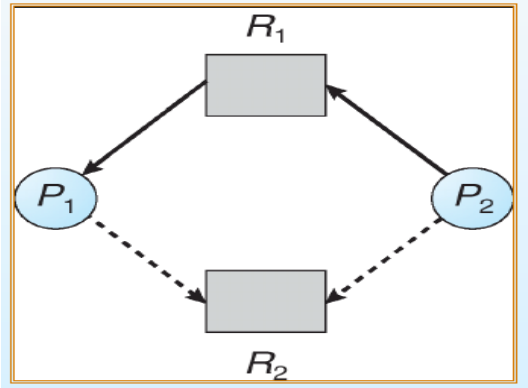


Fig.3.8. Resource Allocation Graph for Deadlock Avoidance

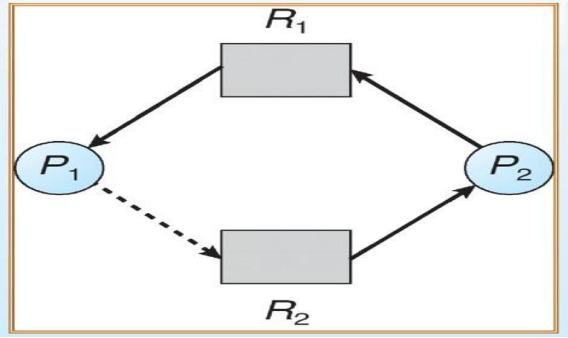


Fig.3.9. Unsafe State In Resource-Allocation Graph

Example formal algorithms

- Banker's Algorithm
- Resource-Request Algorithm
- Safety Algorithm

Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available. *Max: n x m* matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j. *Allocation: n* x *m* matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.

Need: n x *m* matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i,j] = *Max*[i,j] – *Allocation* [i,j]

Safety Algorithm

1, Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:

Work = Available

Finish [*i*] = *false* for *i* - 1,3, …, *n.*

2, Find and *i* such that both:

Finish [*i*] = *false Need*$_i \leq$

Work

If no such *i* exists, go to step 4.

3, Work = Work + Allocation$_i$
Finish[*i*] = *true*

go to step 2.

4, If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

- If *Request*$_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j.
- If *Request*$_i \leq$ *Need*$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- If *Request*$_i \leq$ *Available*, go to step 3. Otherwise P_i must wait, since resources are not available.

Pretend to allocate requested resources to P_i by modifying the state as follows:

Available = Available = Request$_i$*;*

Allocation$_i$ *= Allocation*$_i$ *+ Request*$_i$*; Need*$_i$ =

Need$_i$ *– Request*$_i$

If safe ⇒ the resources are allocated to Pi.

If UNsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5instances), and C (7 instances).

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The content of the matrix. Need is defined to be Max – Allocation.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
- Nodes are processes.
- $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

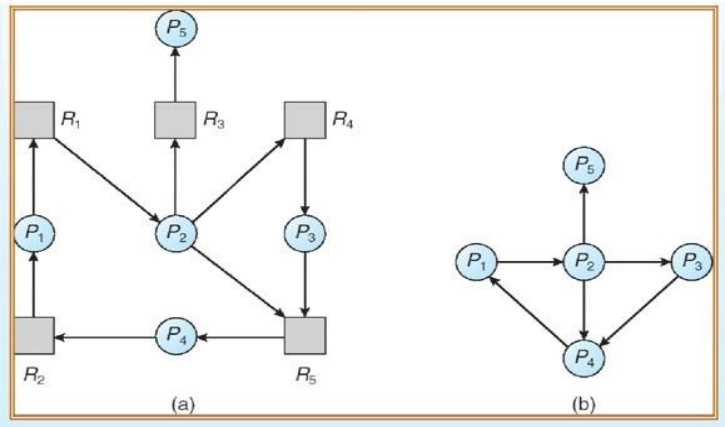


Fig.3.10 Resource-Allocation Graph and Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.
- *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1, Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

a, *Work* = *Available*

b, For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.

2, Find an index *i* such that both:

a, *Finish*[*i*] == *false*

b, $Request_i \leq Work$

If no such *i* exists, go to step 4 3, *Work* =

Work + $Allocation_i$

Finish[*i*] = *true*

go to step 2.

4, If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- SnapShot at Time T0

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- P_2 requests an additional instance of type *C*.

Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests. Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

When, and how often, to invoke depends on:

1. How often a deadlock is likely to occur?
2. How many processes will need to be rolled back?

   one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources the process has used. Resources process needs to complete.
- How many processes will need to be terminated.
- Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

Common to : CSE , IT

UNIT IV

UNIT 4 MEMORY MANAGEMENT **9 Hrs.**

Storage Management Strategies - Contiguous Vs. Non-Contiguous Storage Allocation - Fixed & Variable Partition Multiprogramming - Paging - Segmentation - Paging/Segmentation Systems - Page Replacement Strategies - Demand & Anticipatory Paging - File Concept - Access Methods - Directory Structure - File Sharing - Protection - File - System Structure - Implementation.

UNIT – IV – SCS1301- OPERATING SYSTEM

STORAGE / MEMORY MANAGEMENT:

- Memory management is the functionality of an operating system which handles or manages primary memory.
- Memory management keeps track of each and every memory location either it is allocated to some process or it is free.
- It checks how much memory is to be allocated to processes. It decides which process will get memory at what time.
- It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Memory management provides protection by using two registers, a base register and a limit register.
- The base register holds the smallest legal physical memory address and the limit register specifies the size of the range.
- For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999

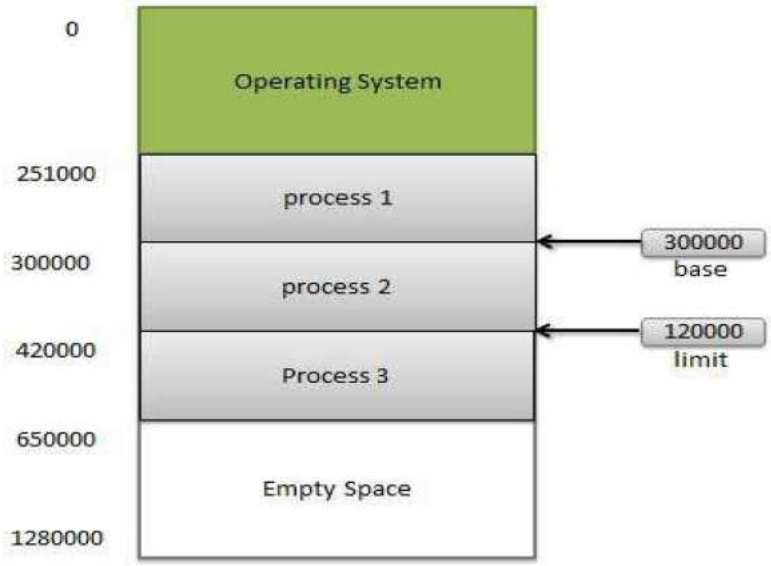


Fig 1 : Memory Management

GOALS OF MEMORY MANAGEMENT:

- Allocate available memory efficiently to multiple processes
- Main functions
- Allocate memory to processes when needed
- Keep track of what memory is used and what is free
- Protect one process's memory from another

MEMORY ALLOCATION STRATEGIES:

- **CONTIGUOUS ALLOCATION :**
  - In contiguous memory allocation each process is contained in a single contiguous block of memory.
  - Memory is divided into several fixed size partitions.
  - Each partition contains exactly one process.
  - When a partition is free, a process is selected from the input queue and loaded into it.
  - The free blocks of memory are known as holes.
  - The set of holes is searched to determine which hole is best to allocate.

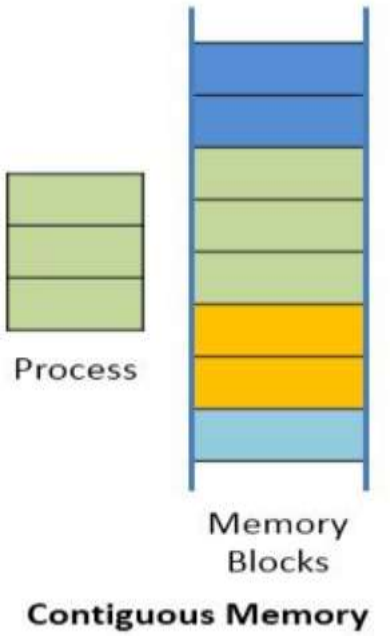


Fig 2 : Contiguous memory

- **NON-CONTIGUOUS ALLOCATION:**
  - Parts of a process can be allocated noncontiguous chunks of memory.

- In context to memory organization, non contiguous memory allocation means the available memory space is scattered here and there it means all the free available memory space is not together at one place.

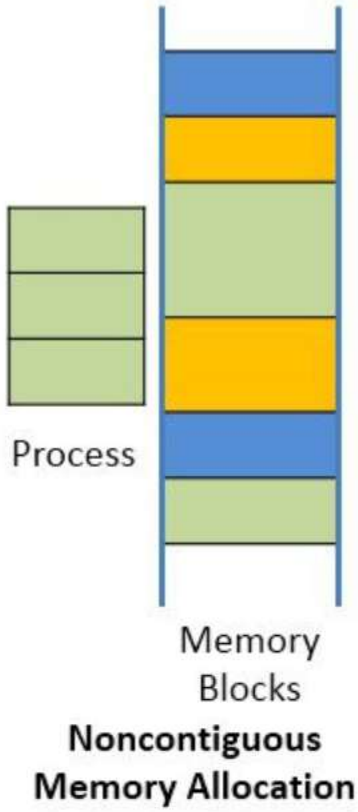
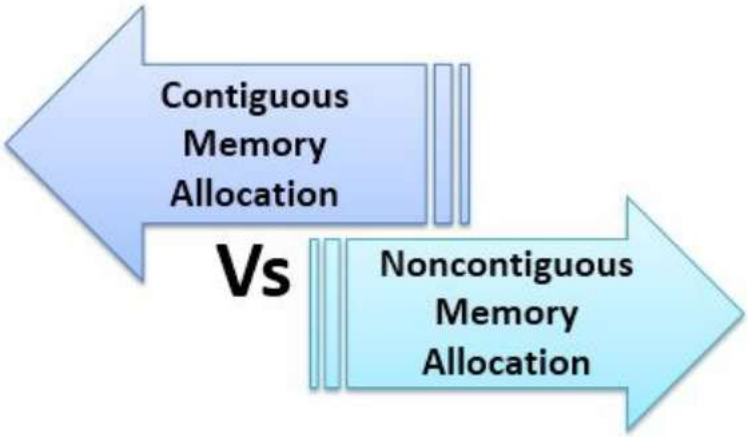


Fig 3 : Non-contiguous memory

CONTAGIOUS Vs NON CONTAGIOUS



CONTIGUOUS MEMORY ALLOCATION	NONCONTIGUOUS MEMORY ALLOCATION
Allocates consecutive blocks of memory to a process.	Allocates separate blocks of memory to a process.
Contiguous memory allocation does not have the overhead of address translation while execution of a process.	Noncontiguous memory allocation has overhead of address translation while execution of a process.
A process executes fatser in contiguous memory allocation	A process executes quite slower comparatively in noncontiguous memory allocation.
The memory space must be divided into the fixed-sized partition and each partition is allocated to a single process only.	Divide the process into several blocks and place them in different parts of the memory according to the availability of memory space available.

Table 1 : Contiguous vs Non-Contiguous

FIXED PARTITION SCHEME:

- Memory is broken up into fixed size partitions
- But the size of two partitions may be different
- Each partition can have exactly one process
- When a process arrives, allocate it a free partition
- Easy to manage
- Problems - Maximum size of process bound by max. partition size, Large internal fragmentation possible

VARIABLE PARTITION SCHEME

- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
    - allocated partitions
    - free partitions (hole)

MULTIPROGRAMMING:

- To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced.
- The multiprogramming is interleaved execution of multiple jobs by the same computer.
- In multiprogramming system, when one program is waiting for I/O transfer, there is another program ready to utilize the CPU.
- SO it is possible for several jobs to share the time of the CPU.
- But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time.
- Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on.

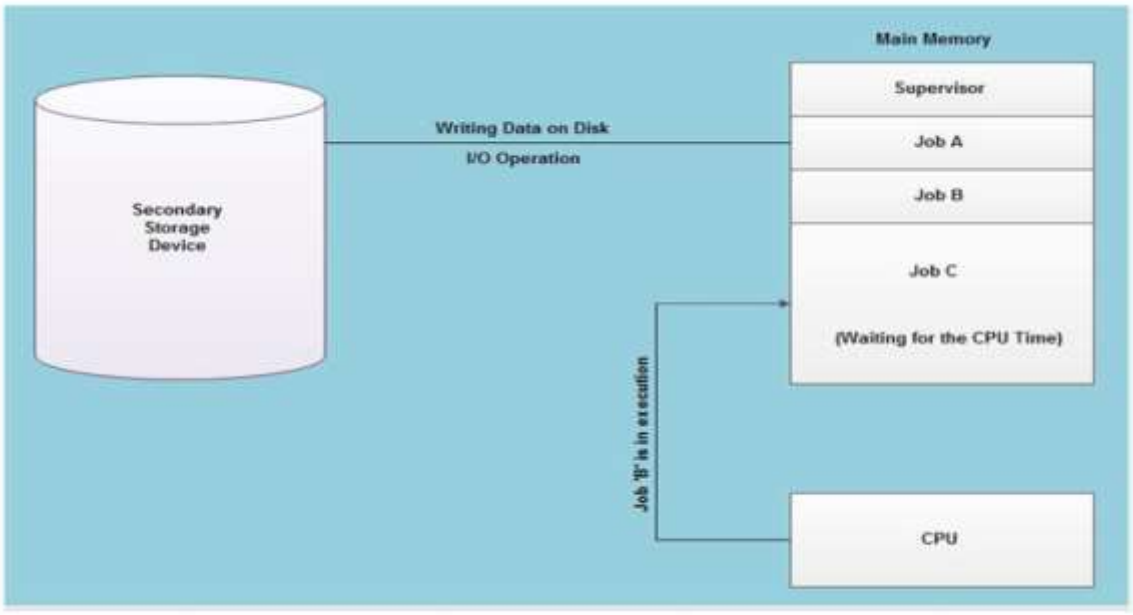


Fig 4 : Multiprogramming

- At the particular situation, job' A' is not utilizing the CPU time because it is busy in I/ 0 operations.
- Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time.
- So in this state the CPU will never be idle and utilizes maximum of its time.
- A program in execution is called a "Process", "Job" or a "Task".
- The concurrent execution of programs improves the utilization of system resources and enhances the system throughput as compared to batch and serial processing.
- In this system, when a process requests some I/O to allocate; meanwhile the CPU time is assigned to another ready process.
- So, here when a process is switched to an I/O operation, the CPU is not set idle.
- Multiprogramming is a common approach to resource management.
- The essential components of a single-user operating system include a command processor, an input/ output control system, a file system, and a transient area.
- A multiprogramming operating system builds on this base, subdividing the transient area to hold several independent programs and adding resource management routines to the operating system's basic functions.

STORAGE HIERARCHY

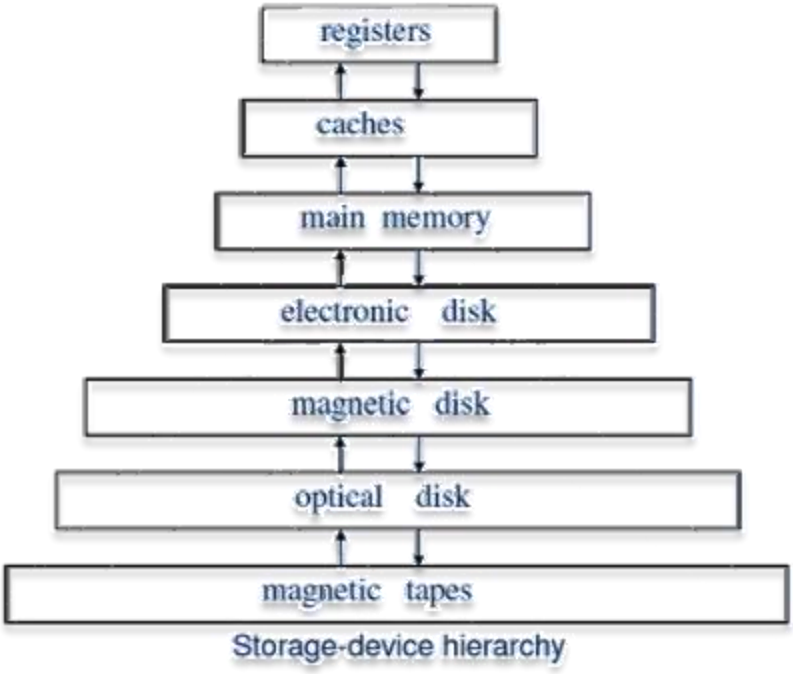


Fig 5 : Storage Hierarchy

- The wide variety of storage systems in a computer system can be organized in a hierarchy according to their speed and their cost.
- The higher levels are expensive but fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time increases.
- The reason for using the slower memory decives is that they are cheaper than the faster ones.
- Many early storage devices, including paper tape and core memories, are found only in museums now that magnetic tape and semiconductor memory have become faster and cheaper.

DYNAMIC LOADING

- In dynamic loading, a routine of a program is not loaded until it is called by the program.
- All routines are kept on disk in a re-locatable load format.
- The main program is loaded into memory and is executed.
- Other routines methods or modules are loaded on request.
- Dynamic loading makes better memory space utilization and unused routines are never loaded.

DYNAMIC LINKING

- Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed.
- Operating system can link system level libraries to a program.
- When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

LOGICAL vs PHYSICAL ADDRESS SPACE:

- An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address.
- Logical address is also known a Virtual address.
- Virtual and physical addresses are the same in compile-time and load-time address-binding schemes.
- Virtual and physical addresses differ in execution-time address-binding scheme.

LOGICAL ADDRESS	PHYSICAL ADDRESS
It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
The user can view the logical address of a program.	The user can never view physical address of program
The user uses the logical address to access the physical address.	The user can not directly access physical address.
The Logical Address is generated by the CPU	Physical Address is Computed by MMU

Table 2 : Logical vs Physical Address

SWAPPING

- Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store, and then brought back into memory for continued execution.
- Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users.
- It must be capable of providing direct access to these memory images.
- Major time consuming part of swapping is transfer time.
- Total transfer time is directly proportional to the amount of memory swapped.
- Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second.
- The actual transfer of the 100K process to or from memory will take

- 100KB / 1000KB per second = 1/10 second = 100 milliseconds

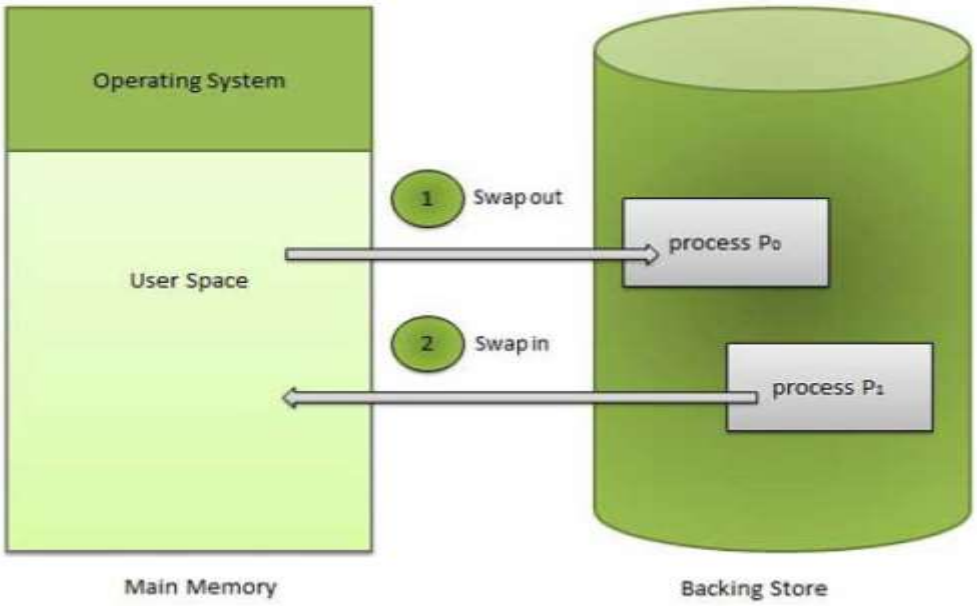


Fig 6 : Swapping

MULTIPLE-PARTITION ALLOCATION

FIXED PARTITION ALLOCATION

- One memory allocation method is to divide the memory into a number of fixed-size partitions.
- Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

DYNAMIC ALLOCATION

- The operating system keeps a table indicating which parts of memory is available (called holes) are available and which are occupied.
- Initially, all memory is available for user processes (i.e., there is one big hole).
- When a process arrives, we select a hole which is large enough to hold this process.
- We allocate as much memory is required for the process and the rest is kept as a hole which can be used for later requests. Selection of a hole to hold a process can follow the following algorithms

- **FIRST-FIT:**
  - Allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or where the previous first-fit search ended. There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request.
- **BEST-FIT:**
  - Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- **WORST-FIT:**
  - Allocate the largest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the largest leftover hole.
  - The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that compared to best fit, another process can use the remaining space.

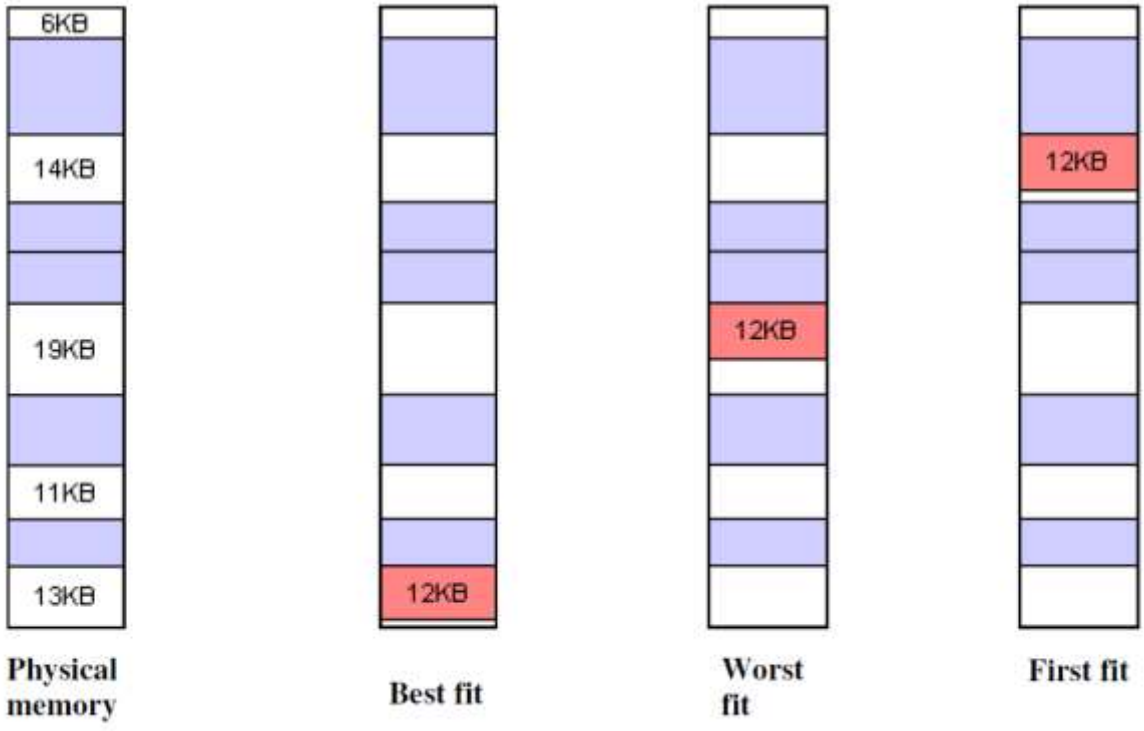


Fig 7 : Best, Worst, First Fit

FRAGMENTATION:

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

OR

The user of a computer continuously load and unload the processes from main memory. Processes are stored in blocks of the main memory. When it happens that there are some free memory blocks but still not enough to load the process, then this condition is called fragmentation.

Fragmentation is a condition that occurs when we dynamically allocate the RAM to the processes, then many free memory blocks are available but they are not enough to load the process on RAM.

TYPES:

- **External fragmentation**
- **Internal fragmentation**

EXTERNAL FRAGMENTATION:

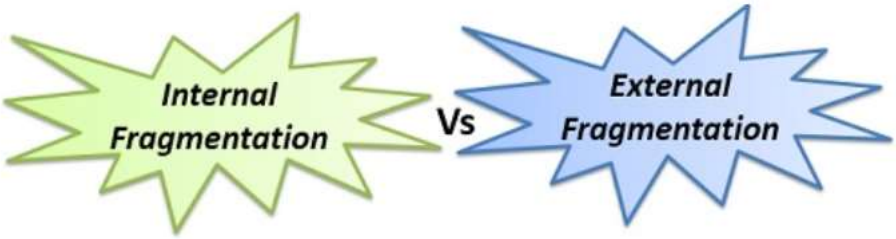
- External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used.
- If too much external fragmentation occurs, the amount of usable memory is drastically reduced.
- Total memory space exists to satisfy a request, but it is not contiguous.
- The problem of fragmentation can be solved by **COMPACTION.**
- The goal is to shuffle the memory contents to place all free memory together in one large block.
- For a relocated process to be able to execute in its new location, all internal addresses (e.g., pointers) must be relocated.
- If the relocation is static and is done at assembly or load time, compaction cannot be done.
- Compaction is possible only if relocation is dynamic and is done at execution time.
- If addresses are relocated dynamically, relocation requires only moving the program and data, and then changing the base register to reflect the new base address.

There may be many compaction algorithms:

- Simply move all processes toward one end of the memory; all holes move in the other direction producing one large hole of available memory.
- Create a large hole big enough anywhere to satisfy the current request.

INTERNAL FRAGMENTATION

- Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks.
- Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.



INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

Table 3 : Internal vs External fragmentation

PAGING

- Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.
- This scheme permits the physical address space of a process to be non – contiguous.
- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space(represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address actually available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses
- Physical memory is broken into fixed-size blocks called **FRAMES**.
- Logical memory is also broken into blocks of the same size called **PAGES.**

- When a process is to be executed, its pages (which are in the backing store) are loaded into any available memory frames. Thus the pages of a process may not be contiguous.

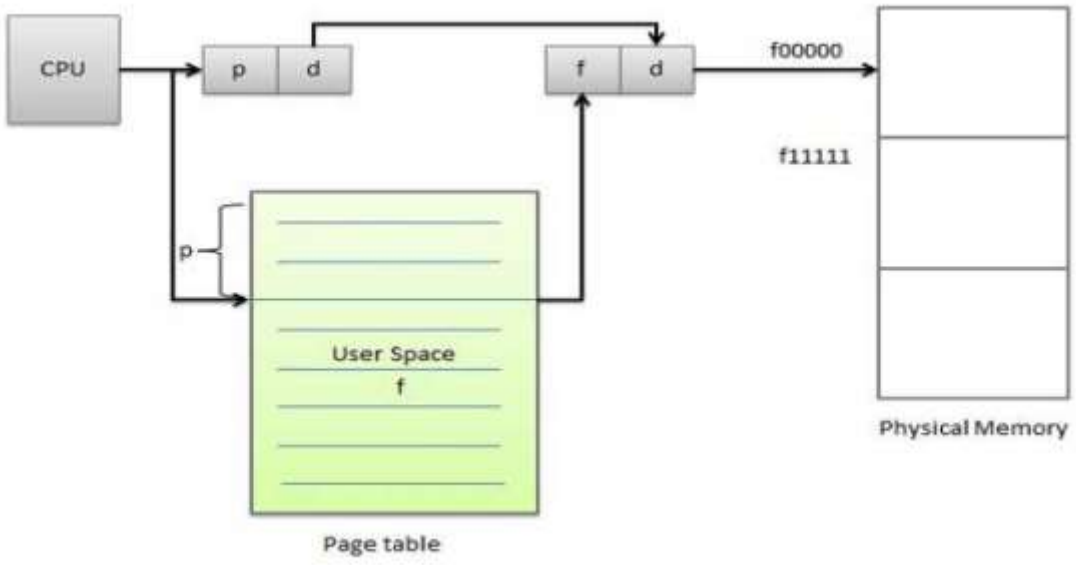


Fig 8 : Paging

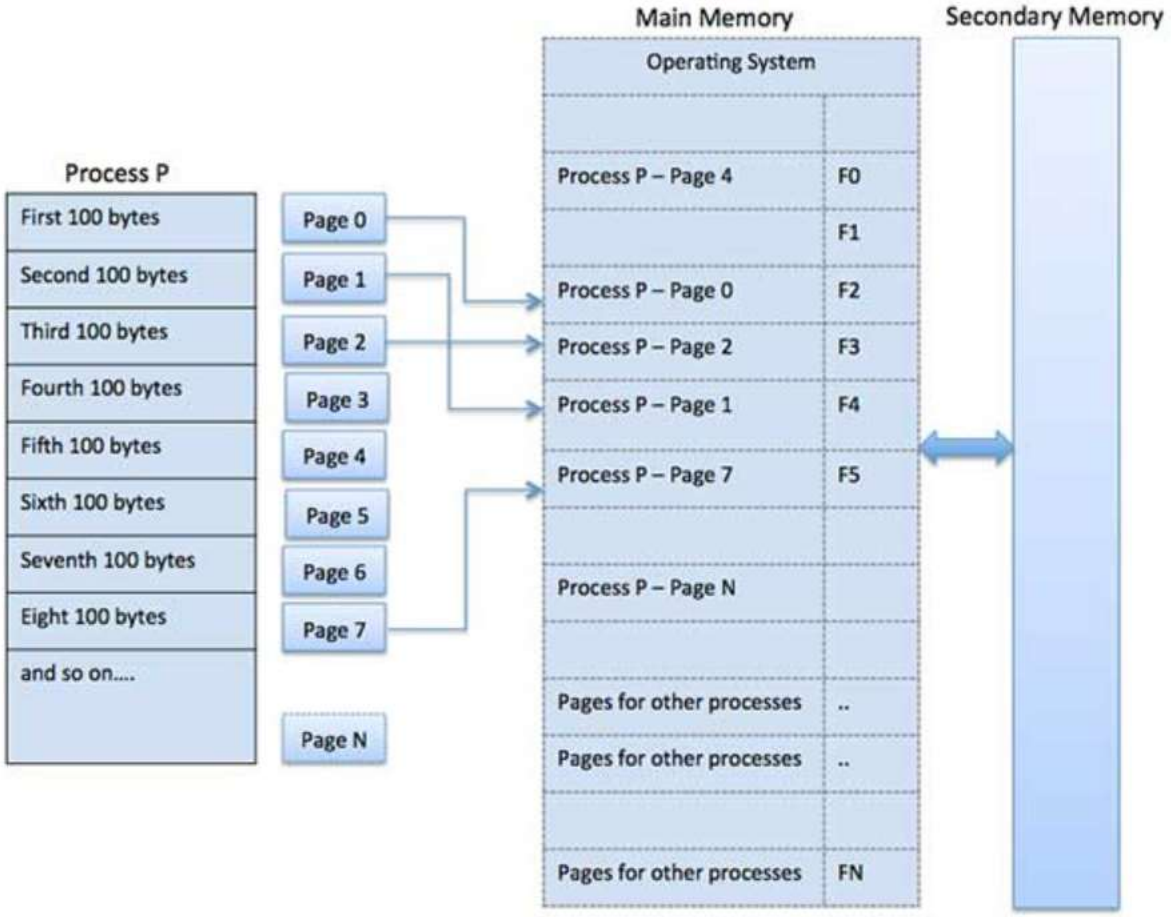


Fig 9 : Paging

ADDRESS TRANSLATION

- Page address is called **logical address** and represented by **page number** and the **offset**.
    - Logical Address = Page number + page offset
- Frame address is called **physical address** and represented by a **frame number** and the **offset**.
    - Physical Address = Frame number + page offset
- Paging eliminates external fragmentation altogether but there may be a little internal fragmentation.

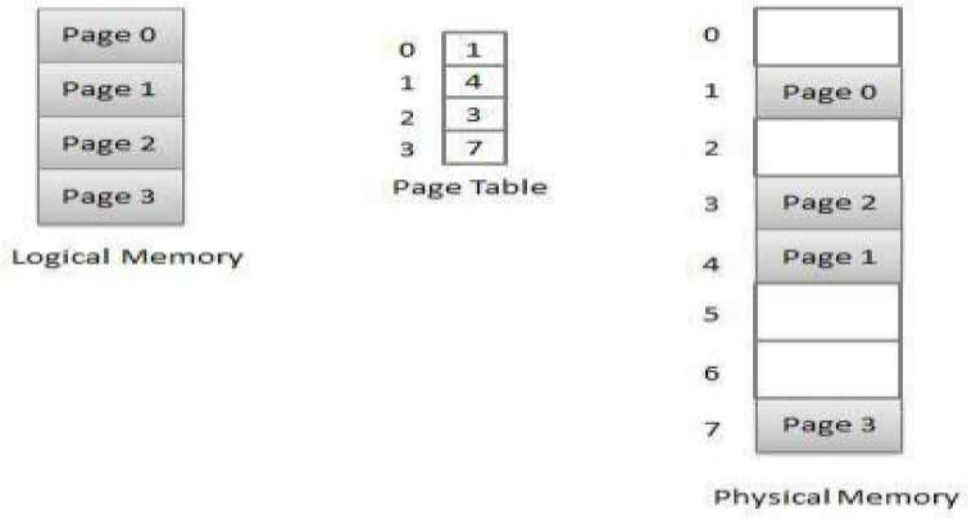


Fig 10 : Address Translation

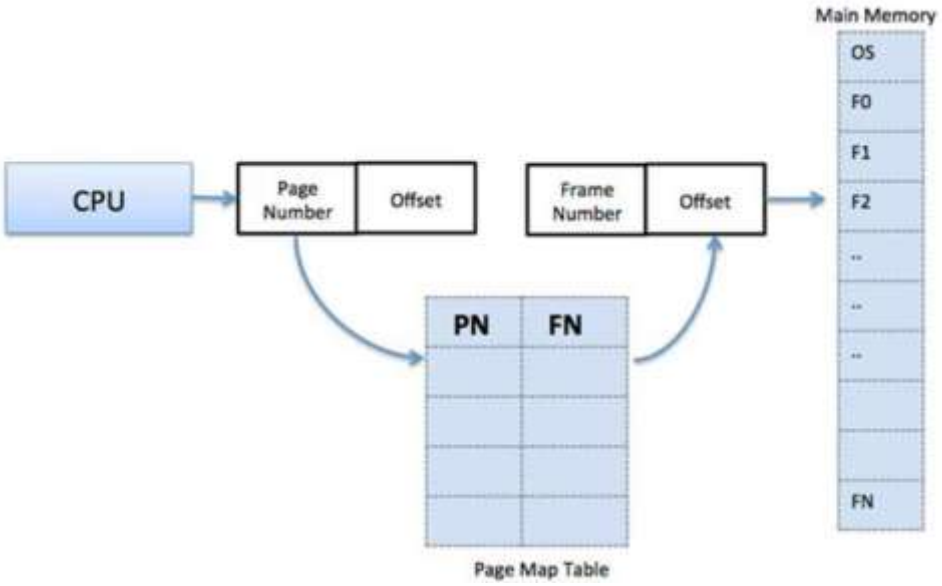


Fig 11 : Address translation

ADVANTAGES AND DISADVANTAGES OF PAGING

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

MULTILEVEL PAGING

- Most computer systems support a very large logical address space (232 to 264). In such a case, the page table itself becomes very very large.
- E.g., consider a 32-bit logical address space. If the page size is 4K bytes (212), then a page table may consist of up to (232 / 212) = 1 million entries. If each entry consists of 4 bytes, each process may need 4 megabytes of physical address alone for the page table
- **DISADVANTAGE**: Page tables consume a large amount of physical memory because each page table can have millions of entries.

INVERTED PAGE TABLE

- To overcome the disadvantage of page tables given above, an inverted page table could be used. An inverted page table has one entry for each (frame) of memory.
- Each entry consists of the logical (or virtual) address of the page stored in that memory location, with information about the process that owns it.
- Thus there is only one inverted page table in the system, and it has only one entry for each frame of physical memory.
- **DISADVANTAGE:**
    - The complete information about the logical address space of a process, which is required if a referenced page is not currently in memory is no longer kept.
    - To overcome this, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table, containing information on where each logical page is located.
    - These external page tables need not be in the memory all the time, because they are needed only when a page fault occurs.

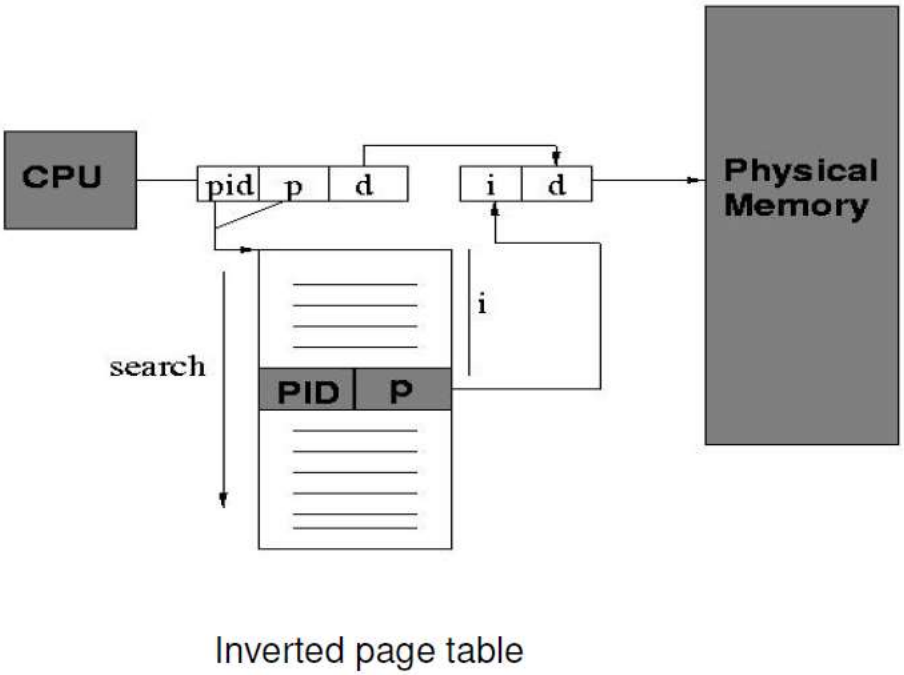


Fig 12 : Inverted page table

PROTECTION

- Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, they are kept in the page table.
- One bit can define a page to be read-and-write or read-only.

SHARED PAGES

- Another advantage of paging is the possibility of sharing common code.
- Consider a system that supports 40 users, each of which executes a text editor.
- If the text editor consists of 150K of code and 50K of data space, then we would need 8000K to support the 40 users.
- If the code is reentrant (non-self-modifying), then it can be shared between the 40 users.

SEGMENTATION:

- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.
- Each segment is actually a different logical address space of the program.
- When a process is to be executed, its corresponding segmentations are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
- A program segment contains the program's main function, utility functions, data structures, and so on.
- The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory.

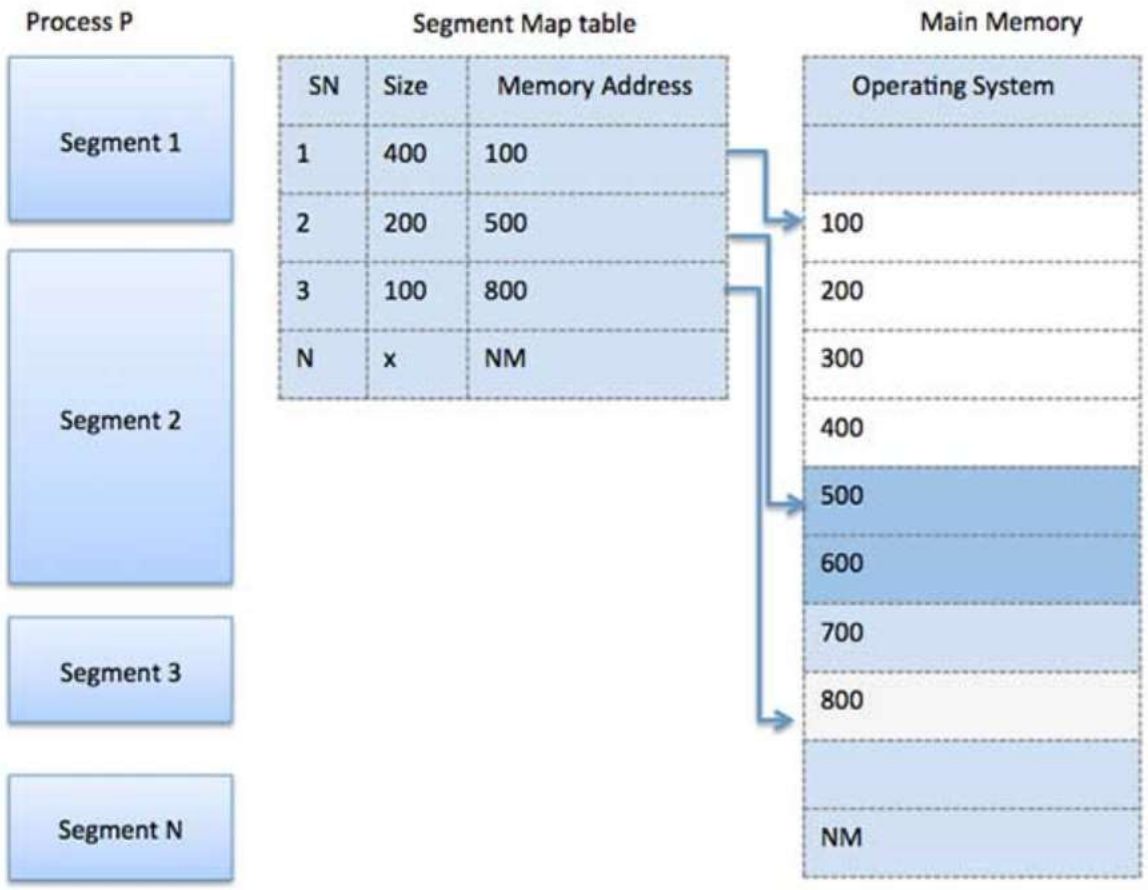


Fig 13 : Segmentation

- Segmentation is a memory-management scheme that suggests that a logical address space be divided into a collection of segments. Each segment has a name and a length.
- Addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: a segment name (or segment number) and an offset.
- **SEGMENT NUMBER (S)** -- segment number is used as an index into a segment table which contains base address of each segment in physical memory and a limit of segment.
- **SEGMENT OFFSET (O)** -- segment offset is first checked against limit and then is combined with base address to define the physical memory address.
- Therefore logical addresses consist of **two tuples**:
    - **<segment-number, offset>**

TUPLES:

- In the context of relational databases, a tuple is one record (one row).
- The information in a database can be thought of as a spreadsheet, with columns (known as fields or attributes) representing different categories of information, and tuples (rows) representing all the information from each field associated with a single record.

SEGMENTATION HARDWARE:

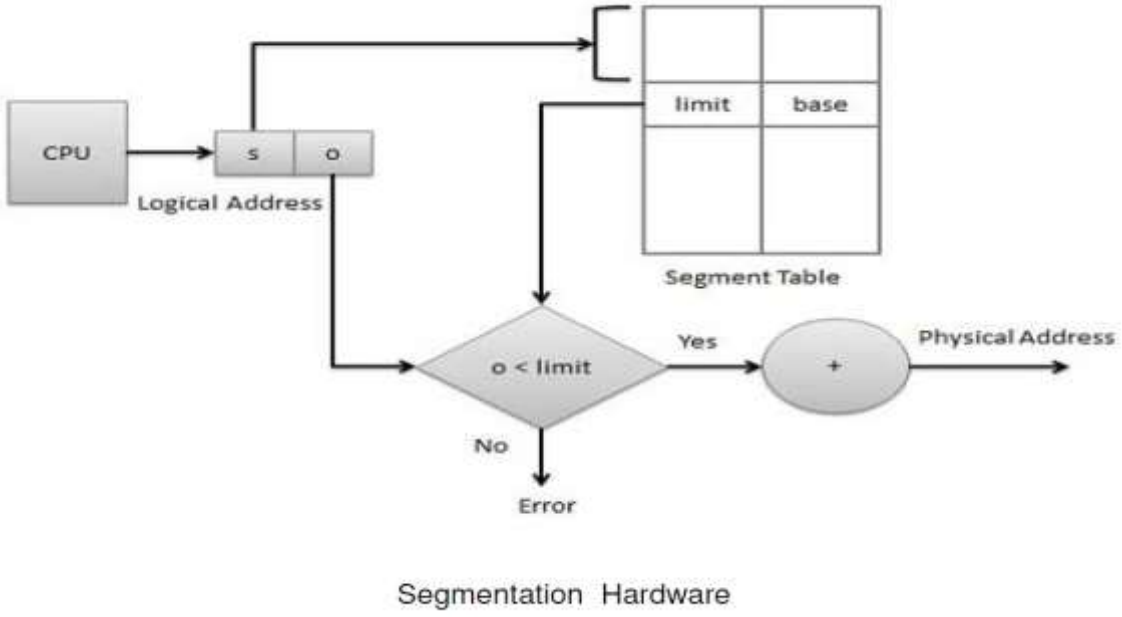


Fig 14 : Segmentation Hardware

- Each entry of the segment table has a segment base and segment limit.
- The segment base contains the starting physical address where the segment resides in the main memory, whereas the segment limit specifies the length of the segment.
- The main difference between the segmentation and multi-partition schemes is that one program may consist of several segments.
- The segment table can be kept either in fast registers or in memory.
- In case a program consists of several segments, we have to keep them in the memory and a **segment-table base register (STBR)** points to the segment table. Moreover, because the number of segments used by a program may vary widely, a **segment-table length register (STLR)** is used.

- One advantage of segmentation is that it automatically provides protection of memory because of the segment-table entries (base and limit tuples).
- Segments also can be shared in a segmented memory system. Segmentation may cause external fragmentation.

PAGED-SEGMENTATION

Paged Segmentation

In paged segmentation, we divide every segment in a process into fixed size pages. We need to maintain a page table per segment CPU's memory management unit must support both segmentation and paging. The following snapshots illustrate these points.

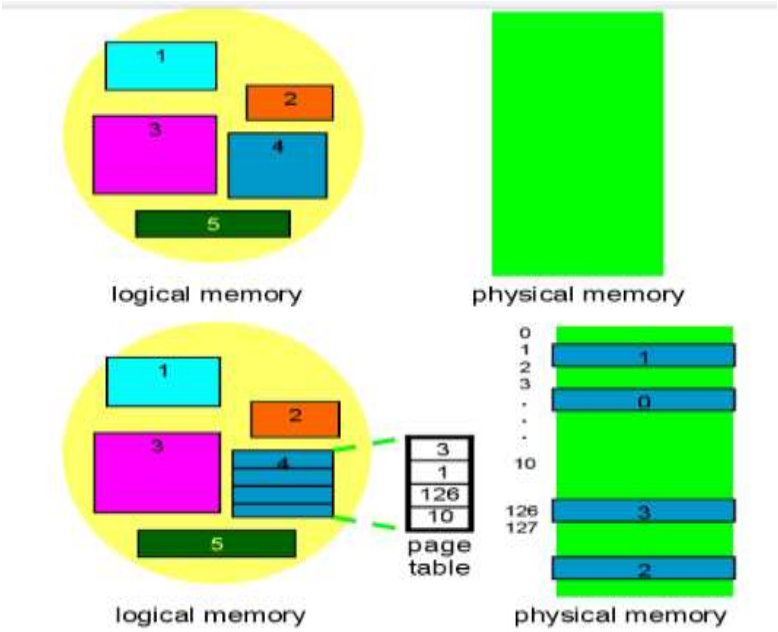
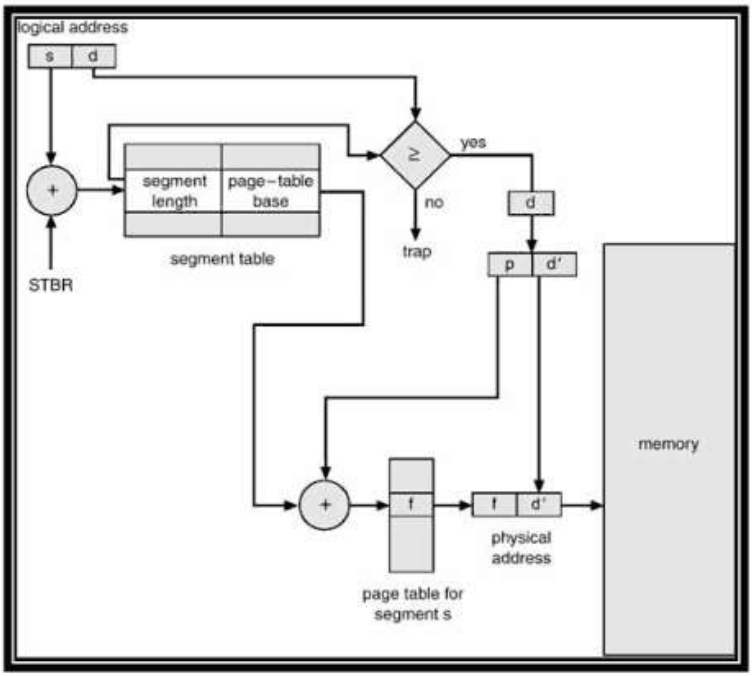


Fig 15 : Paged Segmentation



MULTICS:

We now take the example of one of the finest operating systems of late 1960s and early 1970s, known as the MULTICS operating system. Here are the specifications of the CPU supported by MULTICS and calculation of its various parameters such as the largest segment size supported by MULTICS.

- GE 345 processor
- Logical address = 34 bits

ADVANTAGES OF SEGMENTATION

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

DISADVANTAGE OF SEGMENTATION

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

PAGE REPLACEMENT

- In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in.
- Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page.
- Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

PAGE FAULT

- A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory

PAGE REPLACEMENT ALGORITHMS

- There are many page-replacement algorithms. We select a particular replacement algorithm based on the one with the lowest page-fault rate.
- An algorithm is evaluated by running it on a particular string of memory references (called a reference string) and computing the number of page-faults.
- Reference strings are generated artificially (by a random-number generator, for example) or by tracing a given system and recording the address of each memory reference.

TYPES OF REPLACEMENT ALGORITHMS

- **FIFO –** First In First Out
- **LRU –** Least Recently Used
- **Optimal Algorithm**
- **LFU -** Least Frequently Used
- **MFU -** Least Frequently Used

1

PAGE REPLACEMENT ALGORITHMS:

* The Longest time period repeated pages are to be replaced first.

* page fault: The CPU will demand for the particular page number; If it is not present in the reference string (RAM / main memory). Then it is going to be Page Fault.

* Page Hit: The CPU will demand for the particular page number; If it is present in the RAM/ main memory. Then it is going to be Page Hit.

TYPES: → FIFO: First In First Out.
→ LRU: Least Recently Used.
→ Optimal Algorithm.

} with the help of this 3, we can find page fault, Page Hit, Hit ratio.

FIFO:

* First In First Out.

* From the name itself we can recognize that which page has to be replaced first and which page has to be Deleted first.

* Replace the pages that has been in the RAM/ main memory for the longest time period.

* The Example as follows,

(1) 2

Reference String :

7 0 1 2 0 3 0 4 2 3 0 3 1 2 0

* These page numbers are present in the RAM/memory
* Because the main memory size is lesser than the logical memory.
* The main memory can't store all the pages at a time.
* If the particular demanded page number not present in main memory. It has to be borrowed from secondary memory, by deleting the longest time repeated page number.
* If the page faults are more than RAM gets slower.

(left side).

7 0 1 2 0 3 0 4 2 3 0 3 1 2 0

Page Frames															
F1	(7)	7	7	(2)	2	2	2	(4)	4	4	0	0	0	0	Present 0
F2		(0)	0	0	Present 0	(3)	3	3	(2)	2	2	2	1	1	1
F3			(1)	1	1	1	(0)	0	0	(3)	3	Present 3	3	2	2
	*	*	*	*	Hit	*	*	*	*	*	*	Hit	*	*	Hit

Page Hit : Hits → 3

Pag faults : * → 12

Reference string → 15.

Explanation for FIFO: 3

* Starting RAM / Main memory is free. It does not contains any page numbers.

* It contains (RAM) only three frames for Replacing the longest time period page numbers.

* The CPU is demanding for Page no: 7..

The RAM doesn't contains any page no in the staiting. So CPU will borrow the particular page number from RAM. So page no. 7 is not there in RAM. Then it is going to be 'Page fault *'

* The process continuous like this.....

* Now the CPU is demanding for page '0'.

7 0 1 2 **0** 3 0 4 2 3 0 3 1 2 0

⇒ It is present in the given Reference String Then it is going to be 'page Hit - Hit'.

$$\text{Hit ratio} = \frac{\text{No. of Hits}}{\text{total no. of. Reference String}}$$

$$= \frac{3}{15} \times 100 = 20\% \text{ (approx)}$$

$$\text{page fault} = \frac{\text{No. of fault}}{\text{Ref. string}}$$

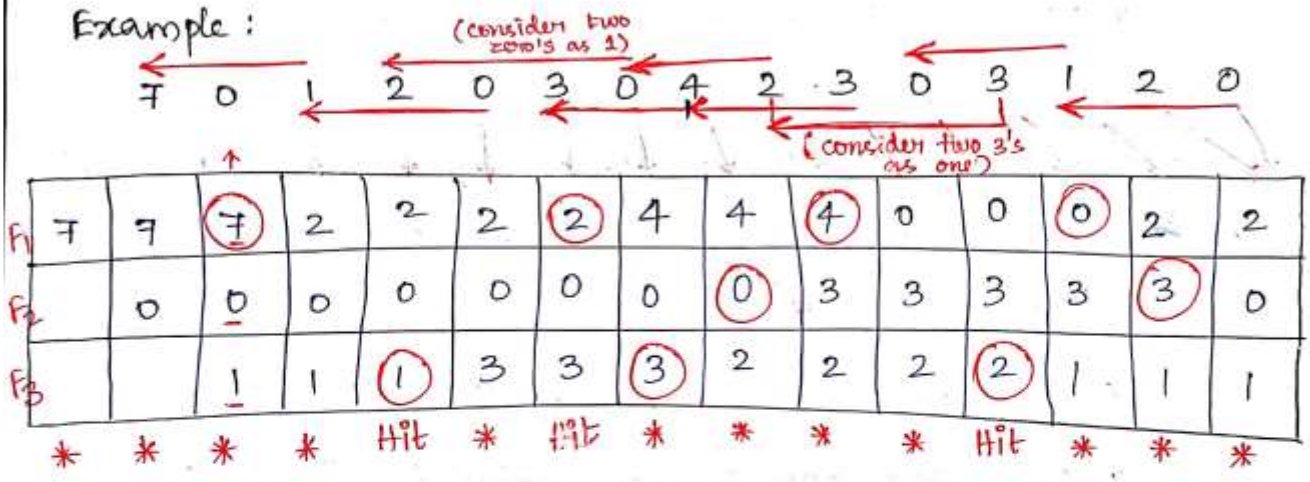
$$= \frac{12}{15} \times 100 = 80\% \text{ (approx)}.$$

4

LRU:

* Least Recently Used.
* Note : 3 Frames = 3 page numbers. (compare)
* Left side page numbers are replaced with another.

Example :



* Step 1 : 7 0 1 (frame) = 7 0 1 (page no). check the Arrow [For your Reference].
* Step 2 : compare the page numbers and frames in left direction.
* Don't compare the Frame F_2 for the particular time period.
* Check the frames with given page numbers.

For example : 7 0 1 (1 is most recently used. 7 is Least recently used. so select '7' and replace with next page number '2'.)

The process continuous.....

5

* Now, check the frame : 4 0 2 and page number (0 & 2). In frames 2 is very most recently used ; 4 is mostly recently used (only 2 times repeated. we should replace the frame with another page number only if particular page number repeats 3 is above times). So In this condition you just delete the frame (F_2) - '0'. with '3'.

The process continuous .....

page Hit = 3

Page fault = 12.

OPTIMAL ALGORITHM:

* This algorithm represents the longest time period page number in future (Right Direction).

Example:

longest time in future. →

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1
*	*	*	*	Hit	*	Hit	*	Hit	Hit	*	Hit	Hit	*	Hit	Hit	Hit	*

Explanation:

* The process starts as usual ;

6

* Now, check for longest time in future.

⇒ 7 0 1 (frames) ; comparing these numbers in given Reference string. 7 is most longest traveled page number (i.e., Repeated in future).

These processes are continues .....

* Now, page number 4 is not there in future reference string. In this condition just replace the longest time repeated frame with another page number (i.e) '0' is replaced with 4.

* Then process repeats. After that 4 is not there in future reference string. So if it is in frames also no use. So just delete that page number and replace '0'.

2 (4) 3 replace with 2 0 3

* The process continuous. Last 7 is not there in future. So just replace with longest traveled page number (2) with 7.

Page Hit – 9

Page fault – 9.

LFU (Least Frequently Used) ALGORITHM

- Replace the least frequently used page
- **Disadvantages:**
    - This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

MFU (Least Frequently Used) ALGORITHM

- Replace the most frequently used page
- **Disadvantage:**
    - This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used. Neither MFU nor LFU replacement is common. Their implementation is fairly expensive.

VIRTUAL MEMORY

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.
- In many cases, in the course of execution of a program, some part of the program may never be executed.
- The advantages of virtual memory are:
    - Users would be able to write programs whose logical address space is greater than the physical address space.
    - More programs could be run at the same time thus increasing the degree of multiprogramming.
    - Less I/O would be needed to load or swap each user program into memory, so each program would run faster.

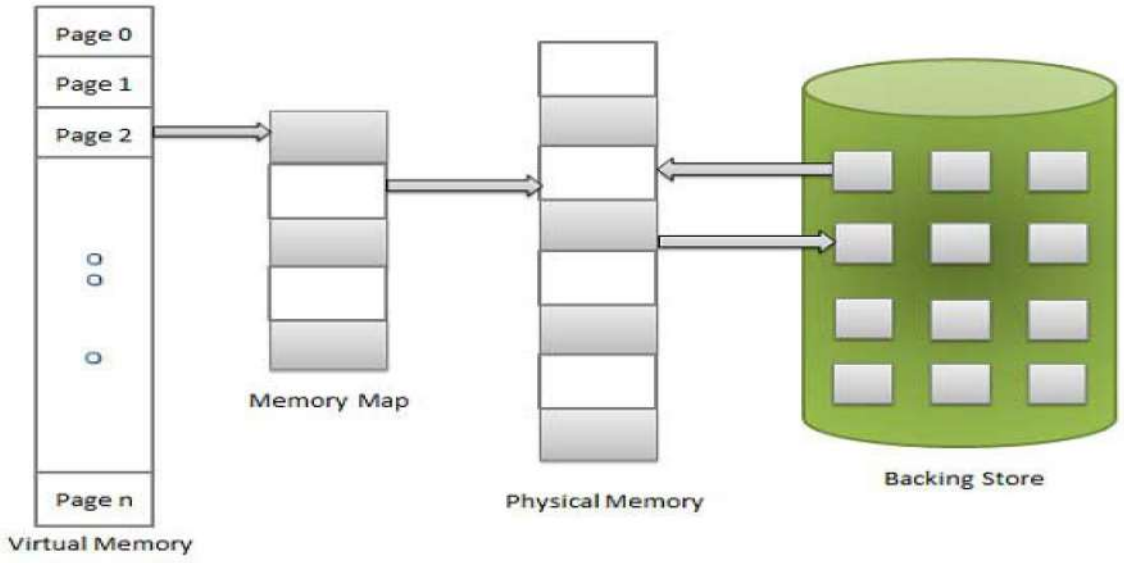


Fig 16 : Virtual Memory

- Virtual memory is the separation of logical memory from physical memory.
- Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system.

DEMAND PAGING

- A demand-paging system is similar to a paging system with swapping.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into the memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

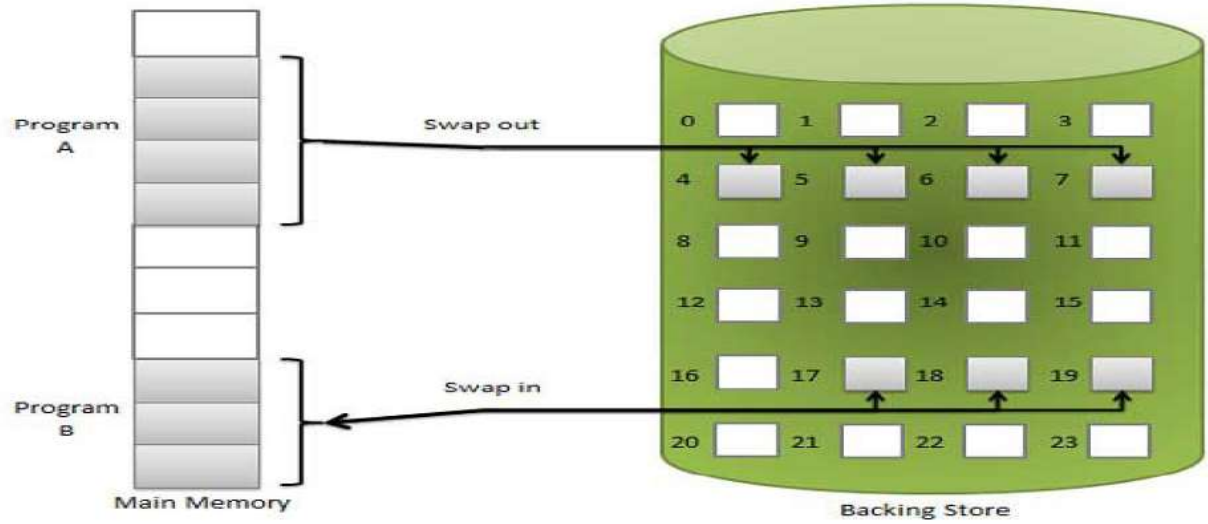


Fig 17 : Demand paging

- To distinguish between those pages that are in memory and those that are on disk, we use an invalid-valid bit which is a field in each page-table entry.
- When this bit is set to "valid", this value indicates that the associated page is both legal and in memory.
- If the bit is set to "invalid", it indicates that the page is either not valid (i.e., not in logical address space of the process), or is valid but is currently on the disk.
- The page-table entry for a page that is not currently in memory is simple marked invalid, or contains the address of the page on disk.
- Access to a page marked invalid causes a page-fault trap. The procedure for handling page fault is given below:

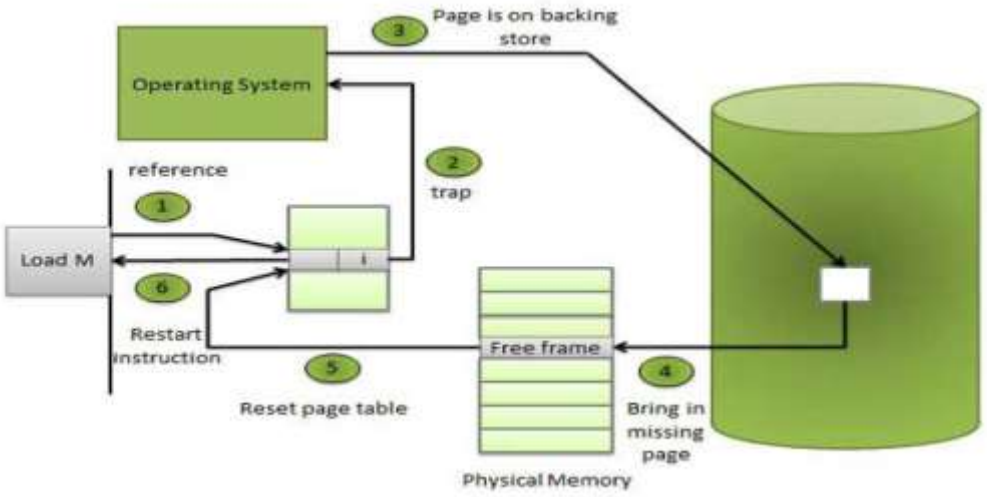


Fig 17 : Demand paging

ADVANTAGES

- Large virtual memory.
- More efficient use of memory.
- Unconstrained multiprogramming.
- There is no limit on degree of multiprogramming.

DISADVANTAGES

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of explicit constraints on jobs address space size.

PURE DEMAND PAGING

- In the extreme case, we could start executing a process with no pages in memory.
- When the operating system set the instruction pointer to the first instruction of the process which is on a non-memory-resident page, the process would immediately fault for the page.
- After this page was brought into memory, the process would continue to execute, faulting as necessary until every page that is needed was actually in memory.
- Then, it could execute with no more faults. This scheme is called **pure demand paging.**

DEMAND PAGING vs ANTICIPATORY PAGING

DEMAND PAGING

- When a process first executes, the system loads into main memory the page that contains its first instruction
- After that, the system loads a page from secondary storage to main memory only when the process explicitly references that page
- Requires a process to accumulate pages one at a time

ANTICIPATORY PAGING

- Operating system attempts to predict the pages a process will need and preloads these pages when memory space is available
- Anticipatory paging strategies must be carefully designed so that overhead incurred by the strategy does not reduce system performance.

FILE CONCEPT

- A file is a named collection of related information that is recorded on secondary storage.
- Data can NOT be written to secondary storage unless they are within a file.

FILE STRUCTURE

- A text file is a sequence of characters organized into lines.
- A source file is a sequence of subroutines and function.
- An object file is a sequence of bytes organized into blocks understandable by the system's linker
- An executable file is a series of code sections that the loader can bring into memory and execute.

FILE ATTRIBUTES

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

- Information about files are kept in the directory structure, which is maintained on the disk

FILE OPERATIONS

- The operating system provides system calls to create, write, read, reposition, delete, and truncate files. We look at what the operating system must do for each of these basic file operations.
- **CREATING A FILE**
    - Firstly, space in the file system must be found for the file. Secondly, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the system.
- **WRITING A FILE.**
    - In the system call for writing in a file, we have to specify the name of the file and the information to be written to the file.
    - The operating system searches the directory to find the location of the file.
    - The system keeps a write pointer to the location in the file where the next write is to take place.
    - The write pointer must be updated whenever a write occurs.
- **READING A FILE**
    - To read a file, we make a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
    - As in writing, the system searches the directory to find the location of the file, and the system keeps a read pointer to the location in the file where the next read is to take place.
    - The read pointer must be updated whenever a read occurs.
- **REPOSITIONING WITHIN A FILE**
    - In this operation, the directory is searched for the named file, and the current-file-position is set to a given value. This file operation is called a **FILE SEEK**.
- **DELETING A FILE**
    - We search the directory for the appropriate entry.
    - Having found it, we release all the space used by this file and erase the directory entry**.**
- **TRUNCATING A FILE**
    - This operation is used when the user wants to erase the contents of the file but keep the attributes the file intact

MEMORY-MAPPED FILES

- Some operating systems allow mapping sections of file into memory on virtual-memory systems.
- It allows part of the virtual address space to be logically associated with a section of a file.
- Reads and writes to that memory region are then treated as reads and writes to the file, greatly simplifying the file use.
- Closing the file results in all the memory-mapped data being written back to the disk and removed from the virtual memory of the process.

OPEN FILES

- **File pointer:** pointer to last read/write location, per process that has the file open
- **File-open count:** the counter tracks the number of opens and closes, and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**: the info needed to locate the file on disk.
- **Access rights**: per-process access mode information so OS can allow or deny subsequent I/O request

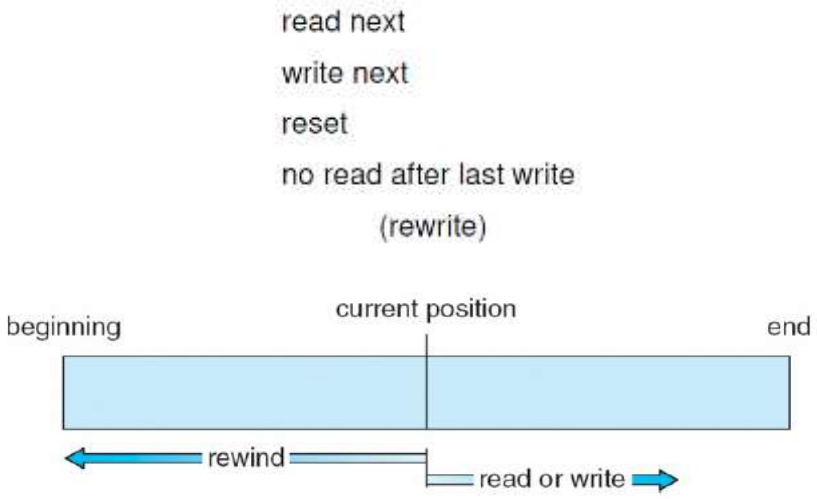
FILE TYPES – NAME, EXTENSION

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

Table 4 : File Concepts and types

ACCESS METHODS

- **Sequential Access**



- **Direct Access**

read n
write n
position to n
read next
write next
rewrite n

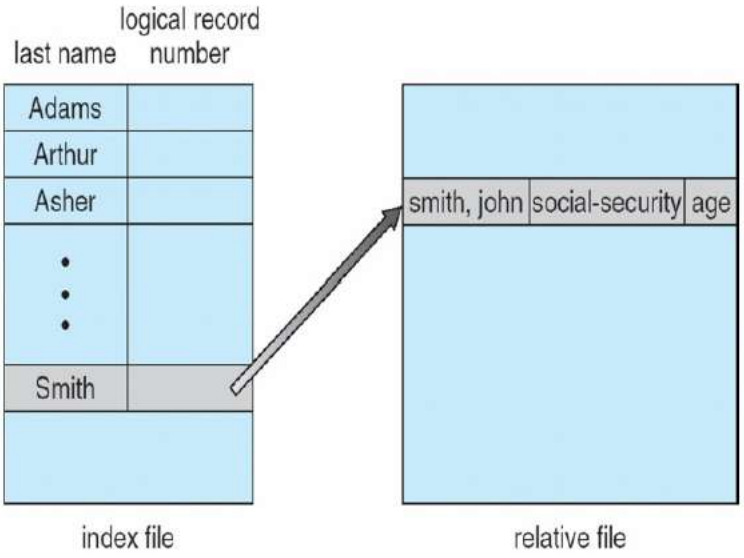
n = relative block number

- **Simulation of Sequential Access on Direct-access File**

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Cp=> current position

- **Index and Relative Files**



  - The index contains pointers to the various blocks.
  - To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record

DISK STRUCTURE

- Disk can be subdivided into partitions
- Disks or partitions can be **Redundant Arrays of Independent Disks (RAID)** protected against failure

- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's info in device directory or volume table of contents
- There are five commonly used directory structures:
    - **Single-Level Directory**
    - **Two-Level Directory**
    - **Tree-Structure Directories**
    - **Acyclic-Graph Directories**
    - **General Graph Directories**

A TYPICAL FILE-SYSTEM ORGANIZATION

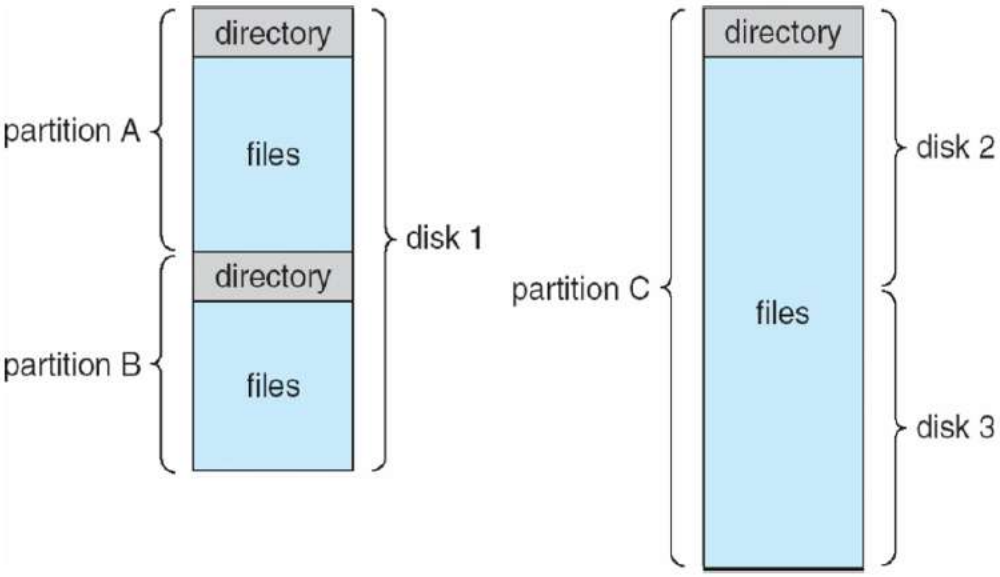


Fig 18 : File system

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file

- Traverse the file system

SINGLE-LEVEL DIRECTORY

- All files are contained in the same directory.
- It is difficult to maintain file name uniqueness.
- CP/M-80 and early version of MS-DOS use this directory structure.

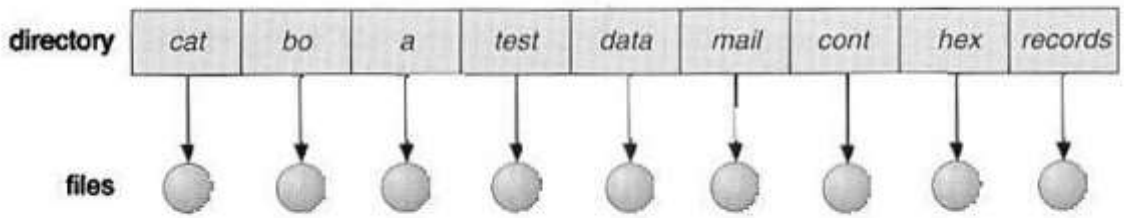


Fig 19 : Single level directory

TREE-STRUCTURED DIRECTORIES

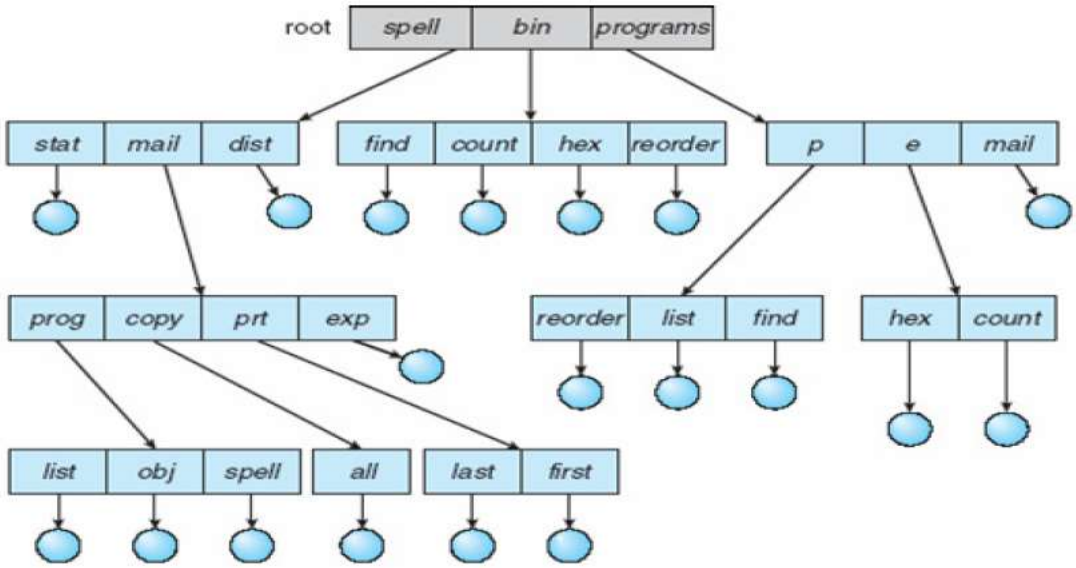


Fig 20 : Multi-level directory , Tree structured directory

- **ABSOLUTE PATH:** begins at the root and follows a path down to the specified file.
    - root/spell/mail/prt/first
- **RELATIVE PATH:** defines a path from the current directory.
    - prt/first given root/spell/mail as current path

ACYCLIC-GRAPH DIRECTORY

- This type of directories allows a file/directory to be shared by multiple directories.
- This is different from two copies of the same file or directory.
- An acyclic-graph directory is more flexible than a simple tree structure.

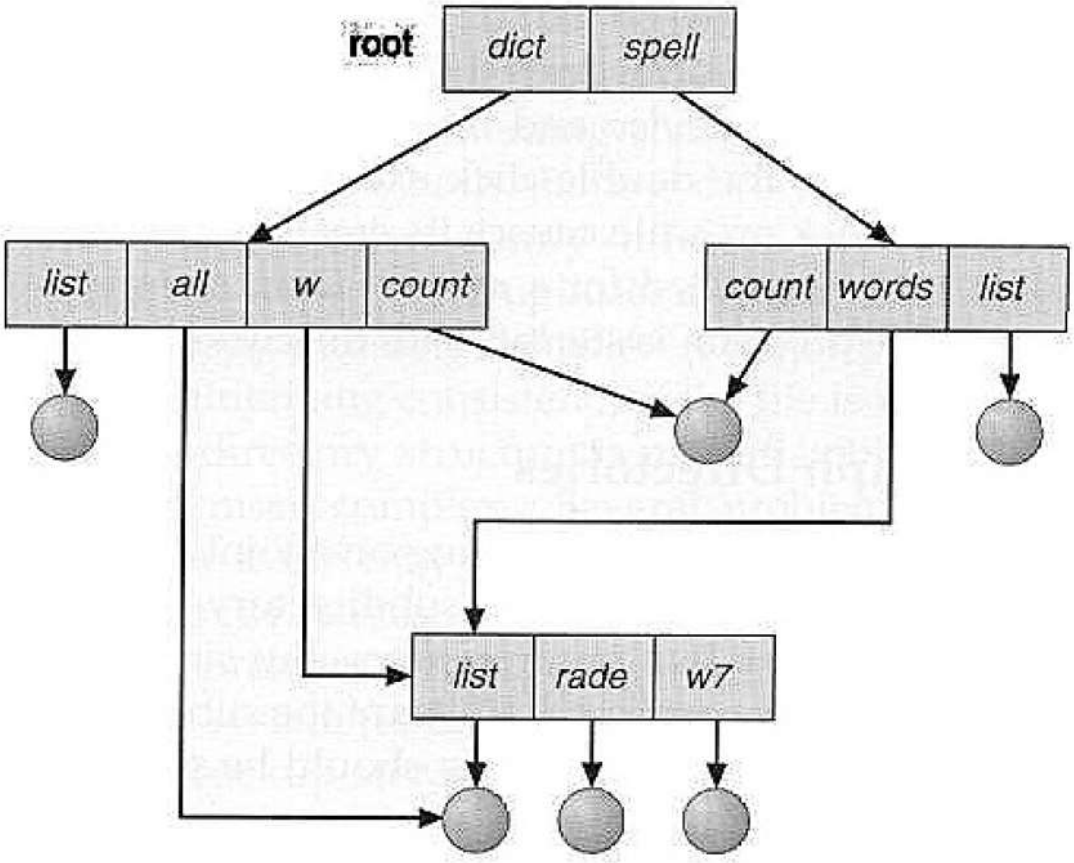


Fig 21 : Acyclic graph directory

GENERAL GRAPH DIRECTORY

- It is easy to traverse the directories of a tree or an acyclic directory system.
- However, if links are added arbitrarily, the directory graph becomes arbitrary and may contain cycles

- Creating a new file is done in current directory
    - Delete a file
        - rm <file-name>
    - Creating a new subdirectory is done in current directory
        - mkdir <dir-name>

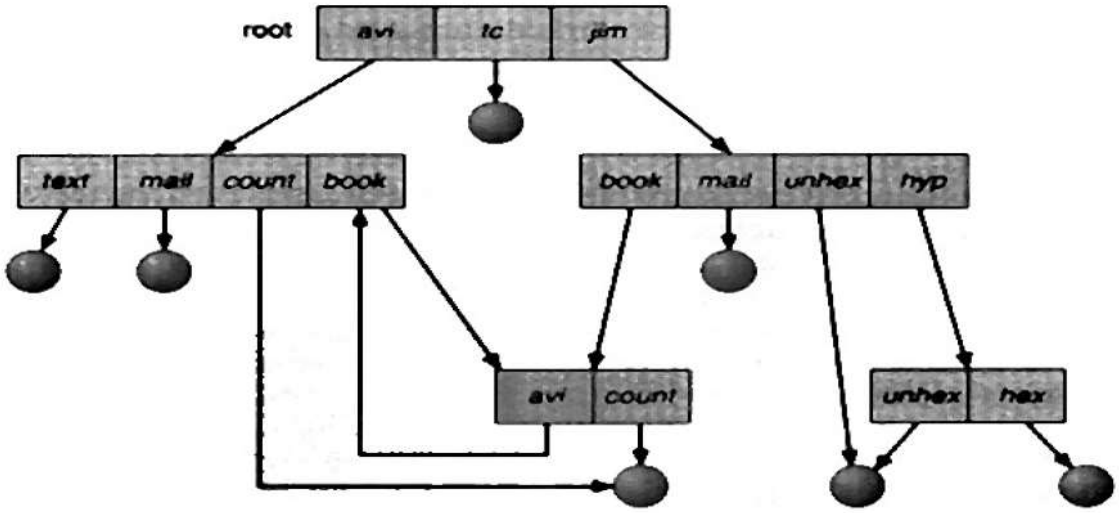
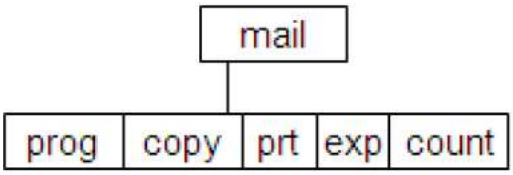


Fig 22 : General directory

Example:

if in current directory /mail

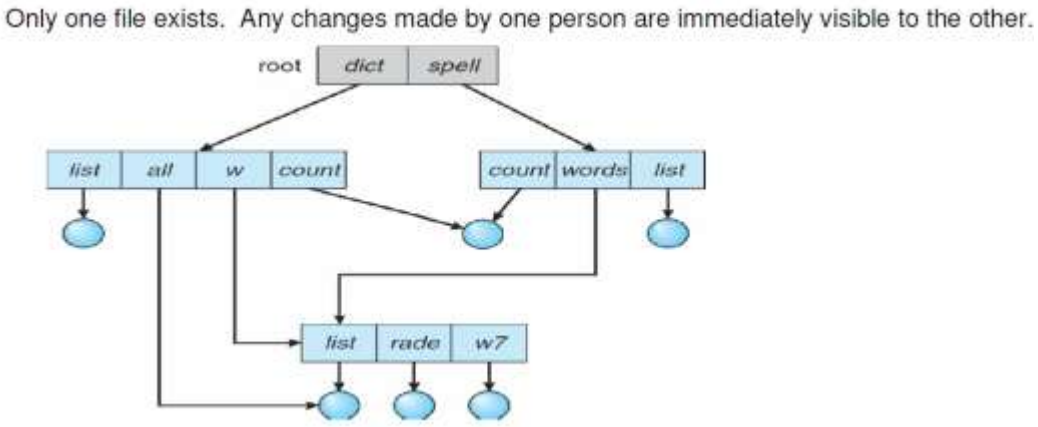
mkdir count



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Option1: do not delete a directory unless it is empty, such as MS-DOS

Option2: delete all files in that directory, such as UNIX rm command with r option



- Efficient searching
- Grouping Capability
    - pwd
    - cd /spell/mail/prog
    - New directory entry type
    - Link – another name (pointer) to an existing file
    - Resolve the link – follow pointer to locate the file

FILE SHARING

- When a file is shared by multiple users, how can we ensure its consistency
- If multiple users are writing to the file, should all of the writers be allowed to write? Or, should the operating system protect the user actions from each other?
- This is the file consistency semantics

FILE CONSISTENCY SEMANTICS

- Consistency semantics is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously.
- Consistency semantics is an important criterion for evaluating any file system that supports file sharing.
- There are three commonly used semantics
    - Unix semantics
    - Session Semantics
    - Immutable-Shared-Files Semantics

Unix Semantics

- Writes to an open file by a user are visible immediately to other users have the file open at the same time. All users share the file pointer.
- A file has a single image that interleaves all accesses, regardless of their origin

Session Semantics

- Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously
- Once a file is closed, the changes made to it are visible only in sessions started later.
- Already-open instances of the file do not affect these changes
- Multiple users are allowed to perform both read and write concurrently on their image of the file without delay.
- The Andrew File System (AFS) uses this semantics.

Immutable-Shared-Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two important properties**:**
    - **Its name may not be used**
    - **Its content may not be altered**

FILE PROTECTION

- We can keep files safe from physical damage (i.e., reliability) and improper access (i.e., protection).
- Reliability is generally provided by backup.
- The need for file protection is a direct result of the ability to access files.
- Access control may be a complete protection by denying access. Or, the access may be controlled.

TYPES OF ACCESS

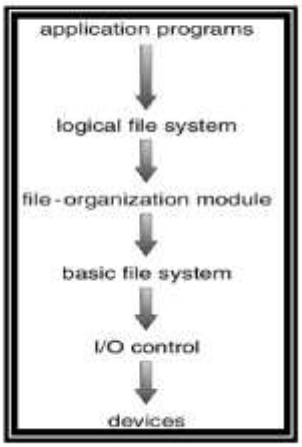
- **Read:** read from the file
- **Write:** write or rewrite the file
- **Execute**: load the file into memory and execute it
- **Append**: write new info at the end of a file

- **Delete:** delete a file
- **List:** list the name and attributes of the file

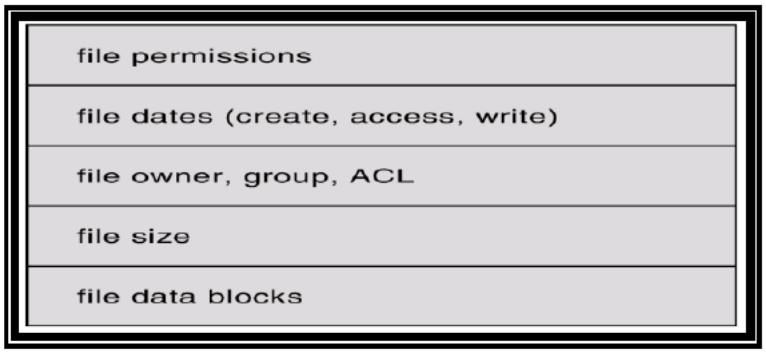
FILE-SYSTEM STRUCTURE

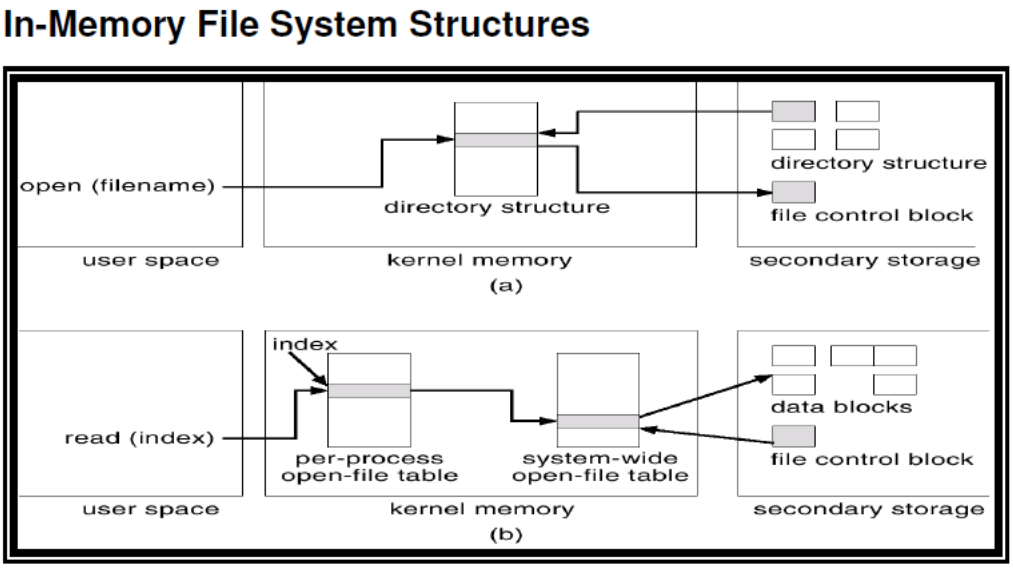
- Logical storage unit
- Collection of related information

Layered File System



A Typical File Control Block





VIRTUAL FILE SYSTEMS

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

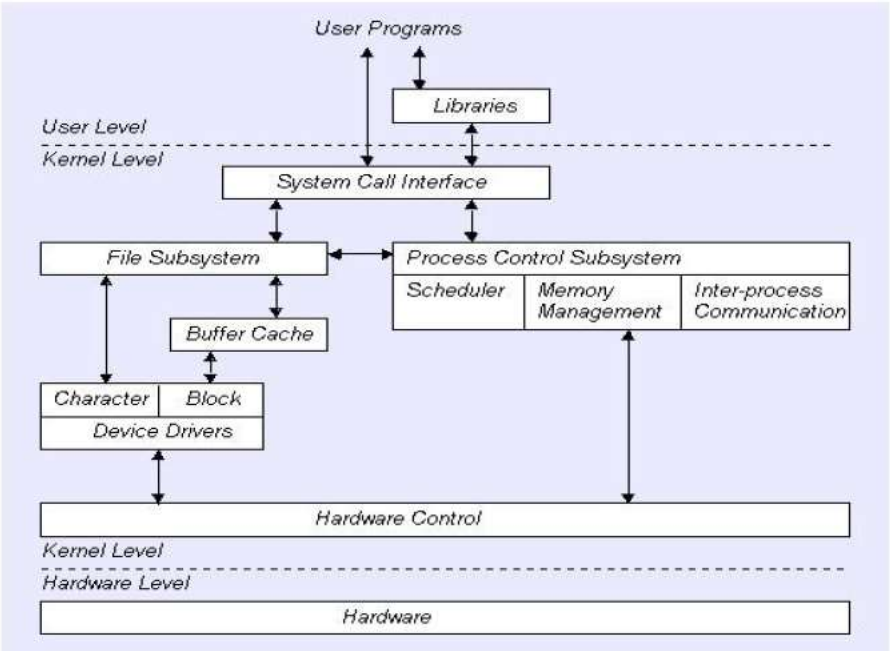
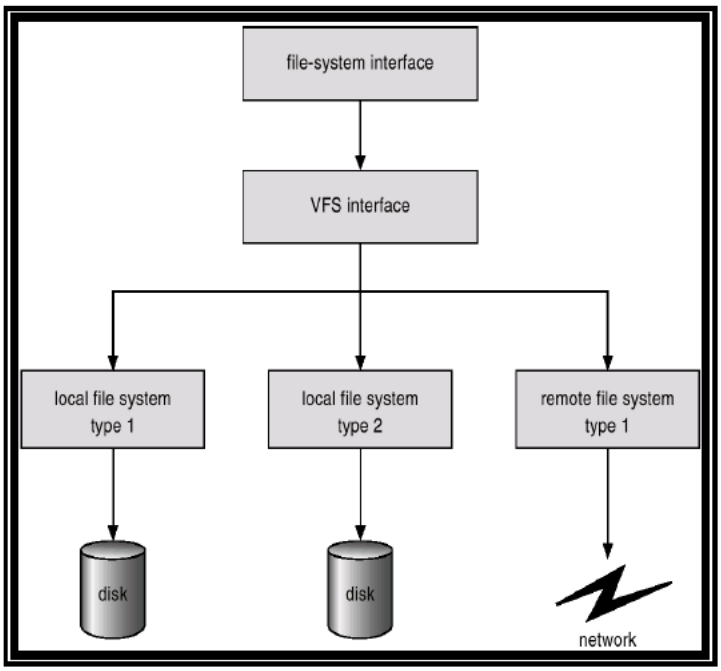


Fig 23 : Virtual file system

Schematic View of Virtual File System



ALLOCATION METHODS

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- Simple – only starting location (block #) and length (number of blocks) are required.
- Random access.
- Wasteful of space (dynamic storage-allocation problem).
- Files cannot grow.

Contiguous Allocation of Disk Space

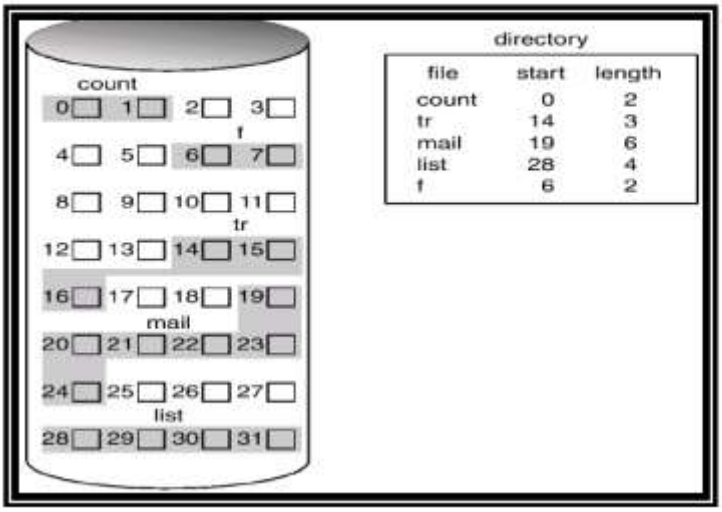


Fig 24 : Disk space

Linked Allocation

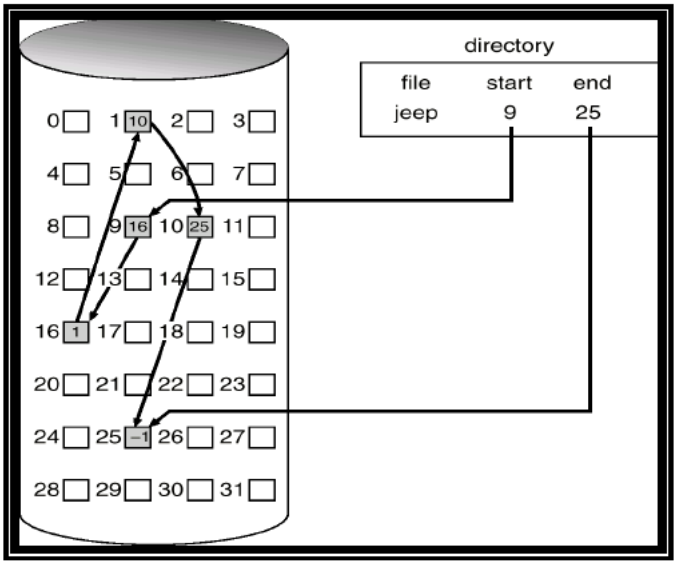
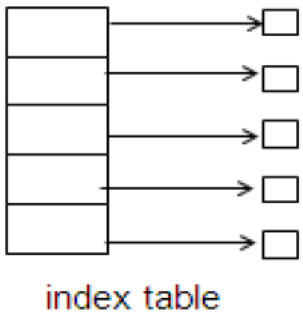


Fig 25 : linked allocation

- Simple – need only starting address
- Free-space management system – no waste of space

- No random access
- Space waste for pointer (e.g. 4byte of 512 B)
- Reliability

Indexed Allocation



- Brings all pointers together into the index block.

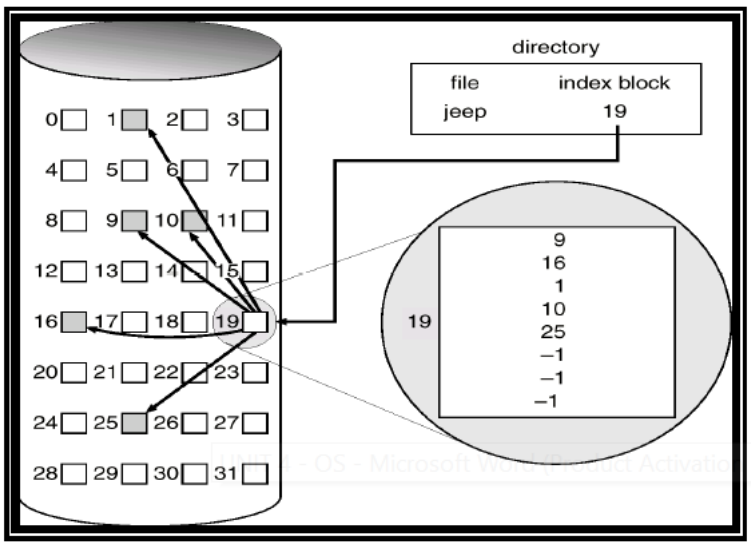
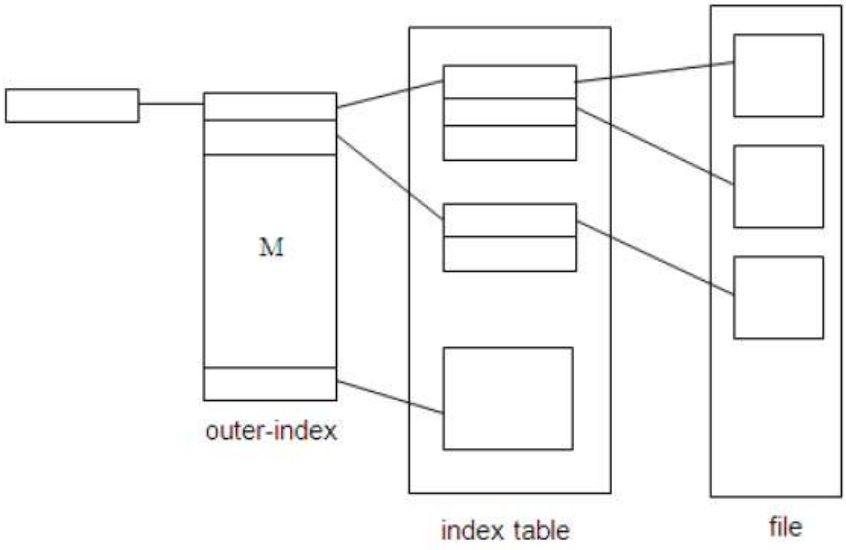


Fig 26 : Indexed allocation

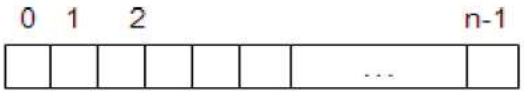
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block

Indexed Allocation – Mapping



Free-Space Management

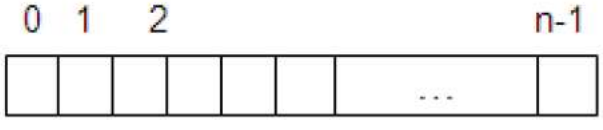
Bit vector (*n* blocks)



bit[*i*] = 0 ⇒ block[*i*] free
1 ⇒ block[*i*] occupied

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit



bit[*i*] = 0 ⇒ block[*i*] free
1 ⇒ block[*i*] occupied

- Bit map requires extra space. Example:

  block size = 2^{12} bytes (4 K)

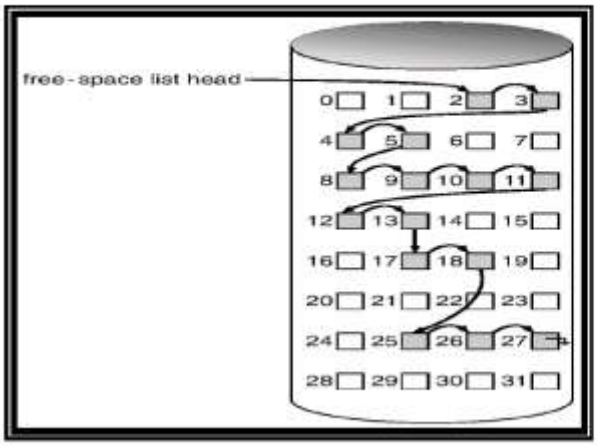
  disk size = 2^{30} bytes (1 gigabyte)

  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

  - Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
  - Large free blocks can be quickly found
- Counting
- Need to protect:
  - Pointer to free list
  - Bit map

- Must be kept on disk
- Copy in memory and disk may differ.

- Linked Free Space List on Disk



EFFICIENCY AND PERFORMANCE

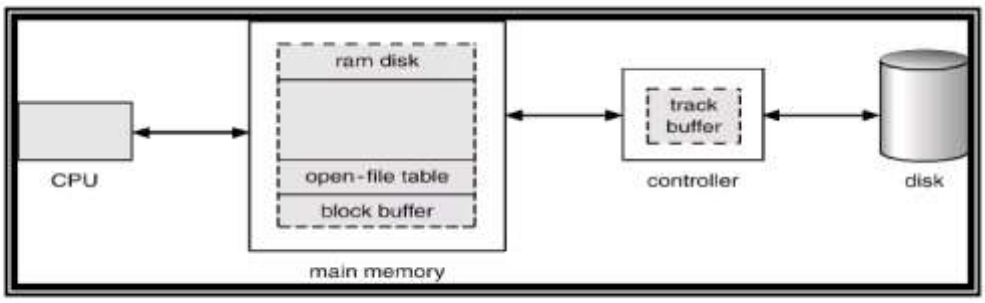
- **Efficiency dependent on:**

  disk allocation and directory algorithms

  types of data kept in file's directory entry

- **Performance**
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

Various Disk-Caching Locations



PAGE CACHE

- A page cache caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.
- This leads to the following figure.

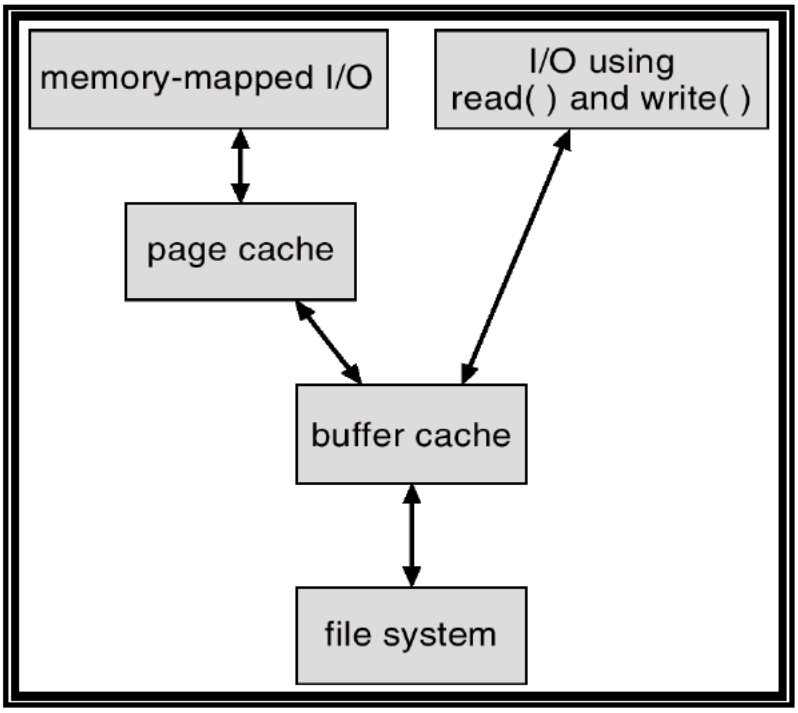


Fig 25 : Paged cache

Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.

I/O Using a Unified Buffer Cache

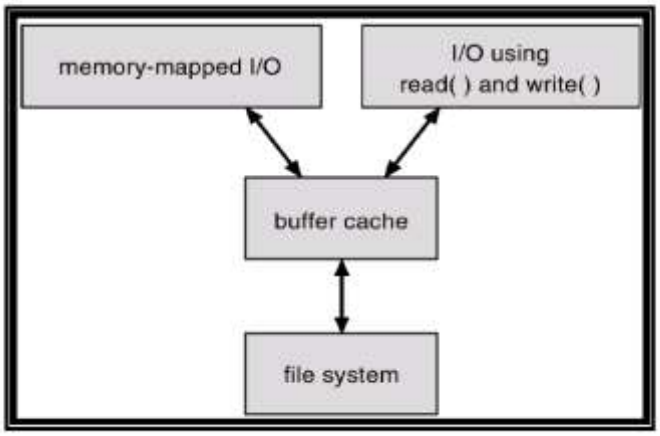


Fig 26 : Buffer cache

RECOVERY

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by restoring data from backup.

LOG STRUCTURED FILE SYSTEMS

- Log structured (or journaling) file systems record each update to the file system as a transaction.
- All transactions are written to a log. A transaction is considered committed once it is written to the log.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed

THE SUN NETWORK FILE SYSTEM (NFS)

- An implementation and a specification of a software system for accessing remote filesacross LANs (or WANs).
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet.
- NFS is designed to operate in a heterogeneous environment of different machines operating systems, and network architectures; the NFS specifications independent of these media.
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services.

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors).
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
- The VFS activates file-system-specific operations to handle local requests according to their file-system types.
- Calls the NFS protocol procedures for remote requests.
- NFS service layer – bottom layer of the architecture; implements the NFS protocol

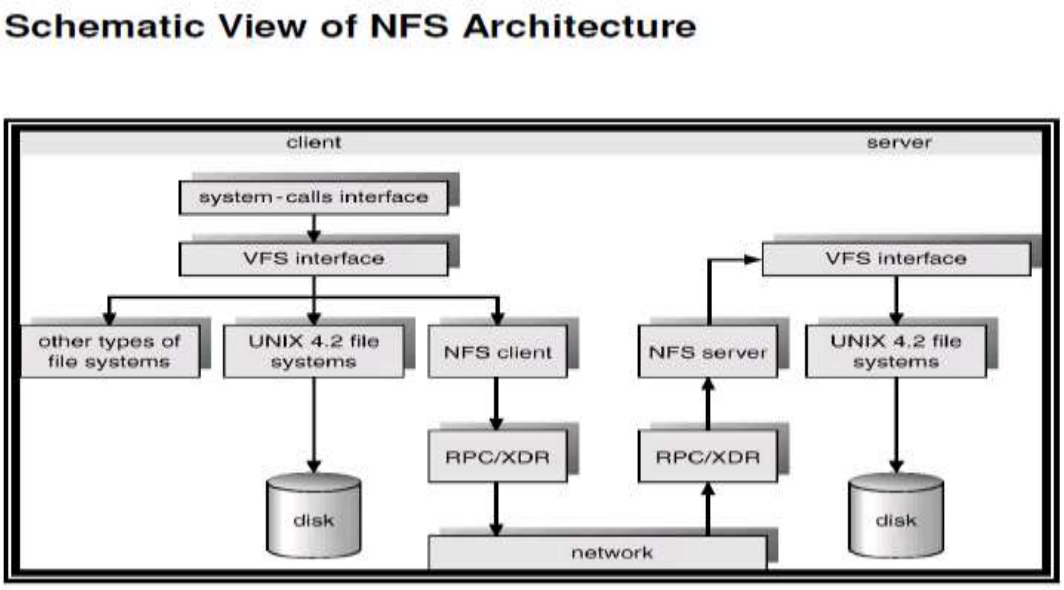


Fig 27 : NFS Architecture

NFS PROTOCOL

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are stateless; each request has to provide a full set of arguments.
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching). The NFS protocol does not provide concurrency control mechanisms

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING
Common to : CSE , IT

UNIT V

UNIT 5 I/O SYSTEM,LINUX & SHELL PROGRAMMING 10 Hrs.

Mass Storage Structure - Disk Structure- Disk Scheduling - Disk Management - Swap Space Management - RAID Structure - Shell Operation Commends - Linux File Structure - File Management Operation - Internet Service - Telnet - FTP - Filters & Regular Expressions - Shell Programming - Variable, Arithmetic Operations, Control Structures, Handling Date, Time & System Information.

UNIT – V – SCS1301- OPERATING SYSTEM

1. Mass Storage Structure

1.1 Overview of Mass-Storage Structure

1.1.1 Magnetic Disks

Traditional magnetic disks have the following basic structure:

- One or more platters in the form of disks covered with magnetic media. Hard disk platters are made of rigid metal, while "floppy" disks are made of more flexible plastic.

- Each platter has two working surfaces. Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.

- Each working surface is divided into a number of concentric rings called tracks. The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a cylinder.

- Each track is further divided into sectors, traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)

- The data on a hard drive is read by read-write heads. The standard configuration (shown below) uses one head per surface, each on a separate arm, and controlled by a common arm assembly which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)

- The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

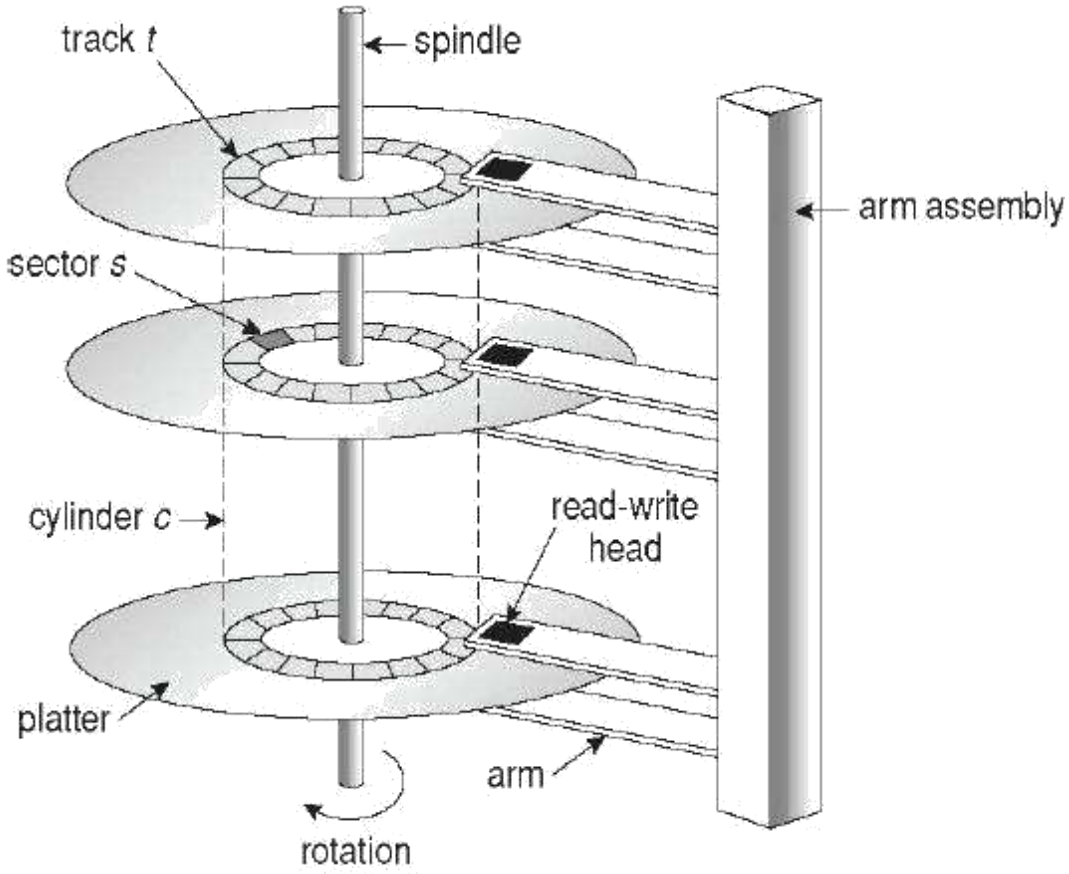


Figure. 1.1 Moving-head disk mechanism

In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:

- The positioning time, a.k.a. the seek time or random access time is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

- The rotational latency is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time.

- (For a disk rotating at 7200 rpm, the average rotational latency would be ½ revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.

- The transfer rate, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the

overall transfer rate, including seeks time and rotational latency as well as the electronic data transfer rate.)

- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a head crash occurs, which May or may not permanently damage the disk or even destroy it completely. For this reason it is normal to park the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.

- Floppy disks are normally removable. Hard drives can also be removable, and some are even hot-swappable, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

- Disk drives are connected to the computer via a cable known as the I/O Bus. Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.

- The host controller is at the computer end of the I/O bus, and the disk controller is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard cache by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

1.1.2 Solid-State Disks – New

As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing use of solid state disks, or SSDs. SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles. Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply. However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.

SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store file system meta-data, e.g. directory and anode information that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter. Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

1.1.3 Magnetic Tapes

Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups. Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives. Capacities of tape drives can range from 20 to 200 GB and compression can double that capacity.

1.2 Disk Structure

The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:

1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.

2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.

3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made. Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:

- With Constant Linear Velocity, CLV, the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.

- With Constant Angular Velocity, CAV, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

1.3 Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

1.3.1 Host-Attached Storage

Local disks are accessed through I/O Ports as described earlier. The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller. SATA is similar with simpler cabling. High end workstations or other systems in need of larger number of disks typically use SCSI disks:

The SCSI standard supports up to 16 targets on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives. A SCSI target is usually a single drive, but the standard also supports up to 8 units within each target. These would generally be used for accessing individual disks within a RAID array. The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses. Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2. SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.

A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the storage-area networks, SANs, to be discussed in a future section. The arbitrated loop, FC-AL that can address up to 126 devices (drives and controllers.)

1.4 Network-Attached Storage

Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS file system mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage. NAS can be implemented using SCSI cabling, or ISCSI uses Internet protocols and standard network connections, allowing long-distance remote access to shared files. NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

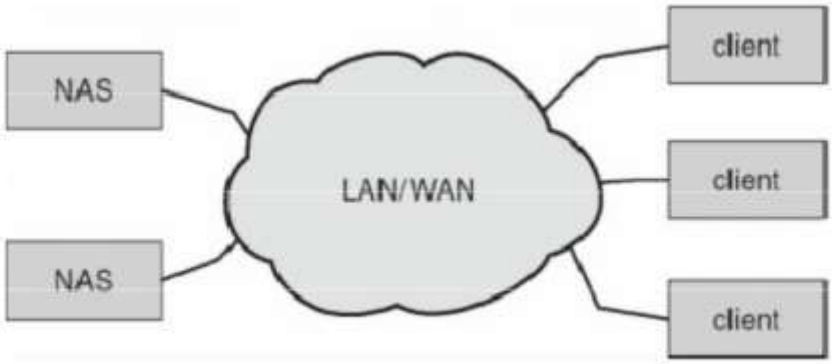


Figure.1.2 Network-attached storage.

1.5 Storage-Area Network

A Storage-Area Network, SAN, connects computers and storage devices in a network, using storage protocols instead of network protocols. One advantage of this is that storage access does not tie up regular networking bandwidth. SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly. SAN is also controllable, allowing restricted access to certain hosts and devices.

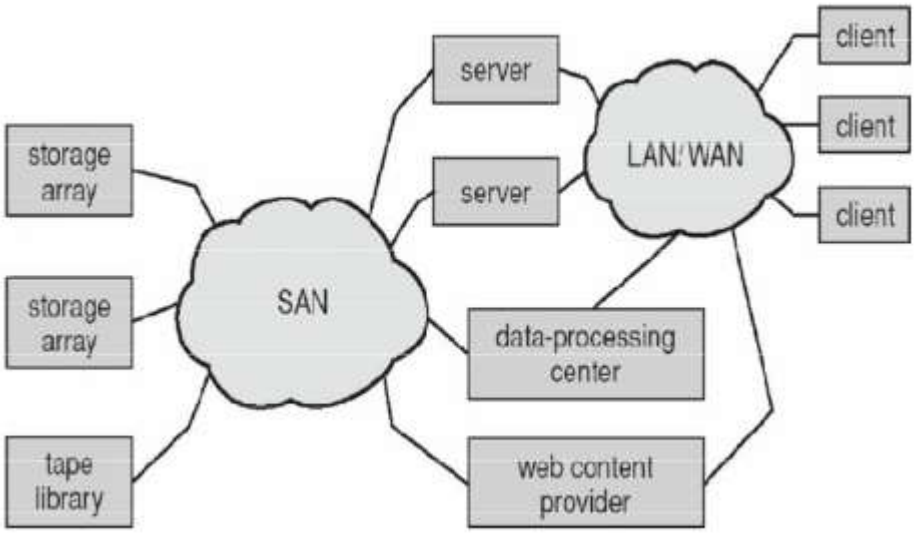


Figure. 1.3 Storage-area network.

1.6 Disk Scheduling

As mentioned earlier, disk transfer speeds are limited primarily by seek times and rotational latency. When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.

Bandwidth is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.) Both bandwidth and access time can be improved by processing requests in a good order. Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

1.2 FCFS Scheduling

First-Come First-Serve is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

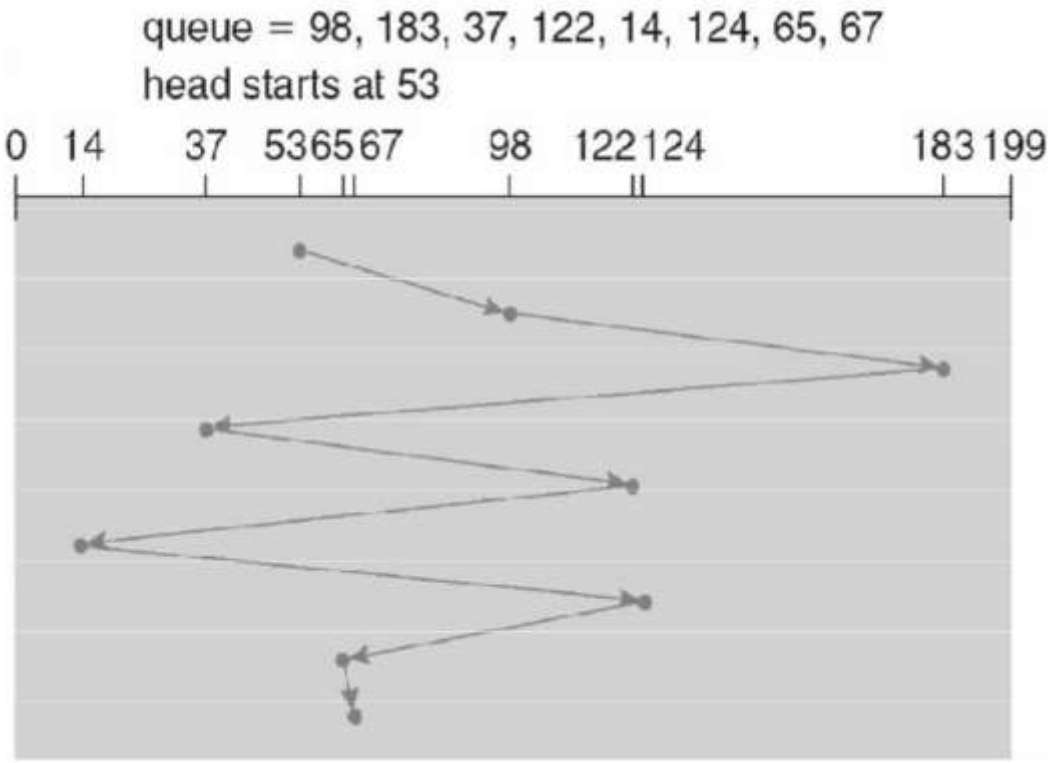


Figure. 1.4. FCFS disk scheduling.

1.2.1 SSTF Scheduling

Shortest Seek Time First scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk. SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

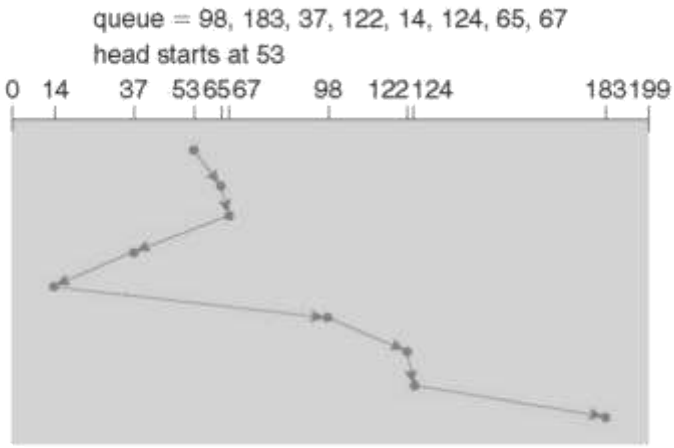


Figure 1.5. SSTF disk scheduling.

1.2.2 SCAN Scheduling

The SCAN algorithm, a.k.a. the elevator algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

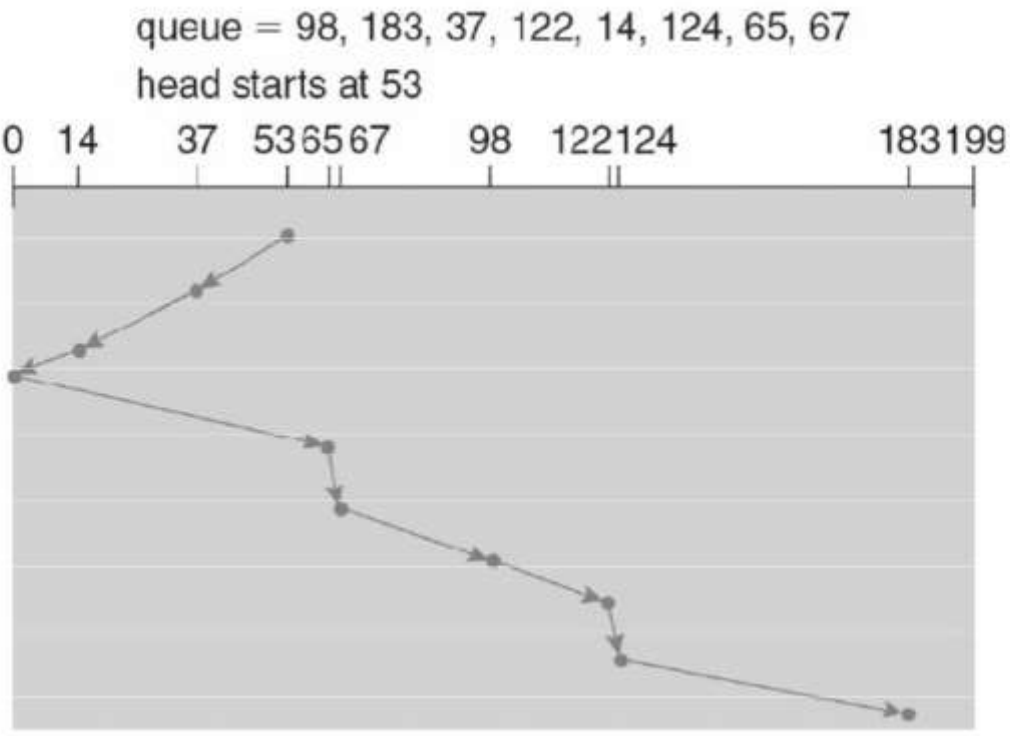


Figure 1.6. SCAN disk scheduling.

Under the SCAN algorithm, if a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon. Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

1.2.3 C-SCAN Scheduling

The Circular-SCAN algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

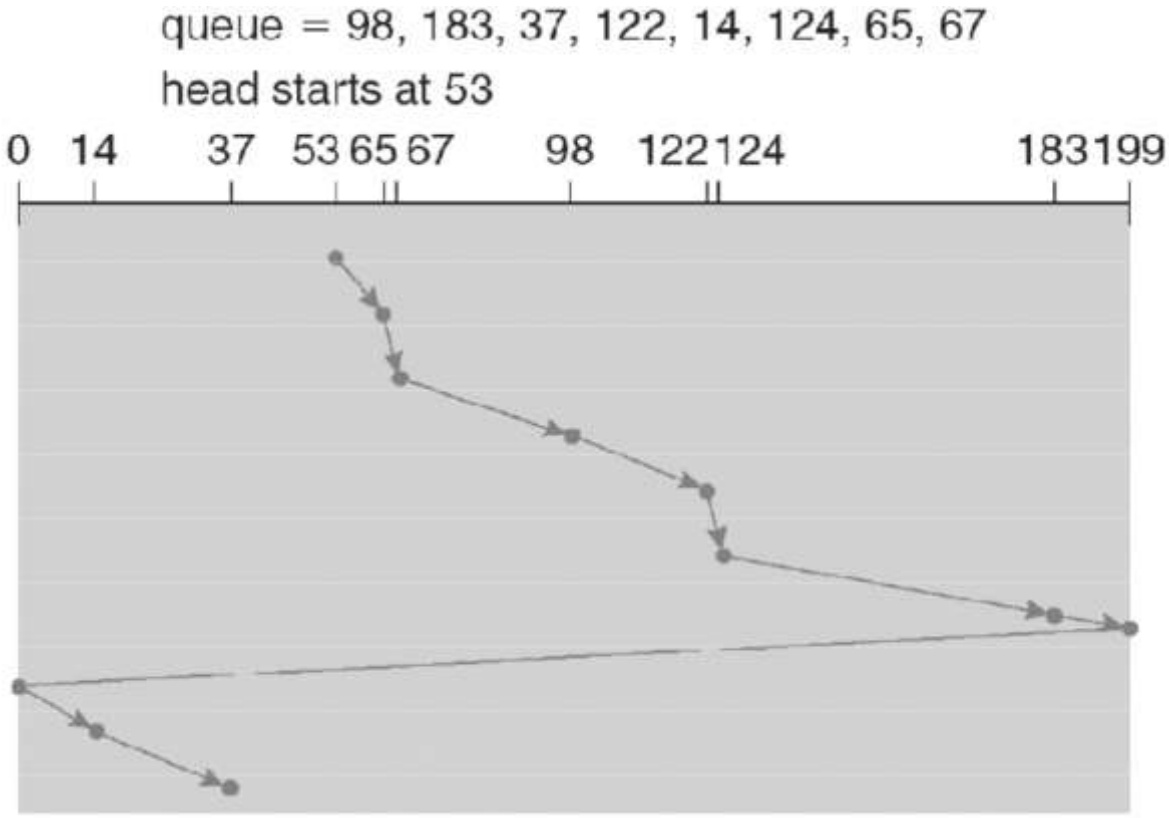


Figure 1.7.C-SCAN disk scheduling.

1.2.4 LOOK Scheduling

LOOK scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

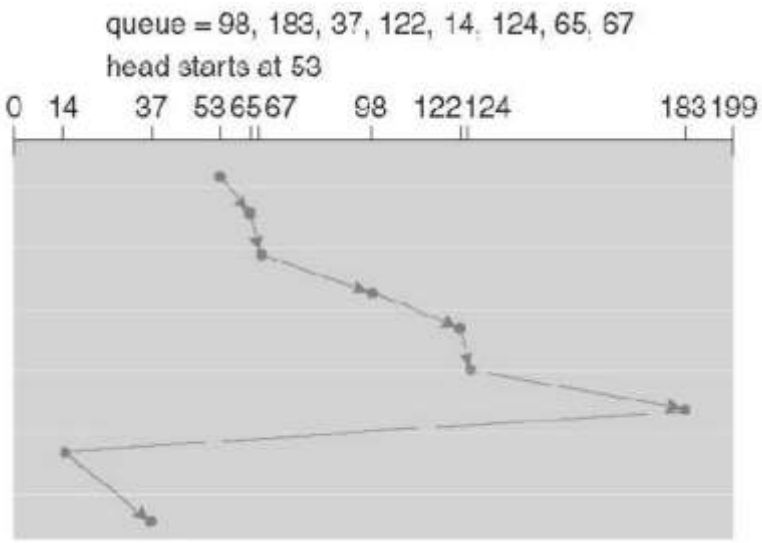


Figure. 1.8. C-LOOK disk scheduling.

1.2.5 Selection of a Disk-Scheduling Algorithm

With very low loads all algorithms are equal, since there will normally only be one request to process at a time. For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough. For busier systems, SCAN and LOOK algorithms eliminate starvation problems.

The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead. Some improvement to overall file system access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.

On modern disks the rotational latency can be almost as significant as they seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.) Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so

that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.

Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.

1.3 Disk Management

1.3.1 Disk Formatting

Before a disk can be used, it has to be low-level formatted, which means laying down all of the headers and trailers marking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and error-correcting codes, ECC, which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.

ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a soft error has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.) Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.

After partitioning, then the file systems must be logically formatted, which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the file system. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the file system structure, but requires that the application progrm manage its own disk storage requirements.)

1.3.2 Boot Block

Computer ROM contains a bootstrap program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.) The first sector on the hard drive is known as the Master Boot Record, MBR, and contains a very small amount of code in addition to the partition table. The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the active or boot partition.

The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.

- In a dual-boot (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.

- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include single-user a.k.a. maintenance or safe modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

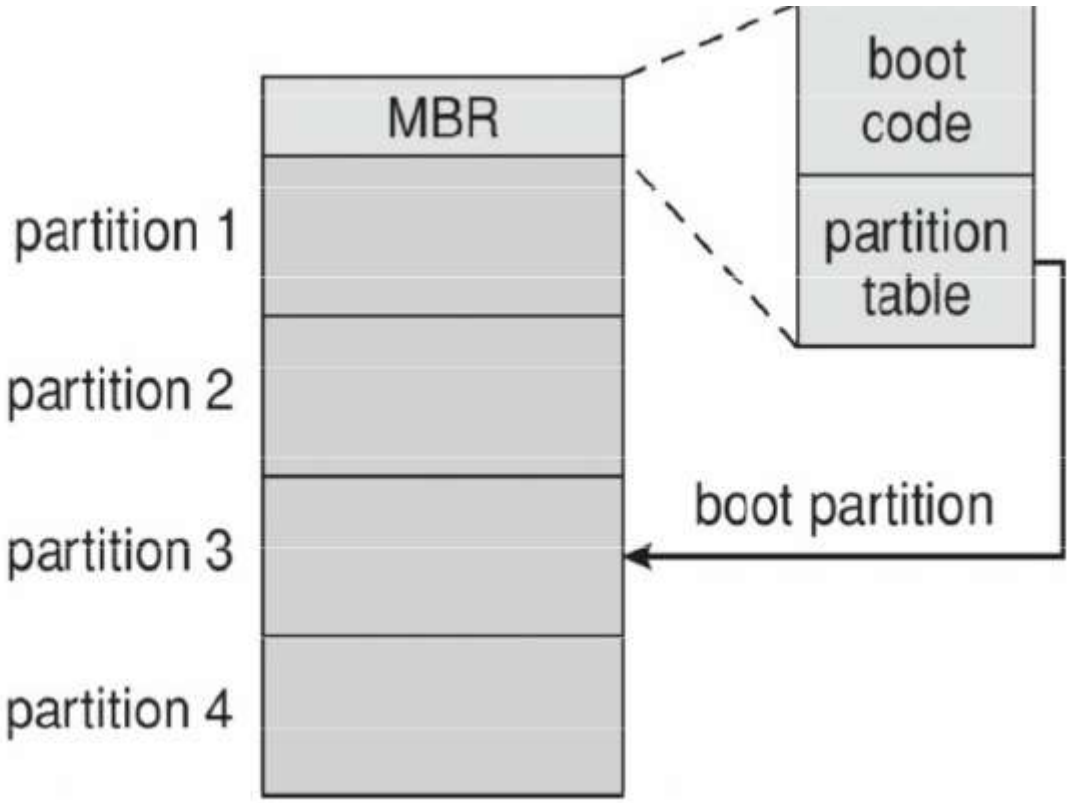


Figure. 1.9. Booting from disk in Windows 2000.

1.3.3 Bad Blocks

No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.

In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or nondestructive.)

Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)

Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder.

Whenever possible abad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. Sector slipping may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained. If the data on a bad block cannot be recovered, then a hard error has occurred. Which requires replacing the file(s) from backups, or rebuilding them from scratch.

2. Swap-Space Management

2.1 Introduction

Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system. Managing swap space is obviously an important task for modern OSes.

2.2 Swap-Space Use

The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all. Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

2.3 Swap-Space Location

Swap space can be physically located in one of two locations:

As a large file which is part of the regular file system. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix. As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

2.4 Swap-Space Management: An Example

Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the file system and read them back in from there than to write them out to swap space and then read them back. In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

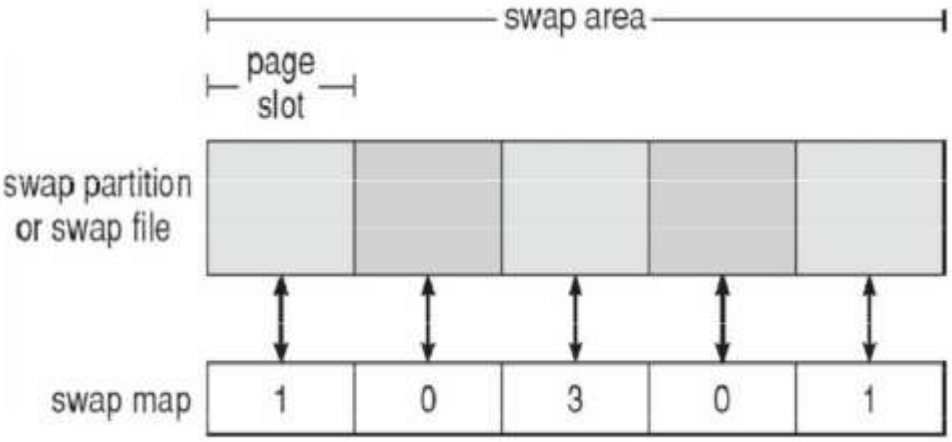


Figure. 2.1. The data structures for swapping on Linux systems.

3. RAID Structure

The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.) RAID originally stood for Redundant Array of Inexpensive Disks, and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to Independent disks.

3.1 Improvement of Reliability via Redundancy

The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually decreases the Mean Time to Failure, MTTF of the system. If, however, the same data was copied onto multiple disks, then the data would not be lost unless both (or all) copies of the data were damaged simultaneously, which a MUCH lower probability than for a single disk is going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the Mean Time to Repair into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the Mean Time to Data Loss would be 500 * 10^6 hours, or 57,000 years!

This is the basic idea behind disk mirroring, in which a system contains identical data on two or more disks. Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

3.2 Improvement in Performance via Parallelism

There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.) Another way of improving disk access time is with striping, which basically means spreading data out across multiple disks that can be accessed simultaneously.

With **bit-level** striping the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.

Block-level striping spreads a file system across multiple disks on a block-by block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when file systems are accessed in clusters of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

3.3 RAID Levels (Redundant Array of Independent Disk)

Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different RAID levels, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

- **Raid Level 0** - This level includes striping only, with no mirroring.
- **Raid Level 1** - This level includes mirroring only, no striping.

Raid Level 2 - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is (are) identified.)

Raid Level 3 - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

Raid Level 4 - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

Raid Level 5 - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

Raid Level 6 - This level extends raid level 5 by storing multiple bits of error recovery codes, (such as the Reed-Solomon codes), for each bit position of data, rather than a single parity

bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

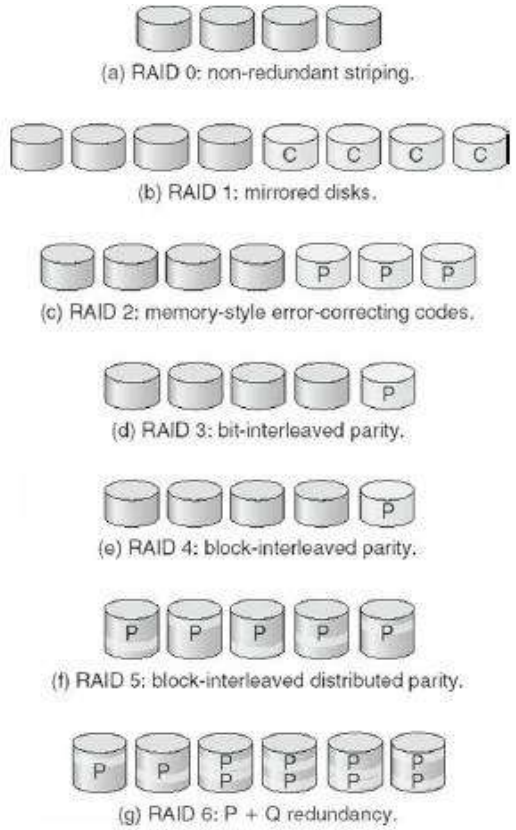


Figure. 3.1. RAID levels.

There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.

- RAID level 0 + 1 disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.

- RAID level 1 + 0 mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.

In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.

- If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.

- However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.

In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.

- If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.

- Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.

- In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

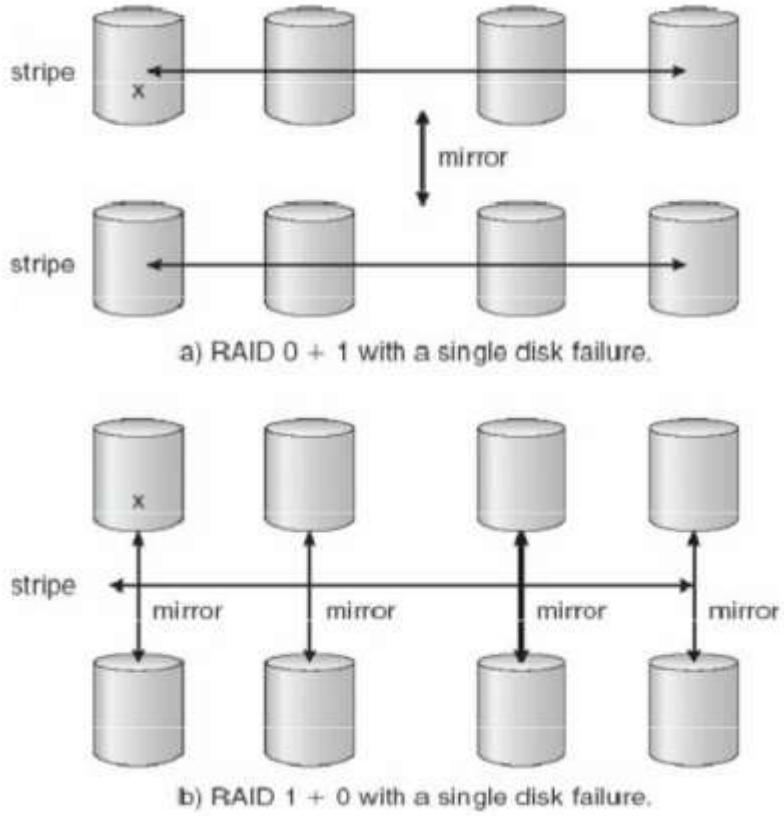


Figure.3.2. RAID 0 + 1 and 1 + 0

3.4 Selecting a RAID Level

Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.

Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

3.4.1 Extensions

RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

3.5 Problems with RAID

RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data. ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

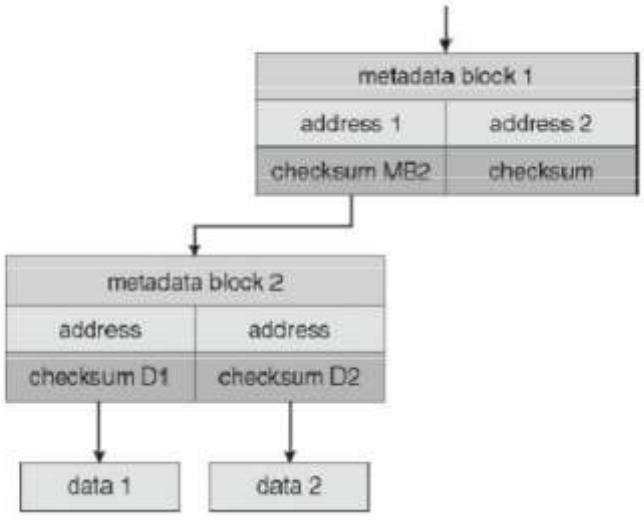


Figure 3.3. ZFS checksums all metadata and data.

Another problem with traditional file systems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust file system sizes, because a file system cannot span across multiple file systems. ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to file systems as needed. File system sizes can be limited by quotas, and space can also be reserved to guarantee that a file system will be able to grow later, but these parameters can be changed at any time by the file system's owner. Otherwise file systems grow and shrink dynamically as needed.

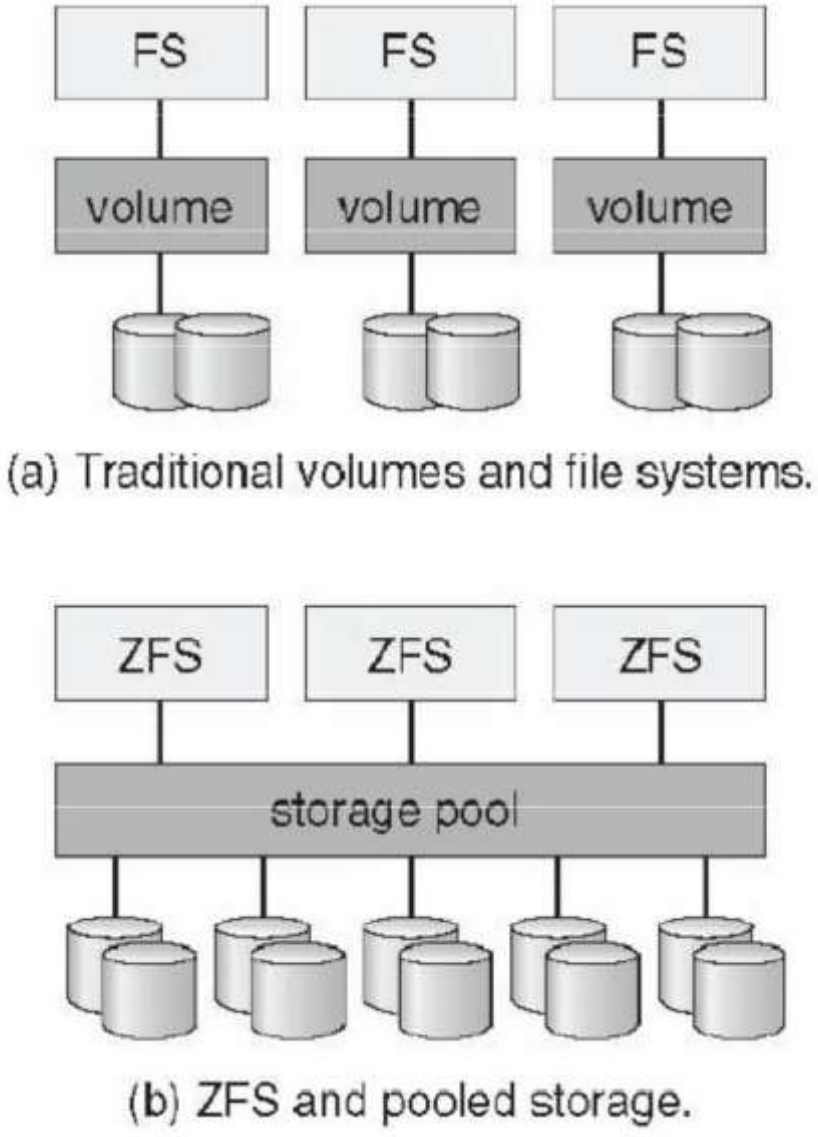


Figure 3.4. (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

4. File Concept

4.1 File Attributes

Different OSes keep track of different file attributes, including:

- Name - Some systems give special significance to names, and particularly
- extensions (.exe, .txt, etc.), and some do not. Some extensions may be of
- significance to the OS (.exe), and others only to certain applications (.jpg)
- Identifier (e.g. inode number)
- Type - Text, executable, other binary, etc.
- Location - on the hard drive.
- Size
- Protection
- Time & Date
- User ID

4.2 File Operations

The file ADT supports many common operations:

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file.

Most OSes require that files be opened before access and closed after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an open file table, containing for example:

- **File pointer** - records the current position in the file, for the next read or write access.
- **File-open count** - How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.
- **Disk location** of the file.
- **Access rights**

Some systems provide support for **file locking**.

- A **shared lock** is for reading only.
- A **exclusive lock** is for writing as well as reading.
- An **advisory lock** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)

- A **mandatory lock** is enforced. (A truly locked door.)
- UNIX used advisory locks, and Windows uses mandatory locks.

4.3 File Types

Windows (and some other systems) use special file extensions to indicate the type of each file:

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

Figure. 4.1. Common File Types

Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call. UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

4.4 File Structure

Some files contain an internal structure, which may or may not be known to the OS. For the OS to support particular file formats increases the size and complexity of the OS. UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.). Macintosh files have two forks - a resource fork, and a data fork. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

4.5 Internal File Structure

• Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)

• Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.

• The number of logical units which fit into one physical block determines its packing, and has an impact on the amount of internal fragmentation (wasted space) that occurs.

• As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

4.6 Access Methods

4.6.1 Sequential Access

• A sequential access file emulates magnetic tape operation, and generally supports a few operations:

O **read next** - read a record and advance the tape to the next position. O
write next - write a record and advance the tape to the next position.
O **rewind**

O **skip n records** - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

Figure 4.2. Sequential File Access

4.6.2 Direct Access

Jump to any record and read that record. Operations supported include:

- read n - read record number n. (Note an argument is now required.)
- write n - write record number n. (Note an argument is now required.)
- jump to record n - could be 0 or the end of file.
- Query current record - used to return back to this record later.
- Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure 4.3. Simulation of sequential access on a direct-access file

4.6.3 Other Access Methods

An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

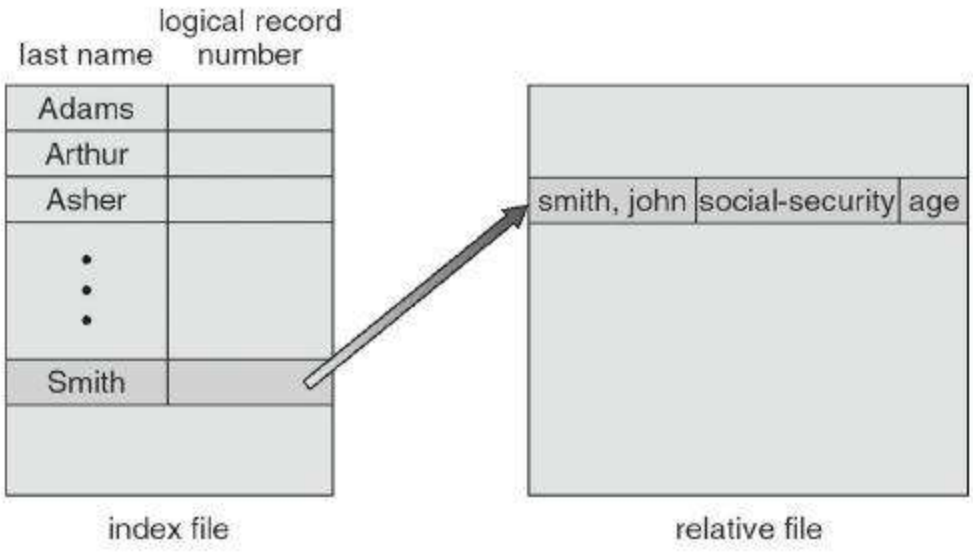


Figure 4.4. Example of index and relative files.

4.7 Directory Structure

4.7.1. Storage Structure

A disk can be used in its entirety for a file system. Alternatively a physical disk can be broken up into multiple partitions, slices, or minidisks, each of which becomes a virtual disk and can have its own file system. (or be used for raw storage, swap space, etc.)

Or, multiple physical disks can be combined into one volume, i.e. a larger virtual disk, with its own file system spanning the physical disks.

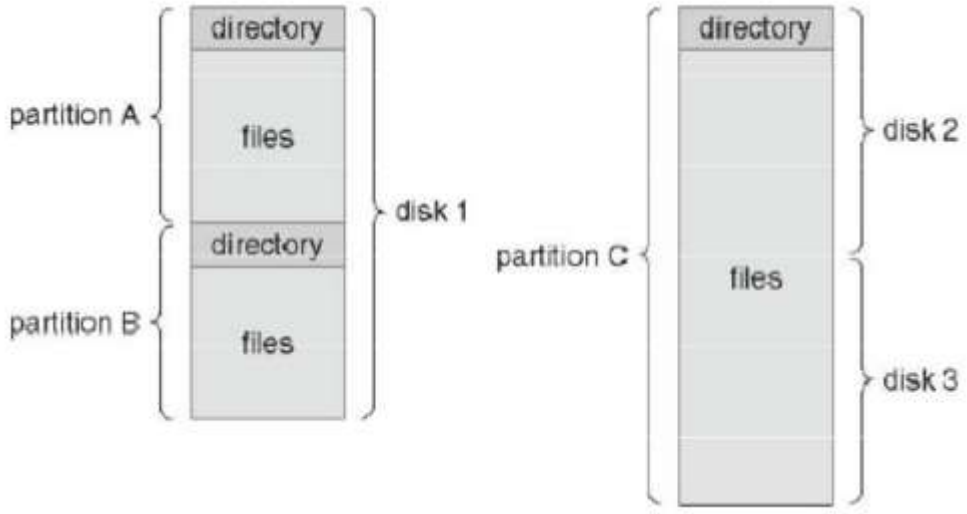


Figure 4.5. A typical file-system organization

4.8 Directory Overview

Directory operations to be supported include:

- Search for a file
- Create a file - add to the directory
- Delete a file - erase from the directory
- List a directory - possibly ordered in different ways.
- Rename a file - may change sorting order

4.8.1 Single-Level Directory

Simple to implement, but each file must have a unique name.

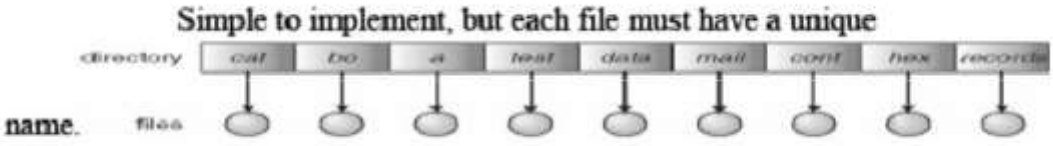


Figure.4.6. Single Level Directory

Draw Backs:
- Naming Problem
- Grouping Problem

4.8.2 Two-Level Directory

Each user gets their own directory space. File names only need to be unique within a given user's directory. A master file directory is used to keep track of each user's directory, and must be maintained when users are added to or removed from the system. A separate directory is generally needed for system (executable) files. Systems may or may not allow users to access other directories besides their own.

If access to other directories is allowed, then provision must be made to specify the directory being accessed. If access is denied, then special consideration must be made for users to run programs located in system directories. A search path is the list of directories in which to search for executable programs, and can be set uniquely for each user.

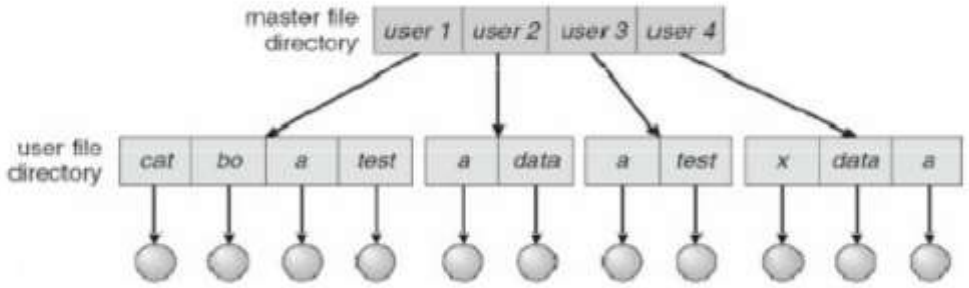


Figure. 4.7. Two-level directory structure.

4.9 Tree-Structured Directories

An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar. Each user / process has the concept of a current directory from which all (relative) searches take place. Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.) Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands. One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

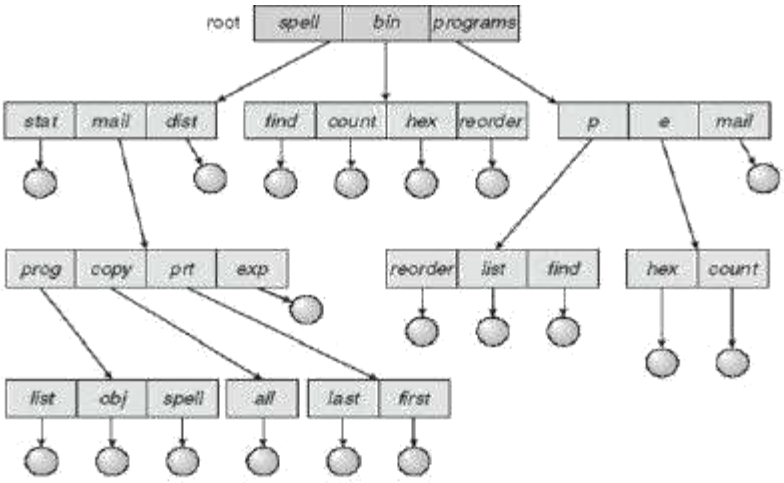


Figure. 4.8. Tree Structured Directory Structure

4.10 Acyclic-Graph Directories

When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child.)

UNIX provides two types of links for implementing the acyclic-graph structure.

- A **hard link** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.

- A **symbolic link** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

Windows only supports symbolic links, termed shortcuts. Hard links require a reference count, or link count for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted: One option is to find all the symbolic links and adjust them also. Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used. What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

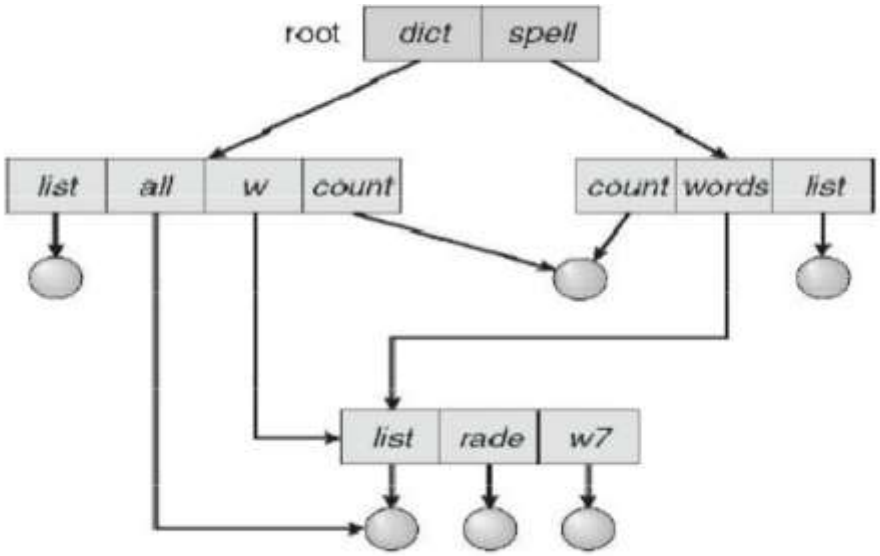


Figure. 4.9. A cyclic-graph Directory Structure

4.11 File-System Mounting

The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure. The mount command is given a file system to mount and a mount point (directory) on which to attach it. Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted. File systems may be mounted read-only, or have other restrictions imposed.

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level. Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found. More recent Windows systems allow file systems to be mounted to any directory in the file system, much like UNIX.

4.12 File Sharing

4.12.1 Multiple Users

On a multi-user system, more information needs to be stored for each file:

- The owner (user) who owns the file, and who can control its access.
- The group of other user IDs that may have some special access to the file.
- What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

4.12.2 Remote File Systems

The advent of the Internet introduces issues for accessing files stored on remote computers. The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or anonymous, not requiring any user name or password.

Various forms of distributed file systems allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)

4.12.3 The Client-Server Model

When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a server, and the system which mounts them is the client.

User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.) The same computer can be both a client and a server. (E.g. cross-linked file systems.) There are a number of security concerns involved in this model: Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk. Servers may restrict remote access to read-only. Servers restrict which file systems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups. The NFS (Network File System) is a classic example of such a system.

4.13 Linux File Structure

All Linux systems have a directory structure that starts at the root directory The root directory is represented by a forward slash, like this: /.

4.13.1 Overview of file structure

Directory	Contents
/	Root directory - the starting point of the directory tree.
/bin	Essential binary files, such as commands that are needed by both the system administrator and normal users. Usually also contains the shells, such as Bash.
/boot	Static files of the boot loader.
/dev	Files needed to access host-specific devices.
/etc	Host-specific system configuration files.
/lib	Essential shared libraries and kernel modules.
/media	Mount points for removable media.
/mnt	Mount point for temporarily mounting a file system.
/opt	Add-on application software packages.
/root	Home directory for the super user root.
/sbin	Essential system binaries.
/srv	Data for services provided by the system.
/tmp	Temporary files.
/usr	Secondary hierarchy with read-only data.
/var	Variable data such as log files
/windows	Only available if you have both Microsoft Windows* and Linux installed on your system. Contains the Windows data.

5. FTP (File Transfer Protocol)

File Transfer Protocol is a standard network protocol used to exchange and manipulate files over a TCP/IP-based network, such as the Internet. FTP is built on client-server architecture and utilizes separate control and data connections between the client and server applications. FTP is used with user-based password authentication or with anonymous user access.

5.1 Features of FTP

The basic features of FTP are:

1. Data representation

- FTP handles three types of data representations-ASCII (7 bit), EBCDIC (8-bit) and 8-binary data.
- The **ASCII file** is the default format for transferring text files
- Each character is encoded using 7-bit ASCII. The sender transforms the file from its own representation into ASCII characters and the receiver transforms the ASCII character to its own representation.
- The **image file** is the default format for transferring binary files. The file is sent as continuous streams of bits without any interpretation or encoding.

2. File organization and Data structures

- FTP supports both unstructured and structured file.
- An unstructured file contains string of bytes and is enl-marked by EOF (End of file). The data structure that corresponds to such a file is called file structure.
- A structured file contains a list of records and each record is delimited by EDR (End of Record).
- The data structure of such file is called record structure i.e. file is divided into records.

3. Transmission modes

FTP can transfer a file by using one of the following three modes:

Stream mode

- It is the default mode.
- File is transmitted as continuous stream of bytes to TCP.
- TCP is responsible for chopping data into segments of appropriate size.
- If data is simply a stream of bytes (file structure), no end-of-file is needed. EOF in this case is the closing of the data connection by the sender.
- If data is divided into records (record structure), each record has a I-byte EOR (End-of-Record) character and the end of the file has a I-byte EOF (End-of-file) character.

- Data is delivered from FTP to TCP in blocks.
- Each block is preceded by 3 bytes header.

- The first byte is called the block descriptor.
- The second and third byte defines the size of the block in bytes.

Compressed mode

- Data is usually compressed if the file to be transmitted is very big.
- The compression method normally used in Run-length encoding.
- In a text file, usually spaces (blanks) are removed.
- In a binary file, null characters are compressed.

5.2 FTP operation

- FTP uses client/server model for communication.
- Two TCP connections are used for file transfer.
- On one connection control signals (commands and responses) are exchanged and the other connection is used for actual data transfer. These two connections are called control connection and data connection respectively.

5.3 Telnet

TELNET (**TEL**ecommunication **NET**work) is a network protocol used on the Internet or local area network (LAN) connections. Telnet (TN) is a networking protocol and software program used to access remote computers and terminals over the Internet or a TCP/IP computer network. It is a network protocol used on the Internet or local area networks to provide a bidirectional interactive communications facility. Typically, telnet provides access to a command-line interface on a remote host via a virtual terminal connection which consists of an 8-bit byte oriented data connection over the Transmission Control Protocol (TCP). User data is interspersed in-band with TELNET control information. The user's computer, which initiates the connection, is referred to as the local computer. Telnet was developed in 1969 to aid in remote connectivity between computers over a network. Telnet can connect to a remote machine that on a network and is port listening. Most common ports to which one can connect to through telnet are:

Port 21 - File Transfer Protocol
Port 22 - SSH Remote Login Protocol
Port 23 - Telnet Server
Port 25 - Simple Mail Transfer Protocol (SMTP)
Port 53 - Domain Name Server (DNS)
Port 69 - Trivial File Transfer Protocol (TFTP)
Port 70 - Gopher
Port 80 - Hyper Text Transfer Protocol (HTTP)
Port 110 - Post Office Protocol 3 (POP3)

5.4 Telnet Protocol Characteristics

Telnet is a terminal emulation protocol. When you start installing and configuring native TCP/IP devices, you are going to need some way to connect to the device to issue its commands. Telnet is versatile. You can establish Telnet sessions over the phone. If there is no phone connection and your device is accessible to the Internet, you can establish a Telnet session over the Internet. In any of these conditions you can establish a Telnet session with a remote host.

6. Shell Programming

6.1 About Shell

Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell. Shell accepts your instruction or commands in English and translates it into computers native binary language.

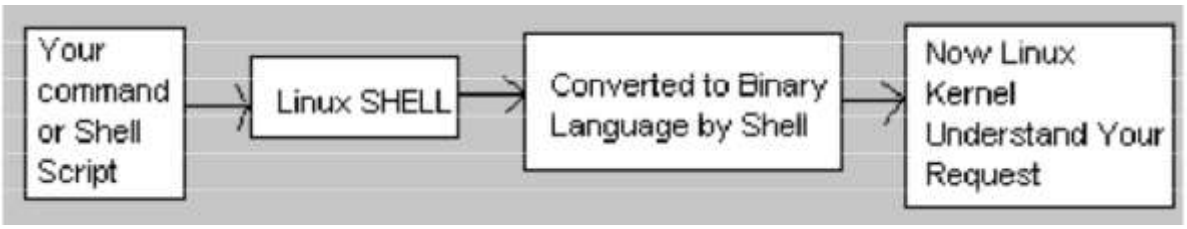


Figure. 6.1 About Shell

The command you type converted in the shell as it shown in the following manner.

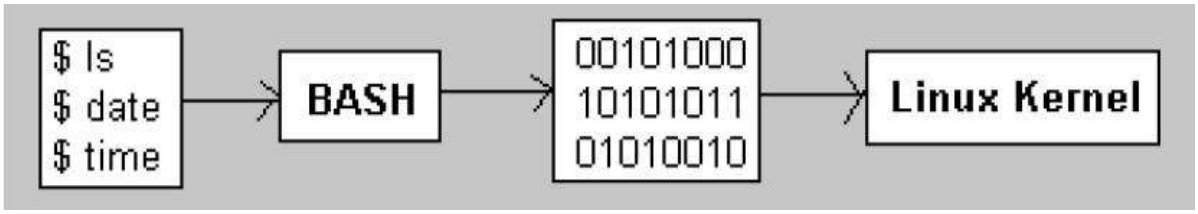


Figure. 6.2. Shell Conversion

It's environment provided for user interaction. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Linux may use one of the following most popular shells (In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!)

6.2 Shell Types

Shell Name	Developed by	Where	Remark
BASH (**B**ourne-Again **SH**ell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (**C SH**ell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (**K**orn **SH**ell)	David Korn	AT & T Bell Labs	

To find your shell type following command

$ echo $SHELL

6.3 Linux Basic Commands

$pwd

Use the **pwd** command to find out the path of the current working directory (folder) you're in. The command will return an absolute (full) path, which is basically a path of all the directories that starts with a forward slash **(/)**. An example of an absolute path is **/home/username**.

$ cd

To navigate through the Linux files and directories, use the **cd** command. It requires either the full path or the name of the directory, depending on the current working directory that you're in.

There are some shortcuts to help you navigate quickly:

- **cd ..** (with two dots) to move one directory up
- **cd** to go straight to the home folder
- **cd-** (with a hyphen) to move to your previous directory

$ls

The **ls** command is used to view the contents of a directory. By default, this command will display the contents of your current working directory.

If you want to see the content of other directories, type **ls** and then the directory's path.

For example, enter **ls /home/username/Documents** to view the content of **Documents**.

There are variations you can use with the **ls** command:

- **ls -R** will list all the files in the sub-directories as well
- **ls -a** will show the hidden files
- **ls -al** will list the files and directories with detailed information like the permissions, size, owner, etc.

$cat

cat (short for concatenate) is one of the most frequently used commands in Linux. It is used to list the contents of a file on the standard output (sdout). To run this command, type **cat** followed by the file's name and its extension. For instance: **cat file.txt**.

Here are other ways to use the **cat** command:

- **cat > filename** creates a new file

- **cat filename1 filename2>filename3** joins two files (1 and 2) and stores the output of them in a new file (3)
- to convert a file to upper or lower case use, **cat filename | tr a-z A-Z >output.txt**

$cp

Use the **cp** command to copy files from the current directory to a different directory. For instance, the command **cp scenery.jpg /home/username/Pictures** would create a copy of **scenery.jpg** (from your current directory) into the **Pictures** directory.

$mv

The primary use of the **mv** command is to move files, although it can also be used to rename files.

The arguments in mv are similar to the cp command. You need to type **mv**, the file's name, and the destination's directory. For example: **mv file.txt /home/username/Documents**.

To rename files, the Linux command is **mv oldname.ext newname.ext**

$mkdir

Use **mkdir** command to make a new directory — if you type **mkdir Music** it will create a directory called **Music**.

There are extra **mkdir** commands as well:

- To generate a new directory inside another directory, use this Linux basic command **mkdir Music/Newfile**
- use the **p** (parents) option to create a directory in between two existing directories. For example, **mkdir -p Music/2020/Newfile** will create the new "2020" file.

$rmdir

If you need to delete a directory, use the **rmdir** command. However, rmdir only allows you to delete empty directories.

$rm

The **rm** command is used to delete directories and the contents within them. If you only want to delete the directory — as an alternative to rmdir — use **rm -r**.

Note: Be very careful with this command and double-check which directory you are in. This will delete everything and there is no undo.

$grep

Another basic Linux command that is undoubtedly helpful for everyday use is **grep**. It lets you search through all the text in a given file.

To illustrate, **grep blue notepad.txt** will search for the word blue in the notepad file. Lines that contain the searched word will be displayed fully.

$chmod

chmod is another Linux command, used to change the read, write, and execute permissions of files and directories.

In general, **chmod** commands take the form:

chmod options permissions file name

If no options are specified, **chmod** modifies the permissions of the file specified by file name to the permissions specified by permissions.

permissions defines the permissions for the **owner** of the file (the "user"), members of the **group** who owns the file (the "group"), and anyone else ("**others**"). There are two ways to represent these permissions: with symbols (alphanumeric characters), or with octal numbers (the digits 0 through 7).

Let's say you are the owner of a file named myfile, and you want to set its permissions so that:

- the **user** can read, write, and execute it;
- members of **your group** can read and execute it; and
- **others** may only read it.

This command will do the trick:

chmod u=rwx,g=rx,o=r myfile

This example uses symbolic permissions notation. The letters **u**, **g**, and **o** stand for "**user**", "**group**", and "**other**". The equals sign ("=") means "set the permissions exactly like this," and the letters "**r**", "**w**", and "**x**" stand for "read", "write", and "execute", respectively. The commas separate the different classes of permissions, and there are no spaces in between them.

Here is the equivalent command using octal permissions notation:

chmod 754 myfile

Here the digits **7**, **5**, and **4** each individually represent the permissions for the user, group, and others, in that order. Each digit is a combination of the numbers **4**, **2**, **1**, and **0**:

- **4** stands for "read",
- **2** stands for "write",
- **1** stands for "execute", and
- **0** stands for "no permission."

So **7** is the combination of permissions **4**+**2**+**1** (read, write, and execute), **5** is **4**+**0**+**1** (read, no write, and execute), and **4** is **4**+**0**+**0** (read, no write, and no execute).

$chown

In Linux, all files are owned by a specific user. The **chown** command enables you to change or transfer the ownership of a file to the specified username. For instance, **chown linuxuser2 file.ext** will make **linuxuser2** as the owner of the **file.ext**.

$kill

If you have an unresponsive program, you can terminate it manually by using the **kill** command.
It will send a certain signal to the misbehaving app and instructs the app to terminate itself.

kill [signal option] PID.

$ping

Use the **ping** command to check your connectivity status to a server. For example, by simply entering **ping google.com**, the command will check whether you're able to connect to Google and also measure the response time.

$history

When you've been using Linux for a certain period of time, you'll quickly notice that you can run hundreds of commands every day. As such, running **history** command is particularly useful if you want to review the commands you've entered before.

Ctrl+C and **Ctrl+Z** are used to stop any command that is currently working. Ctrl+C will stop and terminate the command, while Ctrl+Z will simply pause the command.

6.4 Variables in Linux

Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time). In Linux, there are two types of variable

1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower LETTERS.

```
$ no=10
$ vech=Bus
```

Variables are case-sensitive, just like filename in Linux.

6.5 Operators in Linux

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

6.5.1Arithmetic Operators :

These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:

- **Addition (+)**: Binary operation used to add two operands.
- **Subtraction (-)**: Binary operation used to subtract two operands.
- **Multiplication (*)** :Binary operation used to multiply two operands.
- **Division (/)** :Binary operation used to divide two operands.
- **Modulus (%)** :Binary operation used to find remainder of two operands.
- **Increment Operator (++)** : Uniary operator used to increase the value of operand by one.
- **Decrement Operator (–)** : Uniary operator used to decrease the value of a operand by one

6,5,2 Relational Operators :

Relational operators are those operators which defines the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:

- **‘==’ Operator** : Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
- **‘!=’ Operator** : Not Equal to operator return true if the two operands are not equal otherwise it returns false.
- **‘<' Operator** : Less than operator returns true if first operand is lees than second operand otherwse returns false.
- **‘<=' Operator** : Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- **‘>’ Operator** : Greater than operator return true if the first operand is greater than the second operand otherwise return false.

- **'>=' Operator** : Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

6.5.3 Logical Operators :

They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

- **Logical AND (&&)** : This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)** : This is a binary operator, which returns true is either of the operand is true or both the operands are true and returns false if none of then is false.
- **Not Equal to (!)** : This is a uninary operator which returns true if the operand is false and returns false if the operand is true.

6.5.4 Bitwise Operators : A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

- **Bitwise And (&)** : Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|)** : Bitwise | operator performs binary OR operation bit by bit on the operands.
- **Bitwise XOR (^)** : Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise compliment (~)** : Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<)** : This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>)** : This operator shifts the bits of the left operand to right by number of times specified by right operand.

6.6 How to write shell script

Now we write our first script that will print "Knowledge is Power" on screen. To write shell script you can use in of the Linux's text editor such as vi or mcedit or even you can use cat command. Here we are using cat command you can use any of the above text editor. First type following cat command and rest of text as its

```
        cat > first
        #
        # My first shell script
#
        clear
        echo "Knowledge is Power"
```

Press Ctrl + D to save. Now our script is ready.

To execute it type command **$ ./first**

This will give error since we have not set Execute permission for our script first; to do this type command

$ chmod +x first

$ ./first

First screen will be clear, then Knowledge is Power is printed on screen.

6.7 How to Run Shell Scripts

Because of security of files, in Linux, the creator of Shell Script does not get execution permission by default. So if we wish to run shell script we have to do two things as follows

(1) Use chmod command as follows to give execution permission to our script
Syntax: *chmod +x shell-script-name*
OR Syntax: *chmod 777 shell-script-name*

(2) Run our script as
Syntax: *./your-shell-program-name*

For e.g.
$ ./first

Here '.'(dot) is command, and used in conjunction with shell script. The dot(.) indicates to current shell that the command following the dot(.) has to be executed in the same shell i.e. without the loading of another shell in memory.