OPERATING SYSTEM CONCEPTS AND BASIC LINUX COMMANDS

SHITAL VIVEK GHATE

Operating System Concepts
and Basic Linux Commands

Publishing-in-support-of,

EDUCREATION PUBLISHING

RZ 94, Sector - 6, Dwarka, New Delhi - 110075
Shubham Vihar, Mangla, Bilaspur, Chhattisgarh - 495001

Website: *www.educreation.in*

ISBN: 978-1-5457-0850-7
Price: ₹ 275.00

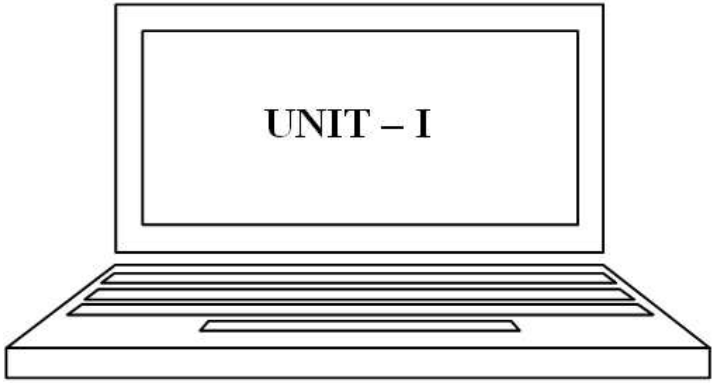
Printed in India

Operating System Concepts and Basic Linux Commands

Shital Vivek Ghate

EDUCREATION PUBLISHING
(Since 2011)
www.educreation.in

Content List



Introduction To Operating System

1.1 Introduction

In the simplest scenario, the operating system is the first piece of software to run on a computer when it is booted. Its job is to coordinate the execution of all other software, mainly user applications. It also provides various common services that are needed by users and applications.

An operating system is a program which acts as an interface between a user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user may execute program.

An operating system acts as a resource manager and allocates resources to specific programs and users as necessary for their task. The commonly required resources are Input/output devices, memory, file storage space, CPU time and so on.

The operating system must ensure the correct operation of the computer system. The hardware must provide appropriate mechanisms to prevent user programs from interfering with the proper operation of the system

The operating system controls and coordinates the use of the hardware among the various application programs for the various users. The operating system provides the means for the proper use of the recourses in the operations of the computer system. The operating system performs no useful function by itself. It simply provides an environment within which other programs can do useful work. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their task. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system efficiently.

An operating system is a control program. A control program controls the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

The fundamental goal of computer system is to execute user programs and solve user problems. The primary goal of an operating system is convenient for user.

1.2 The operating system performs resource management

One of the main features of operating systems is support for multiprogramming. This means that multiple programs may execute "at the same time". But given that there is only one processor, this concurrent execution is actually a fiction. In reality, the operating system juggles the system's resources between the competing programs, trying to make it look as if each one has the computer for itself. At the heart of multiprogramming lies resource management deciding which running program will get what resources. Resource management is akin to the short blanket problem: everyone wants to be covered, but the blanket is too short to cover everyone at once.

The resources in a computer system include the obvious pieces of hardware needed by programs:

- The CPU itself.
- Memory to store programs and their data.
- Disk space for files. But there are also internal resources needed by the operating system:
- Disk space for paging memory.
- Entries in system tables, such as the process table and open files table.

All the applications want to run on the CPU, but only one can run at a time. Therefore the operating system lets each one run for a short while, and then preempts it and gives the CPU to another. This is called time slicing. The decision about which application to run is scheduling.

As for memory, each application gets some memory frames to store its code and data. If the sum of the requirements of all the applications is more than the avail-able physical memory, paging is used: memory pages that are not currently used are temporarily stored on disk..

With disk space (and possibly also with entries in system tables) there is usually a hard limit. The system makes allocations as long as they are possible. When the resource runs out, additional requests are failed. However, they can try again later, when some resources have hopefully been released by their users.

1.3 Structure of operating system:

The structure of operating system consist of 4 layers

- The hardware
- The operating system
- The system program
- The application program

The application program The hardware parts consist of CPU, memory, I/O devices and secondary storage. Above the hardware layer there is operating system Program. Third layer is the system program which consists of compilers., assembler, linker etc. . Finally last layer is application programs through which users can interact with computer system, this consist of database system, video games, business programs etc (depending on users interest).

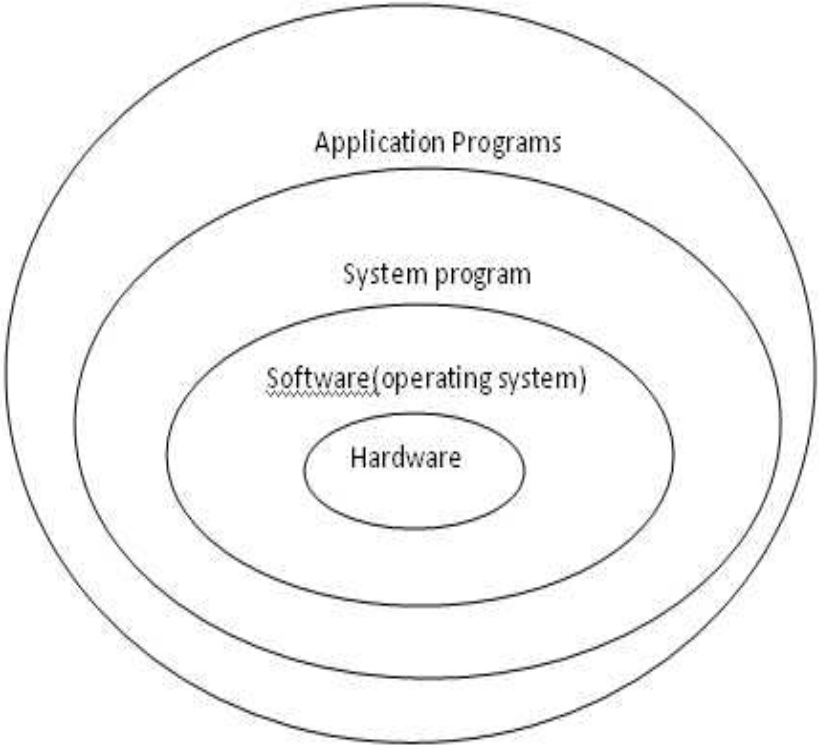


Fig 1.1 Layer View of Structure of Operating System

1.4 Components of computer system:

A computer system is a collection of hardware and software components. Hardware refers to the physical computing equipment. Software refers to the programs written to provide services to the users of the system.

Every computer system consists of four basic components. Those are hardware (Memory, central processing unit and the Input-output unit), operating system, system program, application program. The basic organization of hardware is depicted in fig 1.2.

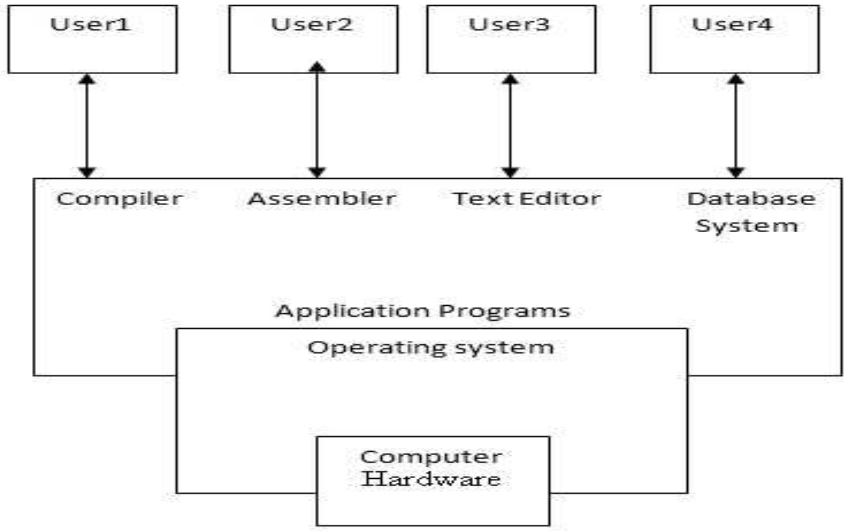


Fig: Abstract view of components of a computer

The hardware provides the basic computing resources. The application program defines the way in which these resources are used to solve the computing problems of the users. The system programs consist of compilers, assemblers, linkers etc. , the operating system controls and coordinates the use of the hardware among the various application programs for the various users. Operating system provides an environment within which other programs can do useful work and controls I/O devices. The fundamental goal of computer system is to execute user programs and solve user problems.

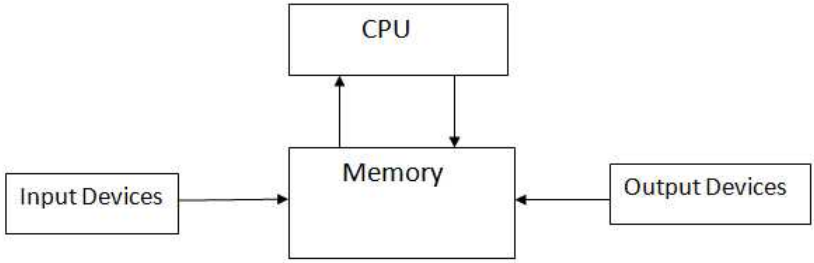


Fig: 1.2 Basic organization of H/W in computer system

1.5 Services Provided By Operating System:

An operating system provides an environment for the execution of programs. The operating system provides certain services to programs and to the users of those programs. The operating system functions provided for the convenience of the programmer to make the programming task easier are as follows:

1. Program Execution:

Users will want to execute programs. The system must be able to load a program into memory and run it. The program must be able to end its execution either normally or abnormally.

2. Input/output operations:

A running program may require input and output. This I/O may involve a file or an I/O device. Since a user program cannot execute I/O operations directly, the operating system must provide some means to do so.

3. File system manipulation:

It should be obvious that user want to read and write files, also want to create and delete files by name, operating system provides all the file manipulation operations like create a file, read a file, write to a file, delete a file.

4. Error detection:

The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware such as a memory error or power failure and in Input/output devices such as a printer out of paper or in the user program such as an arithmetic overflow or access to illegal memory location.

Following set of operating system functions exist for the operation of the system itself.

5. Resource allocation:

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system.

6. Accounting:

In multi-user system, operating system keep track of which user uses how many and which kind of computer resources.

7. Protection:

In multi-user system, when jobs of more than one user executed simultaneously, it should not be possible for one job to interfere with the other.

1.6 Types of Operating System:

There are many different types of Operating system depending upon number of users, processors, and way of executing programs in memory.

A) Serial Processing System:

In early computer system, there was only computer hardware and didn't have operating system. Early computers were physically very large machines runs from console. Programming in 1's and 0's (machine language) was quite common. Instruction and data used to be fed into the computer by means of console switches or perhaps through hexadecimal keyboard, paper tapes or punched cards. Then the appropriate buttons would be pushed to load the starting address and start the execution of the program . as the

program ran, the programmer/operator could monitor its execution by the display lights on the console. If error occurs, the error condition was indicated by the lights. The programmer examines the registers and main memory to identify the cause of the error. When execution finished take the output on the printer and then the programmer was ready for next program to execute.

This type of processing is difficult for users, it takes much time and next program should wait for the completion of previous one. The programs are submitted to the machine one after other, so this method is said to be serial processing.

B) Batch Processing System:

In batch processing system jobs with similar needs are batched together as a group and run through the computer as group.

In early systems compilers and assemblers were normally kept on magnetic tapes. To execute particular language program user had to mount that particular compiler first which produces assembly language code which then need to be assembled. This required mounting another tape with assembler.

During the time that tapes are being mounted or the programmer is operating the console, the CPU sits idle. As in the early days there were very few computers and they were very expensive. The computer time was very valuable and the owner of the computer wanted them to be used as much as possible.

Then the first professional computer operators were hired. Since an operator had more experience with mounting tapes than programmer, setup time was reduced. Again to reduce setup time jobs with similar needs were batched together and run through the computer as a group.

Suppose for example the operator received one FORTRAN program, one COBOL program, and another FORTRAN program. If he runs them in that order, he would have to set up for FORTRAN program environment (loading FORTRAN compiler tapes) then setup COBOL program and finally setup for FORTRAN program again. If he runs the two FORTRAN programs as batch, he could setup only once for FORTRAN, thus saving operator's time.

Resident monitor:

In batch processing during the transition from one job to the next job the CPU sits idle. To overcome this idle time automatic job sequencing is introduced and with it the first rudimentary operating system is created. It contains the procedure for automatically transferring control from one job to the next job. A small program called resident monitor is created for this purpose. The resident monitor is always in memory.

When the computer is turned on the control of the computer system resides in resident monitor, which then transfer it to the program. When the program terminated, it returns control to the resident monitor, which then go on to the next program. Thus the resident monitor will automatically sequence from one program to another and form one job to another.

The operator had been given a short description of what programs were to run on what data. To provide this information directly to the resident monitor control cards were introduced. In addition to the program or data for a job, programmer include special cards (control cards) which are directives for the resident monitor indicating what program is to be run. For example, a normal user program might require one of three programs to run, the FORTRAN compiler (FTN), the assembler (ASM), or the user program. There is a separate control cards for each of these:

$FTN : Execute the FORTRAN compiler.
$ASM : Execute the assembler
$RUN : Execute the user program control card for each of these:

In addition, there are two control cards to define the boundaries of each job:

$JOB : First card of job.
$END : Last card of job.

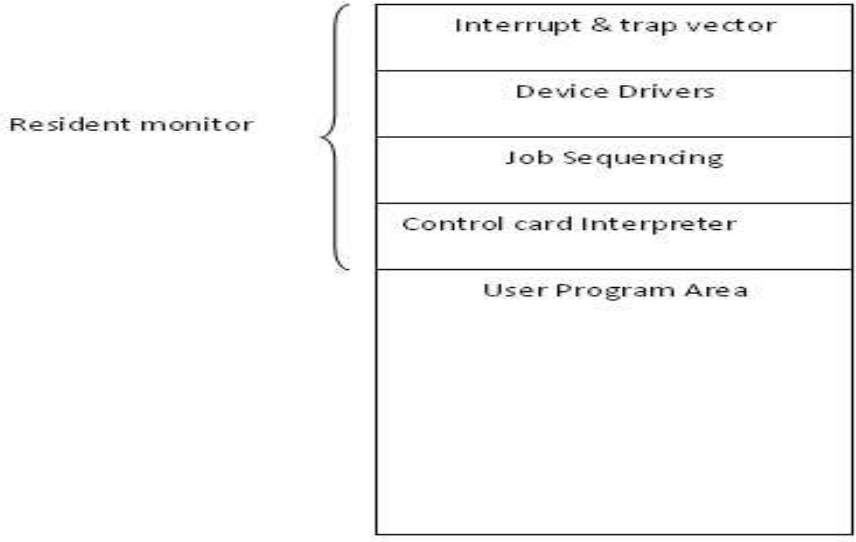


Fig 1.3 Memory Layout for Resident Monitors

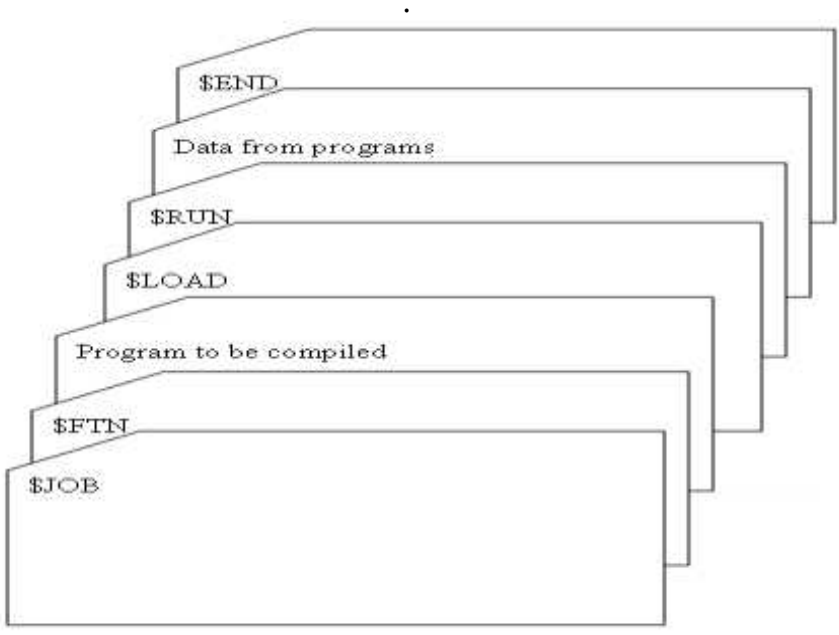


Fig: 1.4 Card deck for simple batch system

Buffering:

Buffering attempts to keep both the CPU and the I/O devices busy all the time. Buffer is a part of memory, used to store data temporarily. After data has been read and the CPU is about to start operating on it, the input device is instructed to begin the next input immediately. The CPU and input devices are then busy. When CPU is ready for the next data item, the input device will have finished reading it. The CPU can then begin processing the newly read data, while the input device starts to read the following data.

Similar buffering can be done for output; the CPU creates data which is put into a buffer until an output device can accept it.

Buffering keeps both CPU and I/O devices busy all the time. If the CPU is working on one record while an input device is working on another, either the CPU or input device will finished first. If the CPU finished first, it must wait; it can not proceed until the next record is read. If the input device is faster than CPU the buffer will become full and input device must wait.

The solution to this problem is that the next I/O operation can only be started when the previous one has finished. Interrupts solve this problem. When I/O device is finished with an operation, it interrupt the CPU , then CPU stops what it is doing and immediately transfer to a fixed location, where the interrupt service routine checks to see if the buffer is not full(for an input device) or empty(for an output device), then starts the next I/O request. The CPU can then resume the interrupt computation. In this way, I/O devices and CPU can be operated in full speed.

If the CPU is much faster than an input device, buffering is of little use. If the CPU is always faster, then it will always find an empty buffer and have to wait for the input device. For output, the CPU can proceed at full speed until, all system buffers are full. Then the CPU must wait for the output device.

Spooling:

The spooling stands for simultaneous peripheral operation on-line. Buffer is a small part of memory used to store data temporarily during program execution. In spooling disk is used as large buffer.

For example, if two or more users issue the print command and at the same time if printer printing some other job, then issued print commands will be loaded into the spool disk. Spool disk is a temporary buffer; it can read data directly from secondary storage devices. While printing the output of some other job at the same time CPU may execute some other job in the spool disk. Thus at the same time user is taking input through keyboard for one job, CPU is busy with executing other job and printer is busy with taking output of some other job.

The advantage of spooling over buffering is that spooling overlaps the I/O of one job with computation of other job. The spooler may read the input of one job while printing the output of a Different job. During this time still another job may be executed. Buffering can only overlap the I/O of a job with its own computation and III/O; spooling can overlap the I/O and computation of many jobs.

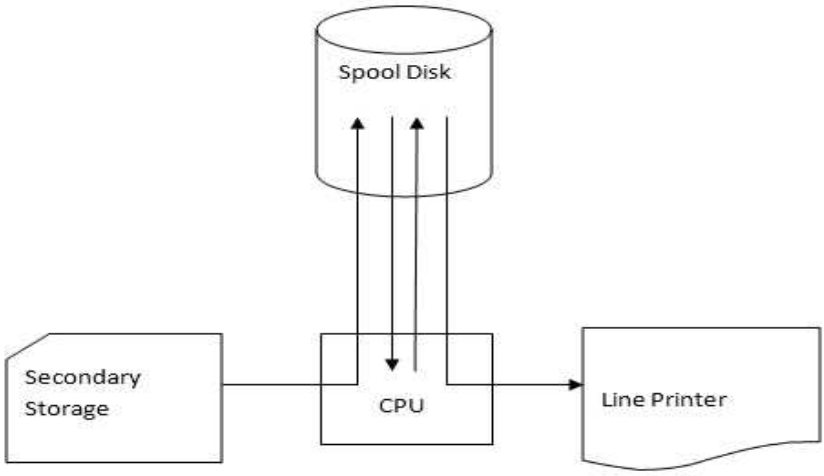


Fig: 1.5 Spooling

C) Single Processor System:

A uniprocessor system is defined as a computer system that has a single central processing unit that is used to execute instructions. Uniprocessor system (Standard personal computer) contains a single CPU and single I/O channels and is most often used as a

single user machines and can also be implemented as a multi-user system.

When used for multi-user system, applications of all users are protected from each other, system resources are shared among many users and operating system maintains control over resources, uses following modes of operations

1. The microkernel, which contains kernel mode code yet is small, modular, executes quickly, etc.
2. The user-mode modules for managing resources. It is often true that portions of the operating system's functionality can be executed outside of kernel mode. For example, the printing system can be managed entirely as a user-level subsystem without adversely affecting the operation of the system as a whole.

Communication in this model is via message passing (even though shared memory may be available).

The main advantage of this approach is its flexibility. It's relatively easy to replace modules, providing performance upgrades, etc. User-level modules can theoretically be moved to other machines. The implication there is that this model is more suitable to being adopted as a distributed OS. In fact, its message-passing communication, which is one of its disadvantages, is another indicator of its suitability for distributed implementation.

Its disadvantages have to do with familiarity and performance. The familiarity aspect is that it's not. People fear change. The performance has to do with the message passing in the form of extra communication (among somewhat separated modules, instead of a unified monolithic structure), which results in a performance loss.

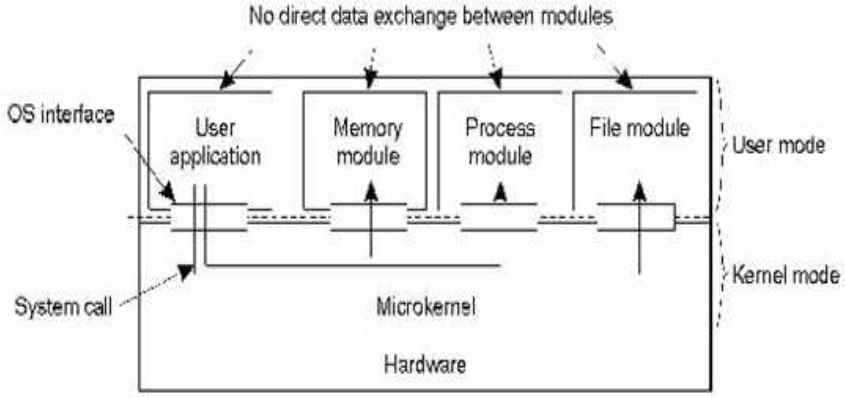


Fig: 1.6 Uniprocessor Operating System

D) Multi Processor System:

The standard system is uniprocessor, that is, it has one main CPU. A multiprocessor system would have more than one CPU, sharing memory and peripherals.

There are two approaches used for multiprocessor operating system.

Master/slave approach:

The most common multiprocessor systems assign to each processor a specific task. There is a one master computer. This master computer controls the system. The other processors (slaves) either look to the master for instruction or have predefined task. This scheme defines a master/slave relationship.

Computer Network:

In this approach, multiple independent computer systems can communicate, sending files and information between them. However, each computer system has its own operating system and operates independently.

One advantage is throughput. By increasing the number of processors, it would get more work done in a shorter period of time.

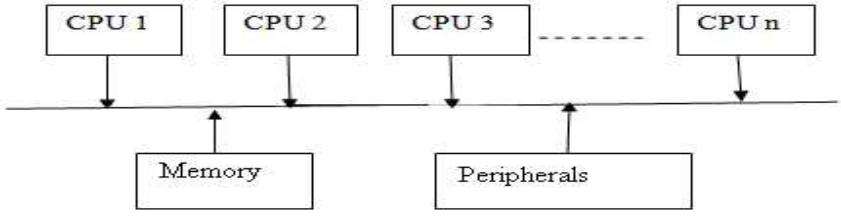


Fig: 1.7 Multiprocessor Operating System

Another advantage is its reliability. If functions can be properly distributed among several processors, then the failure of one processor will not halt the system, but only slow it down. If we have ten processors and one fails, then each of the remaining nine processors must pick up a share of the work of failed processor. Thus the entire system runs only 10 percent slower, rather than failing altogether.

Designing a multiprocessor system is more difficult than designing a single processor system. Multiprocessor systems have three main advantages:

1. Increased throughput.

By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with *N* processors is not *N*, however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, *N* programmers working closely together do not produce *N* times the amount of work a single programmer would produce.

2. Economy of scale.

Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. Increased reliability

If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

E) Multiprogramming System:

The most important aspect of job scheduling is the ability to multiprogram . A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

In multiprogramming, single processor can execute number of programs simultaneously. In this technique, the physical memory is divided into many partitions, each holding a separate program. One of these partitions is holding the operating system.

As there is only one CPU, only one program could be executed at a time. Operating system provides a mechanism to switch the CPU from one program to the next. The operating system picks one job from memory and start executing it. Eventually, the job may have to wait for a command to be typed on a keyboard, or I/O operations to complete. In a non-multiprogramming system (uniprogramming system) CPU becomes idle. But in multiprogramming system, the operating system will simply switch to another job in the main memory and start executing it. When that job needs to wait, the CPU is switched to another job and so on. Eventually the first job will have finished

waiting and will get the CPU back. As long as there is always some job to execute the CPU will never sit idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.

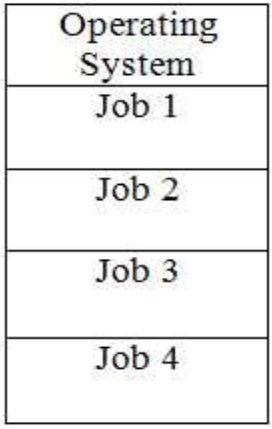


Fig: 1.8 Memory Layout of a Multiprogramming system

The benefits of multiprogramming are increased cpu utilization and higher total job throughput. Throughput is the amount of work done in given time interval.

For example assume that we have two jobs, A and B, to be executed. Each job executes for one second, and then waits for one second. This pattern is repeated 60 times. If we run first job A and then job B, one after the other, it will takes 4 minutes to run the two jobs, job A takes 2 minutes to run and job b takes 2 minutes to run.

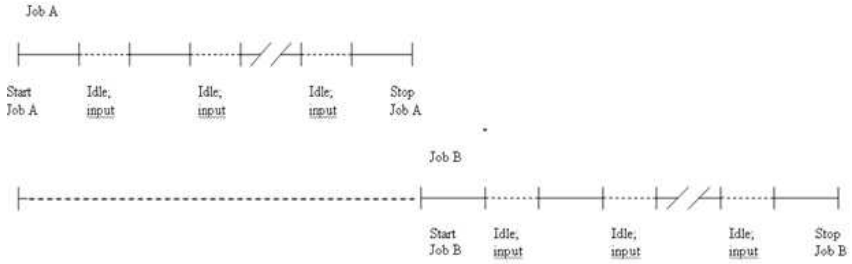


Fig: 1.9 Job Execution Without Multiprogrammin

If we multiprogramming job A and job B, we can greatly improve the system performance. We start with job A, which executes for one second. Then, while job awaits for one second, we execute job B. when job B waits, job A is ready to run. Now the time required to execute both the jobs is only 2 minutes, and there is no idle CPU time. Thus we have increased CPU utilization from 50 to 100 percent, increasing throughput at the same time.

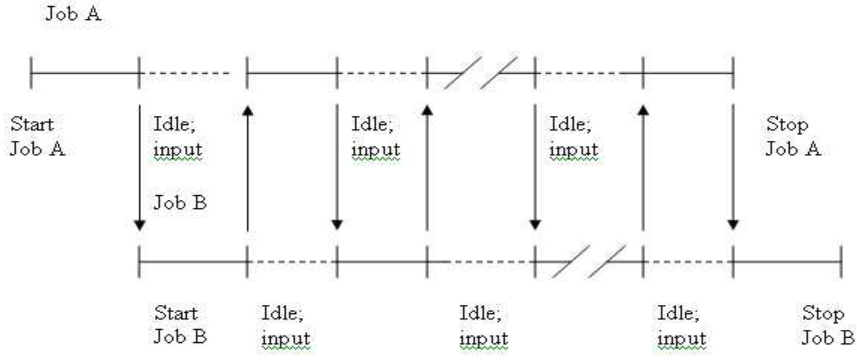


Fig: 1.10 Job Execution with multiprogramming

F) Time Sharing System:

Multiprogrammed batched systems provided an environment in which the various system resources (for example, CPU, memory, peripheral devices) were utilized effectively, but it did not provide for user interaction with the computer system. Time sharing(or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching

among them, but the switches occur so frequently that the users can interact with each program while it is running.

The time shared operating system uses CPU scheduling and multiprogramming. Each user has a separate program in memory, and each user process is executed for a small time unit called time slice or time quantum.

A time shared operating system allows the many users to simultaneously share the computer. Since each users job in time-shared system is executed only for short CPU time, and when that time expired the system switches rapidly from one user to the next user. Users are given the impression that they each have their own computer, while actually one computer is shared among the many users. In this method the CPU time was shared by different processes (or users) so it is said to be "Time shared system"

This system provides efficient CPU utilization. Some examples of time-shared operating system are UNIX, Linux, Multics etc.

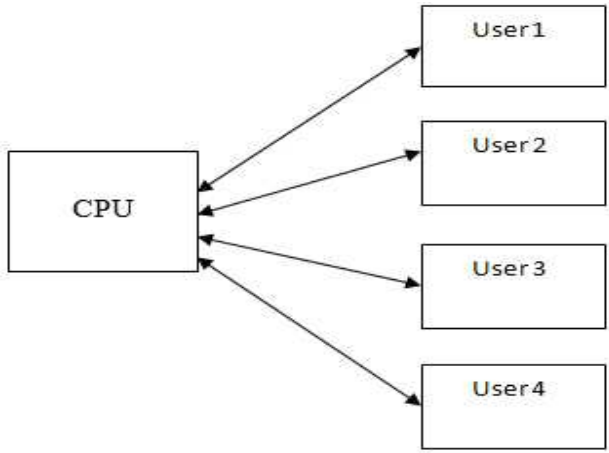


Fig: 1.11 Time-sharing system

G) Multitasking System:

Multitasking operating systems allow more than one program to run at a time. A multi-tasking operating system is an operating system that allows a user to simultaneously run various tasks at the

same time. Actually it is not so because there is only one CPU. The concept behind this is time sharing. The operating system divides CPU time among various tasks, but this time is very small (nanoseconds) & each task is executed for that small time interval & hence the user feels that all programs or tasks are running simultaneously. And switching between the tasks/jobs/processes while they are executing is very fast.

Time sharing, or **multitasking**, is a logical extension of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

A multitasking OS allows you to run multiple processes (tasks) "simultaneously". They do not actually run at the same time, of course, since there is only one CPU. What happens is that one process runs for a while, and then the OS breaks in (through an interrupt), stores away the state (context) of the current process, restores the context of another, and allows that other process to run for a while, etc. UNIX and MULTICS both are multitasking operating systems. But they ran on more expensive hardware. MS-DOS is an example of a non-multitasking OS: as long as you're playing

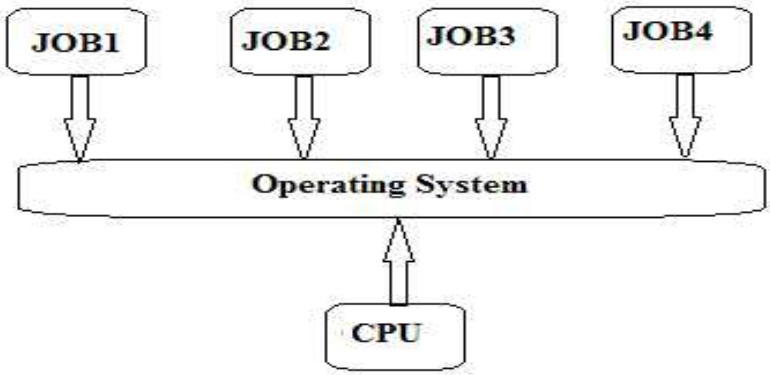


Fig: 1.12 Multitasking operating system

H) Parallel Processing System:

The parallel processing is the simultaneous use of more than one CPU or processor core to execute a program or multiple computational threads. Ideally, parallel processing makes programs run faster because there are more engines (CPUs or Cores) running it. In general, parallel processing means more than one microprocessors handle parts of an overall task. A computer scientist divides a complex problem into component parts using special software specifically designed for the task. He or she then assigns each component part to a dedicated processor. Each processor solves its part of the overall computational problem. The software reassembles the data to reach the end conclusion of the original complex problem

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). With single-CPU, single-core computers, it is possible to perform parallel processing by connecting the computers in a network. However, this type of parallel processing requires very sophisticated software called distributed processing software. Parallel processing is also called parallel computing. The term parallel processing is used to represent a large class of techniques which are used to provide simultaneous data processing tasks for the purpose of increasing the computational speed of a computer system.

Note that parallelism differs from concurrency. Concurrency is a term used in the operating systems and databases communities which refers to the property of a system in which multiple tasks remain logically active and make progress at the same time by interleaving the execution order of the tasks and thereby creating an illusion of simultaneously executing instructions. Parallelism, on the other hand, is a term typically used by the supercomputing community to describe executions that physically execute simultaneously with the goal of solving a problem in less time or solving a larger problem in the same time. Parallelism exploits concurrency

Advantage of Parallel Processing System:-
Faster execution time, so higher throughput.

Disadvantage of Parallel Processing System:-
More hardware required, also more power requirements. Not good for low power and mobile devices.

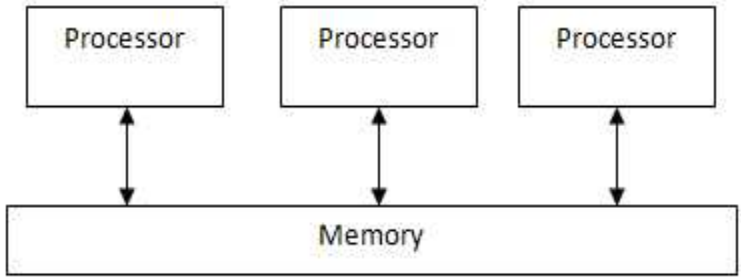


Fig: 1.13 Parallel Processing

I) Distributed system:
A distributed system consists of multiple Computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a distributed program, and distributed programming is the process of writing such programs. A problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by Message passing to In computer networks individual computers were physically distributed within some geographical area. There are several autonomous computational entities, each of which has its own local Memory. The entities communicate with each other by Message passing.

Distributed systems are groups of networked computers, which have the same goal for their work. The processors in a typical distributed system run concurrently in parallel.

Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be

seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors may have access to a Shared memory to exchange information between processors.
- In distributed computing, each processor has its own private memory (Distributed memory). Information is exchanged by passing messages between the processors.

There are two main reasons for using distributed systems and distributed computing. First, the very nature of the application may *require* the use of a communication network that connects several computers. For example, data is produced in one physical location and it is needed in another location.

Second, there are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is *beneficial* for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a Cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can be more reliable than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.

Examples of distributed systems and applications of distributed computing include the following:

Telecommunication networks, telephone networks, cellular networks, computer networks(Internet), wireless sensor networks, distributed database, network file system, world wide web, peer to peer network, Distributed information processing systems such as banking systems and airline reservation systems, Real-time process control.

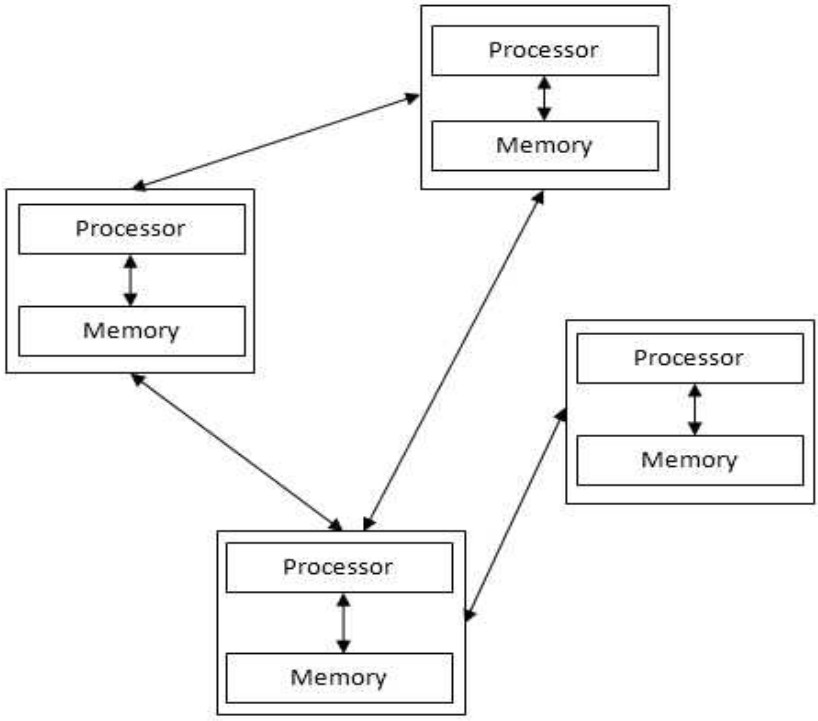


Fig: 1.14 Distributed system.

J) Clustered Systems:

Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together. The definition of the term clustered is not concrete; many commercial packages wrestle with what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via LAN networking.

Clustering is usually used to provide high-availability service; that is, service will continue to be provided even if one or more systems in the cluster fail.

Clustering can be structured asymmetrically or symmetrically. In asymmetric clustering, one machine is in hot-standby modewhile the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In symmetric mode, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

Other forms of clusters include parallel clusters and clustering over a WAN. Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters are usually accomplished by use of special versions of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a distributed lock manager (DLM), is included in some cluster technology.

K) Real Time System:

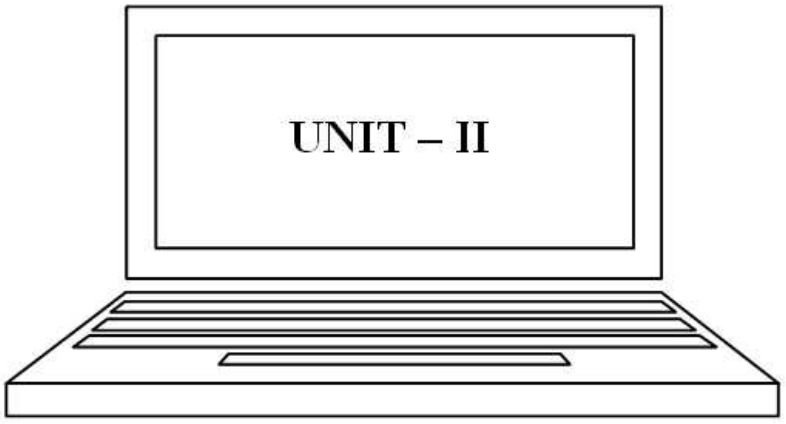
The real time operating system is often used as a control device in a dedicated application. Sensor brings data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. System which controls scientific experiments, medical computer systems, industrial control systems, and some display systems are real time systems. A real time system has well defined fixed time constraints. Processing must be done within the defined constraint or the system will fail.

To be considered "real-time", an operating system must have a known maximum time for each of the critical operations that it performs (or at least be able to guarantee that maximum most of the time). Some of these operations include OS calls and interrupt handling. Operating systems that can absolutely guarantee a

maximum time for these operations are commonly referred to as "hard real-time", while operating systems that can only guarantee a maximum most of the time are referred to as "soft real-time". In practice, these strict categories have limited usefulness - each real time operating system solution demonstrates unique performance characteristics.

A real time operating system can guarantee that a program will run with very consistent timing. Real-time operating systems do this by providing programmers with a high degree of control over how tasks are prioritized, and typically also allow checking to make sure that important deadlines are met.

The main point is that, if programmed correctly, In general, an operating system (OS) is responsible for managing the hardware resources of a computer and hosting applications that run on the computer. A real time operating system performs these tasks, but is also specially designed to run applications with very precise timing and a high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard.



Process & Threads

Introduction

A process is an instance of an application execution. The application may be a program written by a user, or a system application. Users may run many instances of the same application at the same time, or run many different applications. Each such running application is a process. The process only exists for the duration of executing the application.

A user may be able to run several programs at one time: a word processor, a Web browse1 and an e-mail package. And 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.

A thread is part of a process. It represents the actual flow of the computation being done. Each process must have at least one thread. Multithreading is also possible, where several threads execute within the context of the same process, by running different instructions from the same application.

2.1 Process concept

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow Multiple programs to be loaded into memory and executed concurrently.

This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process/ which is a program in execution. A process is the unit of work in a modern time-sharing system.

A system consists of a collection of processes. An operating system processes executing system code and user processes executing user code. Potentially/ all these processes can execute concurrently/ with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

2.2 Process:

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

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, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

2.2.1 Process control block:

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. It contains many pieces of information associated with a specific process, including these:

- Process state: The state may be new, ready, running, waiting, halted, and so on.
- Program counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs.
- CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information: This information may include such information as the value of the base and limit

registers, the page tables, or the segment tables, depending on the memory system used by the operating system

- Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information :This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The contents of PCB vary from process to process.

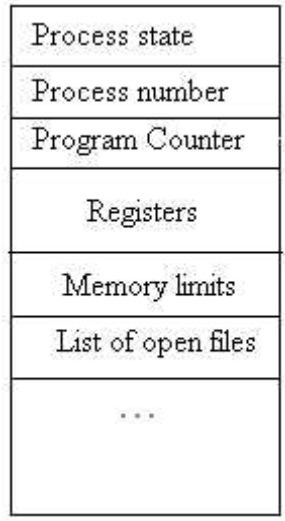


Fig:2.1 Process control block

2.2.2 Process State:

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in following figure.

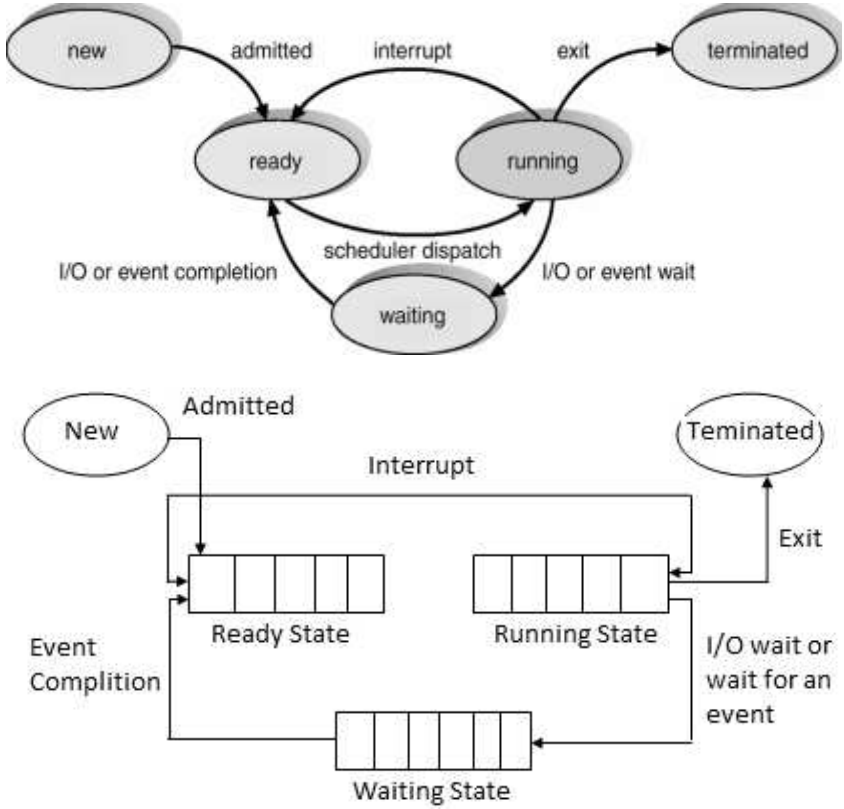


Fig: 2.2 Process State diagram

New →Ready: The operating system creates a process and prepare the process to be executed, then the operating system moved the process into Ready state.

Ready→ Running: The operating system selects one of the job from the ready queue and move the process from ready state to running state.

Running→ Terminated : When the execution of a process has completed, then the operating system terminates that process from running state. Sometimes operating system terminates the process due to some other reasons like memory unavailable, access violation, protection error, I/O failure, data misuse and so on.

Running→Ready: When the time slot of the processor expired, or if the processor received any interrupt signal, then the operating system shifts running process to ready state.

Running→ Waiting: A process is put in to the waiting state, if the process needs an event occurs, or an I/O device require. The operating system does not provide the I/o or event immediately then the process moved to waiting state by the operating system.

Waiting→Ready: A process in the blocked state is moved to the ready state when the event for which it has been waiting occurs. For example a process is in running state need an I/O device. Then the process moved to the blocked or waiting state, when the I/O device provided by the operating system the process moved to ready state from waiting or blocked state.

2.3 Operations on Processes:

2.3.1 Create a new Process:

The main operation on processes is to create a new one. The "process create" call can be used to create a new process. When a user initiates to execute a program, the operating system creates a process to represent the execution of the program. The creation of executable program includes many steps.

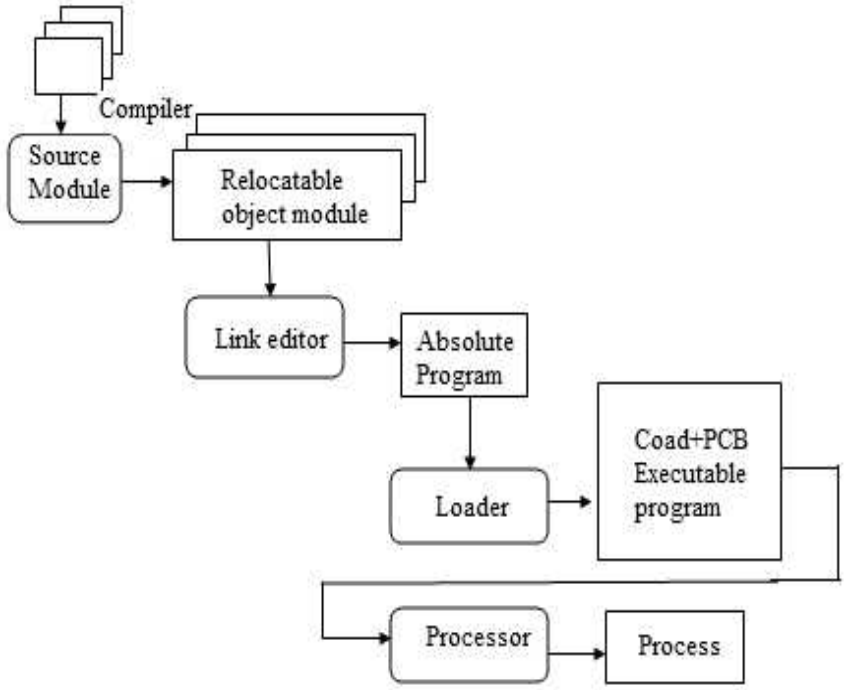


Fig 2.3: Steps to create a process

The source modules or source code (a program written in programming languages) is translated in to object programs or object modules with the help of translator (compiler). The re-locatable object modules converted to absolute programs by linker. The absolute programs are converted into executable programs by loaders. Then the processor executes this program. This executing program is called process.

The process consists of the machine code image of the program in memory and PCB structure.

2.3.2 Terminate an Existing Process:

The dual of creating a process is terminating it. A process can terminate itself by returning from its main function, or by calling the exit system call. Generally the process terminates when execution finished. Some other causes are:

- Time slot expired

- Memory violation
- I/O failure
- Invalid instruction

Time slot expired: When the process execution does not completed within the time quantum, then the process terminated from running state. The CPU picks next job in ready queue to execute.

Memory violation: If the process need more memory than available memory, then the process terminated from running state.

I/O Failure: A process need an I/O operation at the time of execution, but the I/O device is not available at that time. Then the process is moved in to waiting state. The operating system does not provide the I/O device, even the process resides in the waiting state, then the process terminated.

Invalid instruction: If the process having the invalid instructions and the CPU failed to execute those instructions, then the Process terminated.

2.3.3 Suspend execution:

A Process may suspend itself by going to sleep. This means that it tells the system that it has nothing to do now, and therefore should not run. A sleep is associated with a time: when this future time arrives, the system will wake the process up.

2.3.4 Send a signal or message:

A common operation among processes is the sending of signals. A signal is often described as a software interrupt: the receiving process receives the signal rather than continuing with what it was doing before. In many cases, the signal terminates the process unless the process takes some action to prevent this.

2.4 Concurrent process

Concurrent processing is a computing model in which multiple processors executes instructions simultaneously for better performance. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead

of sequentially as they would have to be carried out by a single processor. Concurrent processing is sometimes said to be synonymous with parallel processing.

executed in parallel. Each of the concurrently executing sequential programs is called a process. Process execution, although concurrent, is usually not independent. Processes may affect each other's behavior through shared data, shared resources, communication, and synchronization.

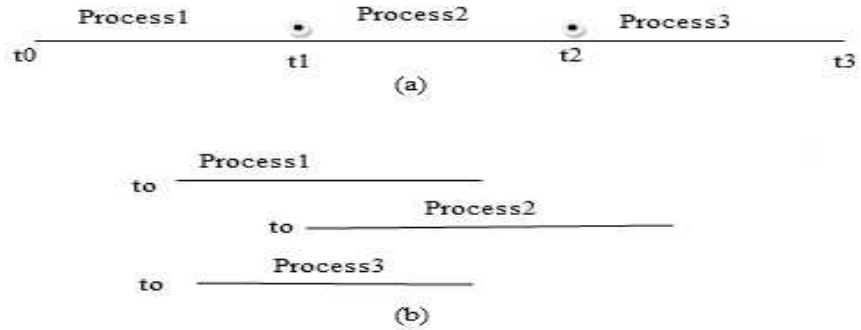


Fig 2.4: Concurrent process

In figure (a) process1 starts its execution at time t0, process2 starts its execution only after process1 finished its execution, and process3 starts its execution only when process2 finished its execution. These process are said to be serial processes.

In figure (b), the execution time of process1, process2, process3 are overlapped, so these are said to be concurrent processing.

2.5 Threads

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a light-weight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the

same process and share resources such as memory, while different processes do not share these resources.

A process is divided into number of light weight processes. Each light weight process is called thread. The thread has a program counter that keeps the track of which instruction to execute next, it has registers, which holds its current working variables. It has a stack, which contains the execution history.

Number of threads can share memory space, open files and other resources. Same as number of processes can share physical memory, disk, printers and so on. Thread has some of the properties of process and it operates in many respects in the same manner as process. Threads can be in one of several states: Ready, Blocked, running or terminated. Threads share the CPU, and only one thread can be active at a time. Threads are not independent of one another. All threads can access every address in the task. A thread can read or write over any other threads stacks.

The thread of program which may be executed concurrently with other thread, this capability is called multi threading.

Example of thread:

Programmer wish to type the text in word processor, then the programmer open a file in word processor, and typing the text, it is one thread, then the text is automatically formatting, it is another thread. The text is automatically specifies the spelling mistakes, is again another thread. And finally the file is automatically saved in the disk, which is again a thread. Thus typing, formatting, spell checking, saving file all these are threads.

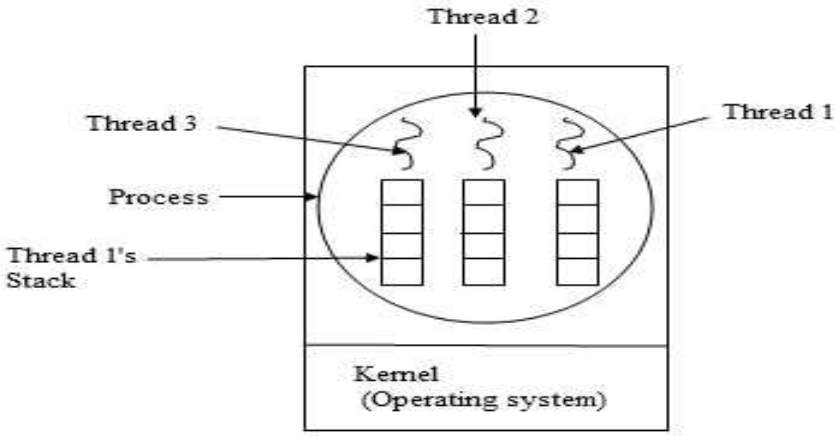


Fig 2.5: Threads

2.6 Multithreading:

A process is divided into number of smaller tasks, each task is called thread. Number of threads within a process execute at a time is called Multithreading. Based on the functionality threads are divided into four categories:

1) One to one
2) One to Many
3) Many to one
4) Many to many

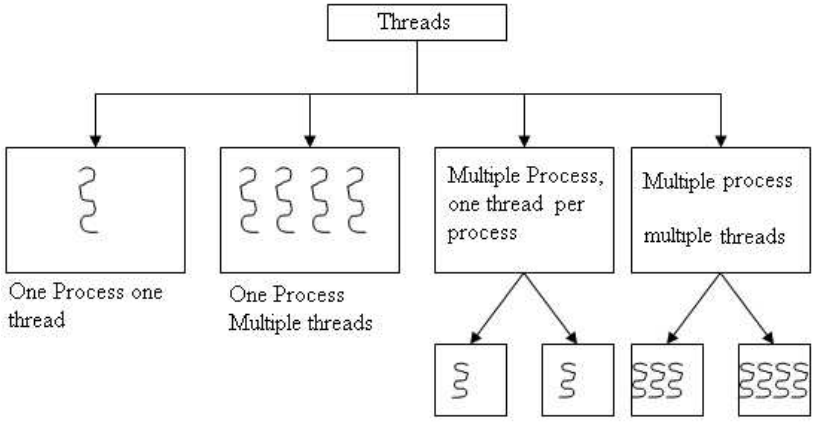


Fig 2.6: Types of multithreading

One to one: (One process one thread)
In this approach, the process maintains only one thread, so it is called as single threaded approach. MS-DOS operating system supports this type of approach.

One to Many :(One process, multiple threads)
In this approach a process is divided into number of threads. The best example of this is JAVA run time environment.

Many to one: (Multiple processes, one thread per process)
An operating system support multiple user processes but only support one thread per process. Best example is UNIX.

Many to many: (Multiple processes, multiple threads per process)
In this approach each process is divided into number of threads. Examples are Windows 2000, Solaris LINUX.
There are two main ways to implement threads:

- In user space
- In Kernel space

User level threads:
This type of threads loaded entirely in user space, the kernel knows nothing about them. When threads are managed in user space, each process has its own private thread table. The thread table contains information like program counter, registers, state etc. When the thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table.

User level threads allow each process to have its own scheduling algorithm. The entire process loaded in user space, so the process does not switch to the kernel mode to do thread management. User level threads can run on any operating system.

When user level thread executes a system call, all the threads with in the process are blocked. Only a single thread with in a process can execute at a time, so multi processing is not implemented.

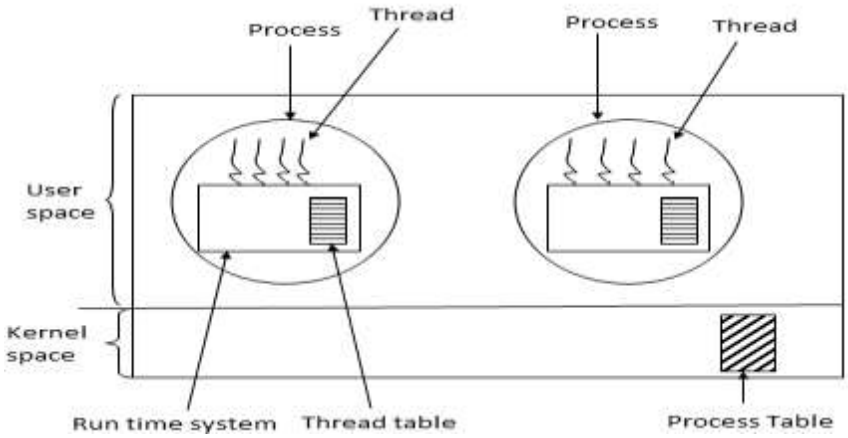


Fig 2.7: User Level Threads

Kernel level threads:

In kernel level threads the kernel does total work of head movement. There is no thread table in each process. The kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy any existing thread it makes call to the kernel, kernel then takes the action.

The kernel thread table holds each thread register, state and other information. The information is the same as with user level threads, but it is now in the kernel instead of the user space.

The kernel can simultaneously schedule a multiple threads from the same process on multiple processors. If a thread in a process is blocked, then the kernel can schedule another thread of the same process.

It requires more cost for creating and destroying threads in kernel. The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

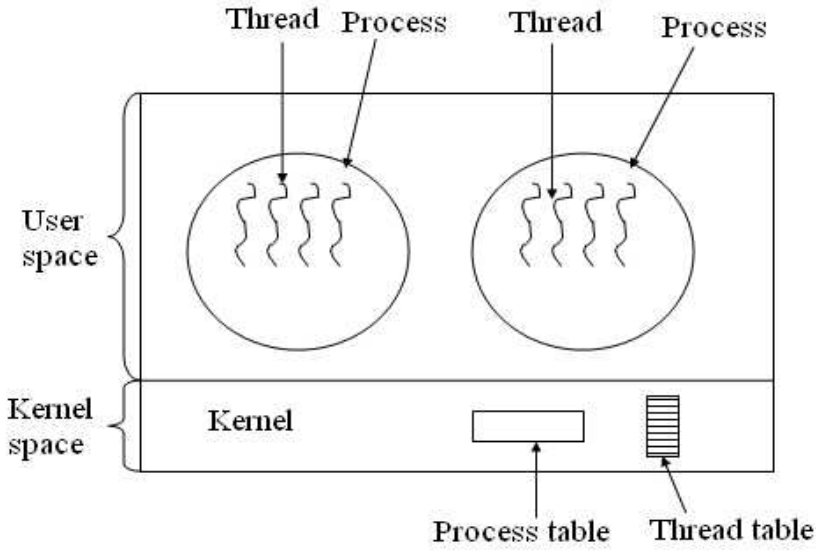


Fig 2.8: Kernel level threads

2.7 CPU scheduling

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. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

2.7.1 Scheduling queues:

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.

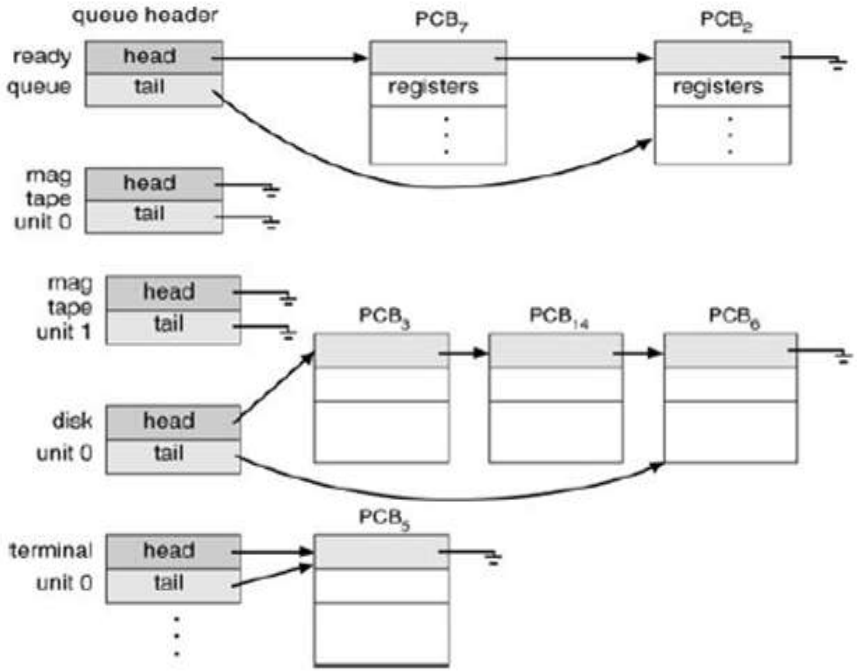


Fig: 2.9 The ready queue & various types of device queues

2.7.2 Schedulers:

There are three main schedulers

- Long term scheduler
- Short term scheduler
- Medium term scheduler

Long term Scheduler:
The function of long term scheduler is, selects processes from job pool and loads them into main memory (ready queue) for execution, so the long term scheduler is called as job scheduler.

For example, assume that a computer lab consisting of 20 dummy nodes connected to the server using local area network, out of 20, ten users want to execute their jobs, then these ten jobs are loaded into spool disk. The long term scheduler selects the jobs from the spool disk and loaded them into main memory in ready queue.

Short term Scheduler:
The **short-term scheduler**, or **CPU scheduler**, selects a job from ready queue (processes that are ready to execute) and allocates the CPU to that process with the help of dispatcher. The method of selecting a process from the ready queue (by short term scheduler)is depending on CPU scheduling algorithm.

Dispatcher:
Dispatcher is a module, which connects the CPU to the process selected by the short term scheduler. The main function of dispatcher is switching the CPU from one process to another process. Another function of dispatcher is jumping to the proper location in the user program, and ready to start execution. The dispatcher should be fast, because it is invoked during each and every process switch. The time taken by dispatcher to stop one process and start another process is known as ‘dispatch latency’. The degree of multiprogramming is depending on the dispatch latency. If the dispatch latency is increasing, then the degree of multiprogramming is decreases.

Medium term Scheduler
If the process request an I/O in the middle of the execution, then the process is removed from the main memory and loaded in to waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations are performed by medium term scheduler.

The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). 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. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.

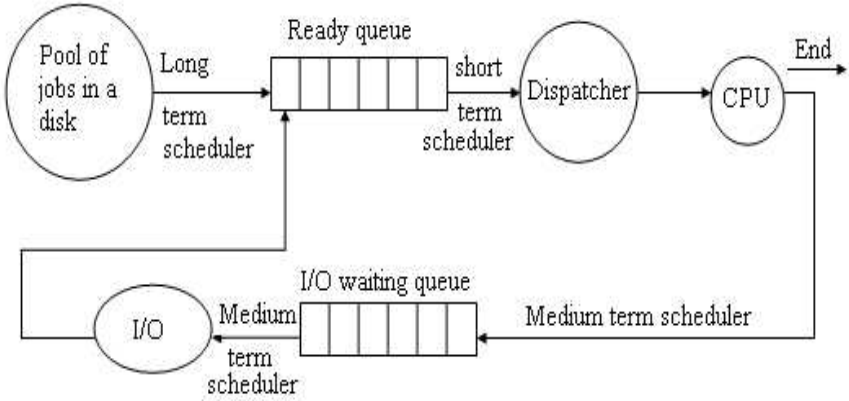


Fig 2.10: Queuing diagram of the CPU Scheduling

2.7.3 Context Switch:

Switching the CPU to another process requires saving the state of the old process and loading the saved state of the new process. This task is known as a **context switch**. The context of a process is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory management information. When a context switch occurs, the operating system saves the context of the old process in its PCB and loads the saved

context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are less than 10 milliseconds.

2.7.4 CPU & I/O burst cycle:

The success of CPU scheduling depends on an observed property of processes:

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

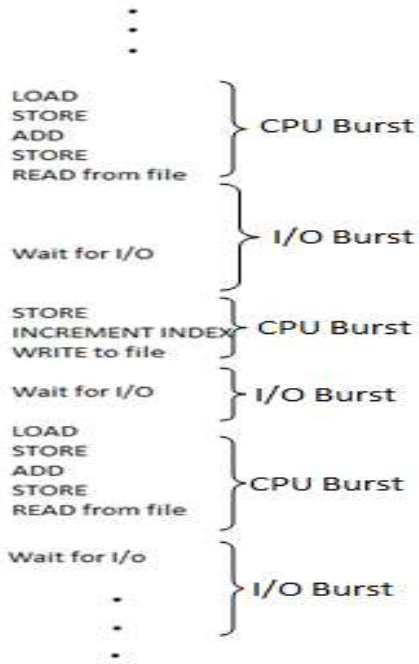


Fig:2.11 Execution is an alternating sequence of CPU and I/O bursts

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

Nearly all processes alternate bursts of computing with (disk) I/O requests.

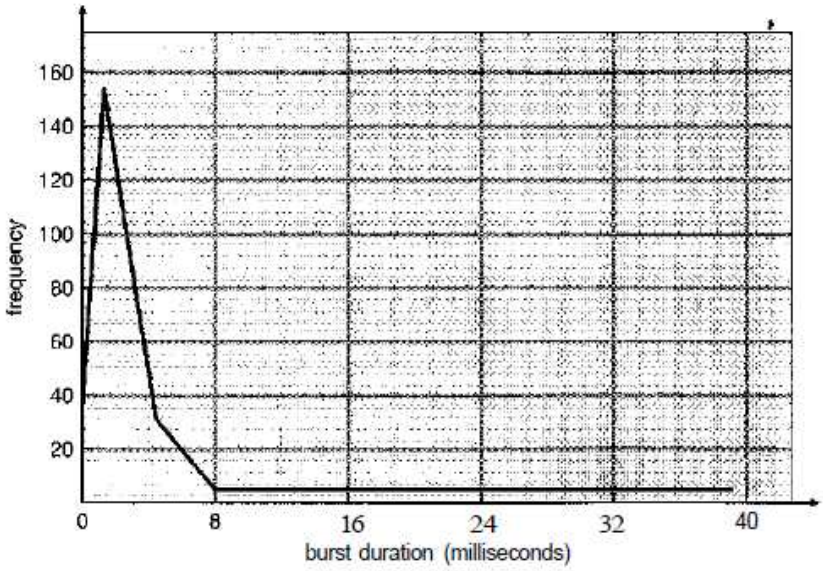


Fig: 2.12 Histogram of CPU burst time

2.8 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. These criteria are used for comparison between the algorithms to choose the best one. The criteria include the following:

- **CPU utilization.** We want to keep the CPU as busy as possible. The CPU utilization may 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.** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput*. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time.** From the point of view of a particular process, the Important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion of that process is the *turnaround time*. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time.** The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. *Waiting time* is the sum of the periods spent waiting in the ready queue.
- **Response time.** Response time is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize Turnaround time, waiting time, and response time.

2.9 Scheduling Algorithms:

CPU scheduling deals with the problem of deciding which of the process in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

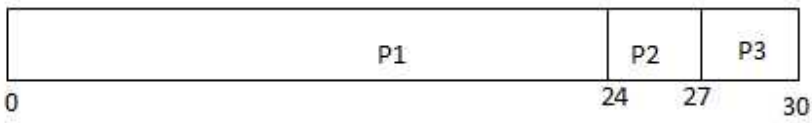
2.9.1 First-Come, First-Served Scheduling

The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO

queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	**Burst Time**
P1	24
P2	3
P3	3

If the processes arrive in the order *P1, P2, P3,* and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process *P2* , and 27 milliseconds for process *P3* .

Thus, the average waiting time is calculated as follows

Average waiting time

Waiting time= starting time – Arrival time

Waiting time for P1= 0 - 0 =0

Waiting time for P2= 24 - 0 =24

Waiting time for P3= 27 - 0 =27

The average waiting time = (0+24+27)/3 =17 milliseconds.

Average turnaround time

Turnaround time= Finish time – Arrival time

Turnaround time for P1= 24 - 0 =24

Turnaround time for P2= 27 - 0 =27

Turnaround time for P3= 30 - 0 =30

The average turnaround time=(24+27+30)/3=27

Average response time

Response time= First response – Arrival time

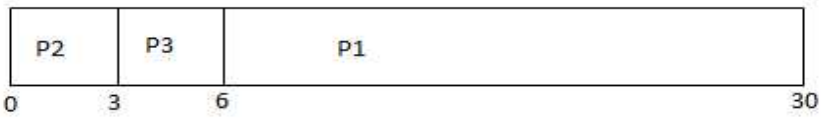
Response time for P1= 0 - 0 =0

Response time for P2= 24 - 0 =24

Response time for P3= 27 - 0 =27

The average Response time = (0+24+27)/3 =17 milliseconds

If the processes arrive in the order P2, *P3* , P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

Consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process

will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and I/O device utilization than might be possible if the shorter processes were allowed to go first.

The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

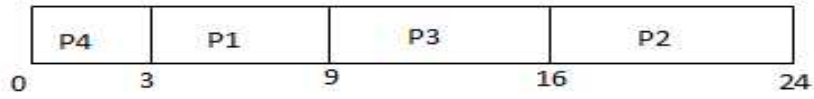
2.9.2 Shortest-Job-First Scheduling:

In this approach the CPU is allocated to the process having the shortest CPU burst time. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

As an example of SJF scheduling, consider the following set of processes arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Following Gantt chart illustrate the SJF scheduling Process:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.

Average waiting time

Waiting time= starting time – Arrival time
Waiting time for P1= 3 - 0 =3
Waiting time for P2= 16 - 0 =16
Waiting time for P3= 9 - 0 =9
Waiting time for P4= 0 - 0 =0
The average waiting time = (3+16+9+0)/4=7 milliseconds.

Thus, the average waiting time is (3 + 16 + 9 + 0) *I* 4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJ F scheduling algorithm is provably optimal, in that it gives the Minimum average waiting time for a given set of processes. Moving a short Process before a long one decreases, the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. There is no way to know the length of the next CPU burst.

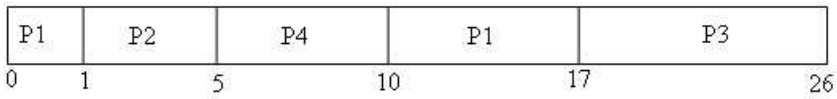
2.9.3 Preemptive SJF scheduling algorithm (shortest-remaining-time-first):

The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.
As an example, consider the following four processes, with the length of The CPU burst given in milliseconds:

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If the processes arrive at the ready queue at the time s shown and need the Indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart.



ProcessP1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milli-seconds) is Larger than the time required by process P2 (4milliseconds), so process P1 is preempted, and process P2 is scheduled.

Average waiting time
Waiting time= starting time – Arrival time
Waiting time for P1= 0 - 0 =0
Waiting time for P2= 1 - 1 =0
Waiting time for P3= 17 - 2 =15
Waiting time for P4= 5 - 3 =2
The average waiting time = (0+0+15+2)/4 =4.25 milliseconds.

Non preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

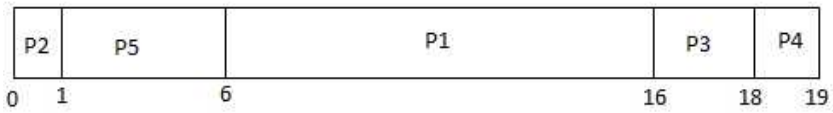
2.9.4 Priority Scheduling:

The SJF algorithm is a special case of the general **priority scheduling algorithm.** A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority. As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2,P3,P4, *P5*, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



Average waiting time

Waiting time= starting time – Arrival time

Waiting time for P1= 6 - 0 =6

Waiting time for P2= 0 - 0 =0
Waiting time for P3= 16 - 0 =16
Waiting time for P4= 18 - 0 =18
Waiting time for P5= 1 - 0 =1
The average waiting time = (6+0+16+18+1)/5 =8.2 milliseconds.
The average waiting time is 8.2 milliseconds

Priority scheduling can be either preemptive or non preemptive. When a process arrives at the ready queue, its priority is compared with the priority if the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

2.9.5 Round-Robin Scheduling:

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready

queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

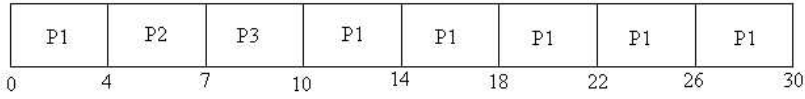
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process it self will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	**Burst time**
P1	24
P2	3
P3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, Process P2. Since process P1 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The Gantt chart is shown in fig.



Average waiting time

Waiting time= starting time – Arrival time

Waiting time for P1=(0-0)+(10-4)+(14-14)+(18-18)+(22-22)+ (26-26)+(30-30) =6

Waiting time for P2= 4 - 0 =4

Waiting time for P3= 7 - 0 =7

The average waiting time = (6+4+7)/3 =5.66 Milliseconds.

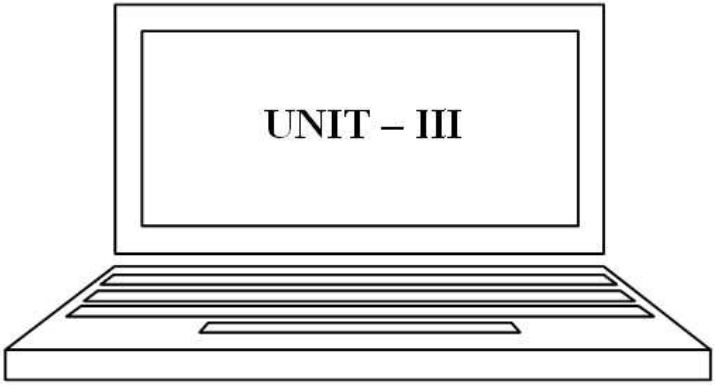
The average waiting time is 17/3=5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

Summary:

First-come-first-serve scheduling is the simplest scheduling algorithm, but it can cause short jobs to wait for very long time. Shortest-Job-first scheduling is provably optimal, providing the shortest average waiting time. The shortest job first is difficult to implement because it is difficult to predict the length of next CPU burst. Shortest-Job-first is special case of general priority scheduling algorithm, which simply allocates the CPU to the highest priority process. Both priority and Shortest-Job-first scheduling may suffer from starvation. Aging is technique to prevent starvation.

Round-robin scheduling is more appropriate for a time-shared system. Round-robin is a preemptive algorithm. FCFS is non-preemptive. Shortest-Job-first and priority algorithms may be either preemptive or non-preemptive. Round-robin allocates the CPU to the first process in the ready queue for q time units, where q is the time quantum. After the q time units, the CPU is preempted and the process is put at the tail of the ready queue. The

major problem is the selection of the time quantum . if the quantum is too large Round-robin degenerates to FCFS scheduling, is the quantum is too small, scheduling overhead in the form of context switch time becomes excessive.



Dead Lock & Memory Management

Deadlock:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

3.1 Resource-Allocation Graph:

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph. This graph consists of a set of vertices *V* and a set of edges *E.* The set of vertices *V* is partitioned into two different types of nodes: P= {P1, P2, ---, Pn} the set consisting of all the active processes in the system, and *R = {R1, R2, ----,Rn },* the set consisting of all resource types in the system.

A directed edge from process *Pi* to resource type *Rj* is denoted by Pi→Rj;

It signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj* → Pi; it signifies that an instance of resource type *Rj* has been allocated to process Pi. A directed edge Pi→*Rj* is called a **request edge;** a directed edge *Rj* → Pi is called an **assignment edge.**

We represent each process Pi as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. The resource-allocation graph shown in Figure depicts the following situation.

The sets P, *R,* and £:

- P={ P1, P2, P3}
- R={R1, R2, R3, R4}
- £ = {P1→R1, P2→R3,R1→P2, R2→P2, R2→P1, R3→P3 }

Resource instances:

- One instance of resource type R1.
- Two instances of resource type R2.
- One instance of resource type *R3.*
- Three instances of resource type *R4.*

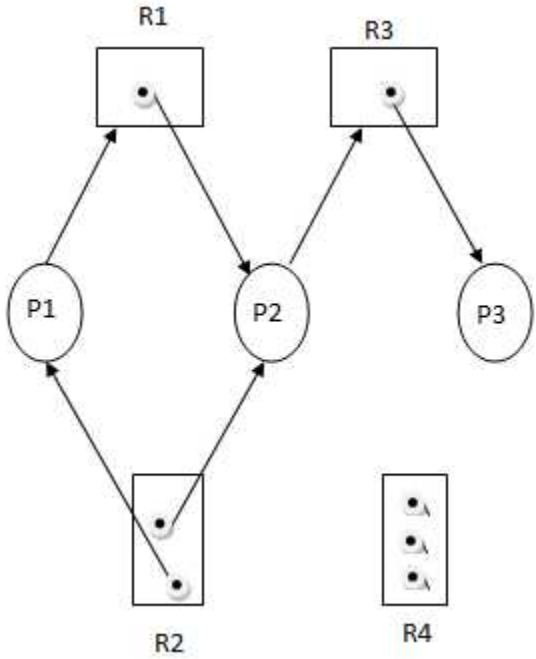


Fig. 3.1: Resource allocation graph

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of *R3.*
- Process P3 is holding an instance of R3.

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Suppose that process P3 requests an instance of resource type *R2*. Since no resource instance is currently available, a request edge P3 →*R2* is added to the graph. At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P1

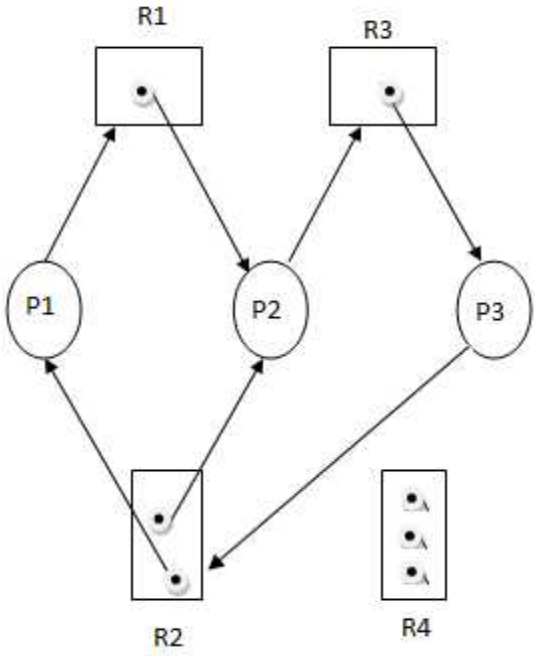


Fig. 3.2 Resource-allocation graph with a deadlock.

Processes *P1, P2,* and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process *P1* or process *P2* to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

3.2 Conditions for deadlock:

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

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource an waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set {*P*0, *P*1, …, *Pn*} of waiting processes must exist 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, …, *Pn*−1 is waiting for a resource that is held by *Pn*, and *Pn* is waiting for a resource that is held by *P*0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

3.3 Deadlock Prevention:

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

3.3.1 Mutual Exclusion:

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general,

however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

3.3.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end. The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3.3.3 No Preemption:

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following

protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of available resources for which the other process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

3.3.4 Circular Wait:

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to give some particular ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let $R = \{R1, R2, ..., Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which, allows us to compare two resources and to determine whether one precedes another in our ordering. we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers.

For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type *Ri*. After that, the process can request instances of resource type *Rj* if and only if *F*(*Rj*) > *F*(*Ri*). If several instances of the same resource type are needed, a *single* request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that, whenever a process requests an instance of resource type *Rj*, it has released any resources *Ri* such that $F(Ri) \geq F(Rj)$. If these two protocols are used, then the circular-wait condition cannot hold.

3.4 Deadlock Avoidance:

The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the **deadlock-avoidance** approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections we explore two deadlock avoidance algorithms.

Safe state: A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

A system is in a safe state only if there exists a **safe sequence**. A sequence of processes <*P*1, *P*2, …,*Pn*> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that *Pi* can be satisfied by the currently available resources plus the resources held by all *Pj*, with $j < i$.

In this situation, if the resources that process *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi*+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources and so a deadlock occurs: The behavior of the processes controls unsafe states.

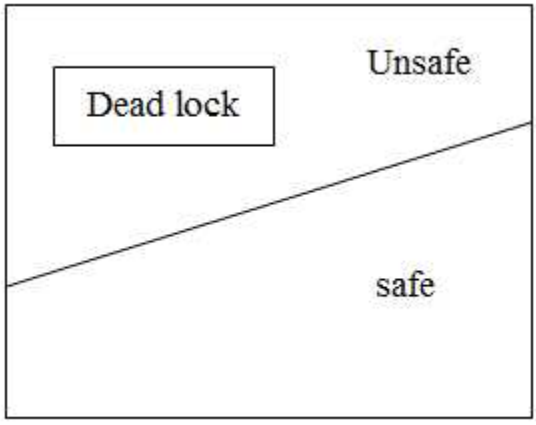


Fig 3.3: Safe, Unsafe and dead lock state space

3.4.1 Banker's algorithm:

It is the dead lock avoidance algorithm, the name was chosen because bank never allocates more than the available cash.

- **Available:** A vector of length *m* indicates the number of available resources of each type. If *Available*[*j*] = *k*, there are *k* instances of resource type *Rj* available.

- **Max:** An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j] = k$, then process Pi may request at most k instances of resource type Rj.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j] = k$, then process Pi is currently allocated k instances of resource type Rj.
- **Need:** An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j] = k$, then process Pi may need k more instances of resource type Rj to complete its task.
- Note that $Need[i][j] = Max[i][j] - Allocation[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \ldots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as *Allocationi* and *Needi*, respectively. The vector *Allocationi* specifies the resources currently allocated to process *Pi*; the vector *Needi* specifies the additional resources that process *Pi* may still request to complete its task.

3.4.2 Safety Algorithm:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize
Work = *Available* and
$Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.

2. Find an i such that both
a. $Finish[i] == false$
b. $Needi \leq Work$
If no such i exists, go to step 4.

3. *Work = Work + Allocationi*
Finish[*i*] = *true*
Go to step 2.

4.If *Finish*[*i*] == *true* for all *i*, then the system is in a safe state.
This algorithm may require an order of $m \times n^2$ operations to decide whether a state is safe.

3.4.3 Resource-Request Algorithm

Let *Requesti* be the request vector for process *Pi*. If *Requesti*[*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If *Requesti* ≤ *Needi*, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If *Requesti* ≤ *Available*, go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

Available = Available − Requesti;
Allocationi = Allocationi + Requesti;
Needi = Needi − Requesti;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for *Requesti*, and the old resource-allocation state is restored.

3.5 Deadlock Detection:

When a deadlock situation occurs, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Detection mechanism of deadlocks for single instance of resource type and multiple instances of resource type is different.

We can detect the deadlocks using wait for graph for single instance resource type and detect using detection algorithm for multiple instances of resource type.

3.5.1 Single Instance of Each Resource Type:

Single instance of resource type means, the system consisting of only one resource of one type. This type of dead lock can be detected with the help of wait for graph. Wait for a graph is a graph which is derived from 'Resource allocation graph'. It consisting of only processes as vertices. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

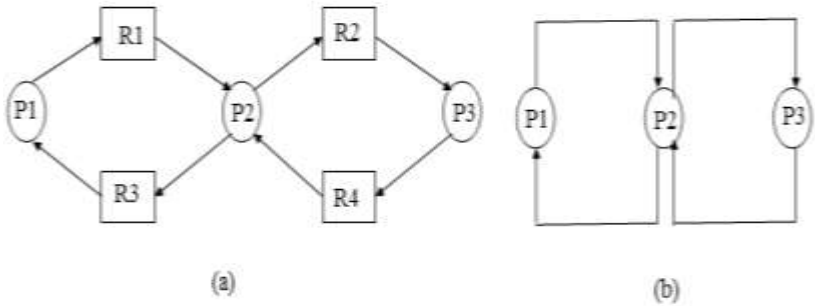


Fig 3.4: (a) Resource-allocation graph.
(b) Corresponding wait-for graph.

An edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs. An edge $Pi \rightarrow Pj$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $Pi \rightarrow Rq$ and $Rq \rightarrow Pj$ for some resource *Rq.* a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and 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 $n2$ operations, where n is the number of vertices in the graph.

3.5.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. For multiple instances of each resource a deadlock-detection algorithm is applicable. Following are several time varying data structures

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

To simplify notation, we again treat the rows in the matrices *Allocation* and *Request* as vectors; we refer to them as *Allocationi* and *Requesti*, respectively

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
Work = Available.
For *i* = 0, 1, …, *n*−*1*,
If *Allocationi* ≠ 0, then *Finish*[*i*] = *false*;
otherwise, *Finish*[*i*] = *true*.

2. Find an index *i* such that both
a. *Finish*[*i*] = *false*
b. *Requesti* ≤ *Work*
If no such *i* exists, go to step 4.

3. *Work = Work + Allocationi*
Finish[*i*] = *true*
Go to step 2.

4. If *Finish*[*i*] == *false*, for some *i*, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if *Finish*[*i*] == *false*, then process *Pi* is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes *P*0 through *P*4 and three resource types *A, B, C*. Resource type *A* has 7 instances, resource type *B* has 2 instances, and resource type *C* has 6 instances. Suppose that, at time *T*0, we have the following resource-allocation state:

Process	**Allocation**			**Request**			**Available**		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in *Finish*[*i*] ==*true* for all *i*.

Suppose now that process *P*2 makes one additional request for an instance of type *C*. The *Request* matrix is modified as follows:

Process	**Request**		
	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

3.6 Recovery from deadlock:

There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

3.6.1 Process Termination

There are two methods for terminating process. We use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense.
- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

3.6.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

There are three methods to eliminate deadlocks using resource preemption.

1. Selecting a victim: Select resources and processes are to be preempted. Cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has consumed during its execution.

2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation:** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a *starvation* situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Memory Management

In uni-programming system, the main memory is divided into two parts, one part is for operating system and another part is for currently executing job. Consider the following figure.

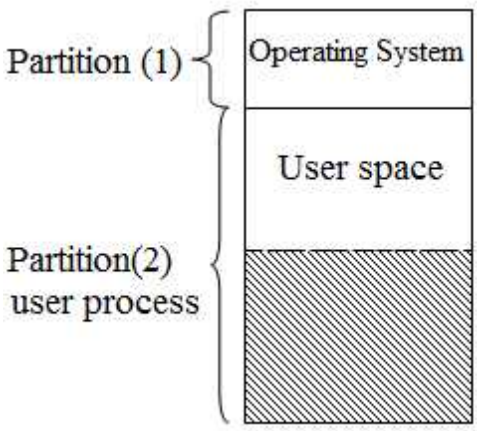


Fig 3.5: Main memory partition

Partition 1 is used operating system and partition 2 is used to store the user process. In partition 2 some part of memory is wasted, it is indicated by blacked lines in figure.

In multiprogramming environment the user space is divided in to number of partitions. Each partition is for one process. The task of sub division is carried out dynamically by the operating system; this task is known as "Memory Management". The efficient memory management is possible with multiprogramming.

3.7 Logical- versus Physical-Address Space

An address generated by the CPU is commonly referred to as a logical address, where as an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address. We use *logical address* and *virtual address* interchangeably in this text. The set of all logical addresses generated by a program is a **logical-address space**; the set of all physical addresses corresponding to these logical addresses is a **physical-address space**. Thus, in the execution-time

address binding scheme, the logical- and physical-address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

The base register is now called a **relocation register**. The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user generates only logical addresses and thinks that the process runs in locations 0 to *max*. The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

The concept of a *logical-address space* that is bound to a separate *physical-address space* is central to proper memory management

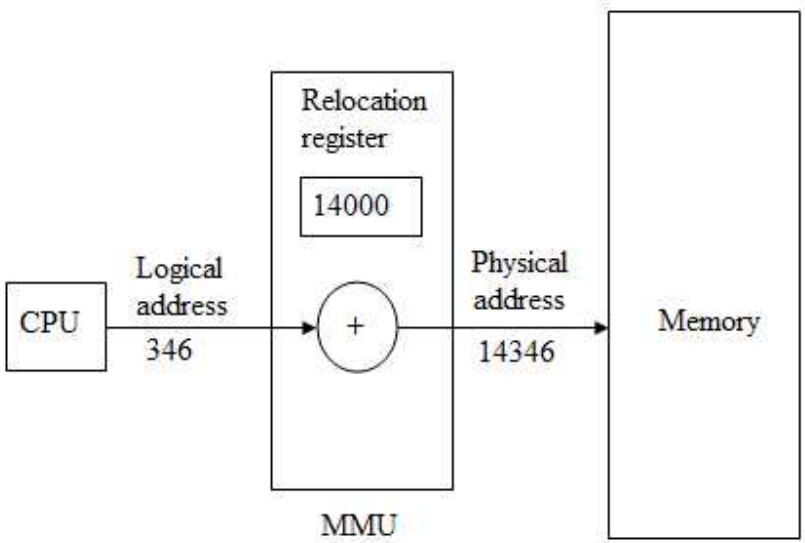


Fig.3.6: Dynamic relocation using a relocation register.

3.8 Swapping

Swapping is used to increase main memory utilization. For example main memory consisting of 15 processes, assume that it is the maximum capacity of memory to hold the processes. The CPU currently executing the process no:14 in the middle of the execution the process 14 needs I/O. then the CPU switches to the another job and process 14 is moved to a disk and the another process is loaded in to the main memory in place of process 14. When the process 14 is completed its I/O operation then the process 14 is moved to the main memory from disk. Switching process from main memory to disk is known as "swap out" and switching from disk to main memory is called "swap in". This type of mechanism is said to be "Swapping". We can achieve the efficient memory utilization with swapping.

Swapping requires 'Backing store'. The backing store is commonly a fast disk. It must be large enough to accommodate the copies of all process images for all users. When a process is swapped out, its executable image is copied into backing store. When it is swapped in, it is copied into the main memory at new block allocated by the memory manager.

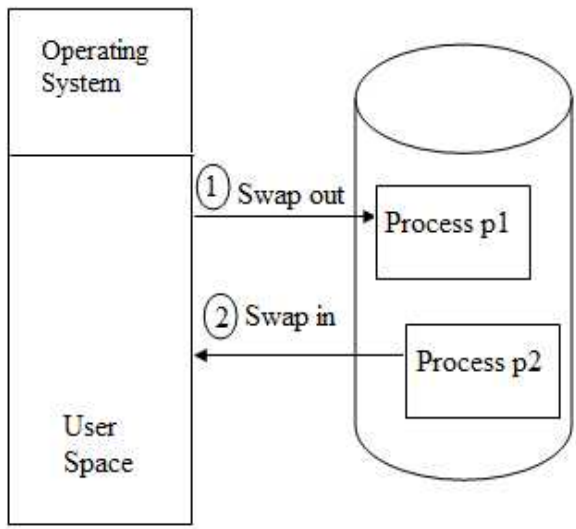


Fig.3.7: Swapping

3.9 Memory Protection

Protecting the operating system from user processes and protecting user processes from one another. We can provide this protection by using a relocation register, with a limit register. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory .

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

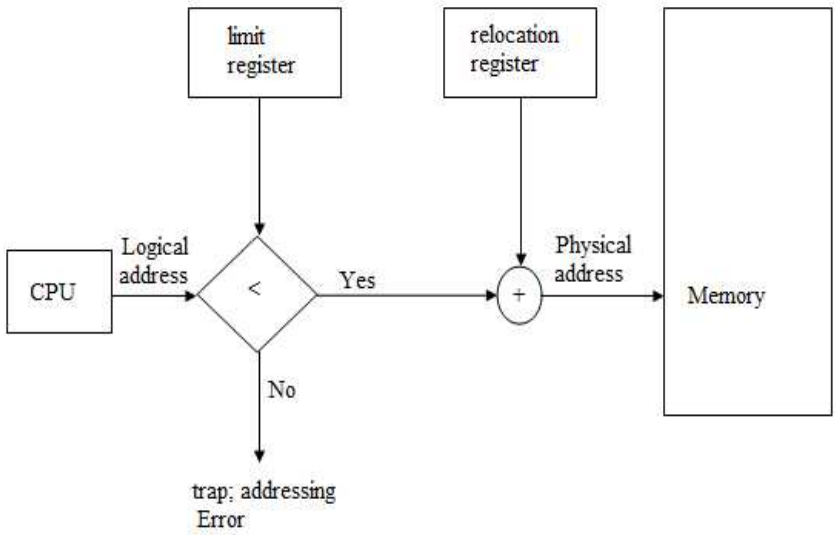


Fig. 3.8: Hardware support for relocation and limit registers.

3.10. Memory allocation methods:

The main memory contains both the operating system and the users processes. Following are the various memory allocation methods.

3.10.1 Single Partition allocation

In this method, the operating system resides in the lower part of the memory, and the remaining memory is treated as a single partition. This single partition is available for the user's space. Only a single job can be loaded in this user space at time.

The short term scheduler selects a job from ready queue for execution, and the dispatcher loads that job in main memory. The main memory consists of only one process at time, because the user space treated as a single partition.

The main disadvantage of this scheme is the memory is not utilized fully. A lot of memory is wasted.

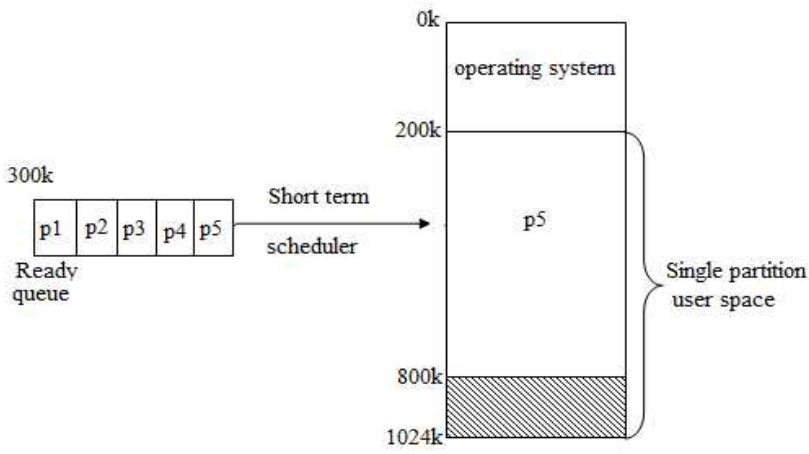


Fig. 3.9: Memory allocation for single partition

3.10.2 Multiple partitioning:

This method can be implemented in three ways

a. Fixed equal multiple partitioning

b. Fixed variable multiple partitioning

c. Dynamic multiple partitioning

3.10.2.1 Fixed equal multiple partitioning:

In this scheme the operating system resides in the low memory and rest of main memory is used as user space. The user space is divided in to fixed partitions. The size of these partitions is depending up on the operating system. For example the total main memory size is 6MB; 1MB is occupied by the operating system. The remaining 5MB is partitioned in to 5 equal fixed partitions of 1MB each. P1, P2, P3, P4, P5 are the 5 jobs to be loaded in the main memory, there size is given in the table below:

Job	size
P1	450kb
P2	1000kb
P3	1024kb
P4	1500kb
P5	500kb

Internal and external fragmentation:

Process P1 is loaded into partition1. The maximum size of partition1 is 1024kb, the size of P1 is 450kb. So 1024-450=574kb space is wasted, this wasted memory is said to be 'Internal fragmentation'. But there is no enough space to load process P4, because the size of process P4 is greater than all the partitions, so the entire partition (partition5) is wasted. This wasted memory is said to be 'External fragmentation'.

Therefore the total internal fragmentation is

=(1024-450)+(1024-1000)+(1024-500)

=574+24+524

=1122kb

The external fragmentation is 1024 kb.

A part of memory wasted within a partition is called internal fragmentation and the wastage of an entire partition is called external fragmentation.

Advantages:
It supports multiprogramming. Effective utilization of the processor and I/O devices is possible.

Disadvantage:
It suffers from internal and external fragmentation.

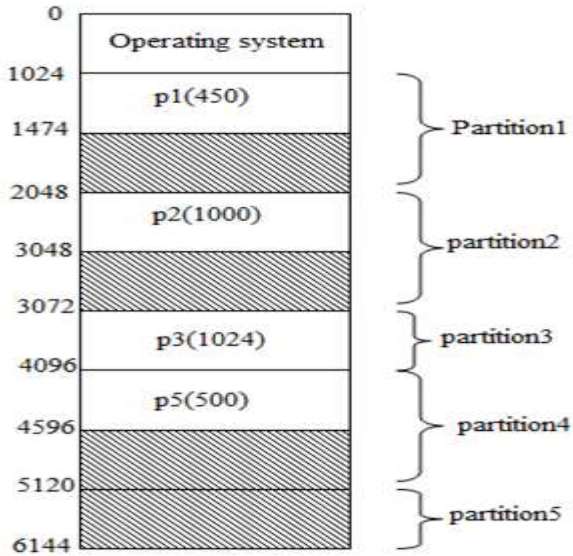


Fig.3.10: Main memory allocation

3.10.2.2 Fixed variable multiple partitioning:

In this method the user's space of main memory is divided into number of partitions, but partitions size are different in length. The operating system keep a table indicating which partition of memory are available and which are occupied. When a process arrives and needs memory, we search for a partition large enough for this process. If we find the space large enough to fit the process, allocate the partition to that process.

There are three strategies used to allocate memory in this scheme

First-Fit: It allocates the first partition that is big enough. Searching can start either from low memory or high memory. We

can stop searching as soon as we find a free partition that is large enough.

Best-Fit: It allocates the smallest partition that is big enough. Searching can be started from either end of memory, Searches entire memory, and allocates the smallest partition that is big enough for the process.

Worst-Fit: Search the entire memory and selects the partition which is largest of all.

For example, assume that we have 4000kb of main memory available, and operating system occupies 500kb. The remaining 3500kb of memory is used for user's processes.

Job Queue		
Job	Size	Arrival time (in ms)
J1	825KB	10
J2	600KB	5
J3	1200KB	20
J4	450KB	30
J5	650KB	15

Partitions	
Partition	Size
P1	700KB
P2	400KB
P3	525KB
P4	900KB
P5	350KB
P6	625KB

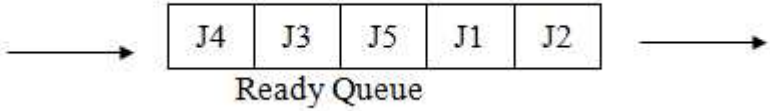


Fig.3.11: Scheduling example

Here we use first-fit strategy to illustrate the problem. Out of 5 jobs J2 arrives first, the size of J2 is 600KB,. Searching can be started from low memory to high memory and first partition which is big enough is allocated. Here P1 is first partition which is big enough for J2, so load the J2 in P1.

J1 is next job in the ready queue. The size is 825KB; P4 is the first partition that is big enough, so load the J1 in to P4.

J5 arrives next, the size is 650KB, and there is no large enough partition to load that job, so J5 has to wait until enough partition is available.

J3 arrives next, the size is 1200KB. There is no large enough space to load this one also. J4 arrives last, the size is 450KB, and partition P3 is large enough to load this process. So load J4 in to P3.

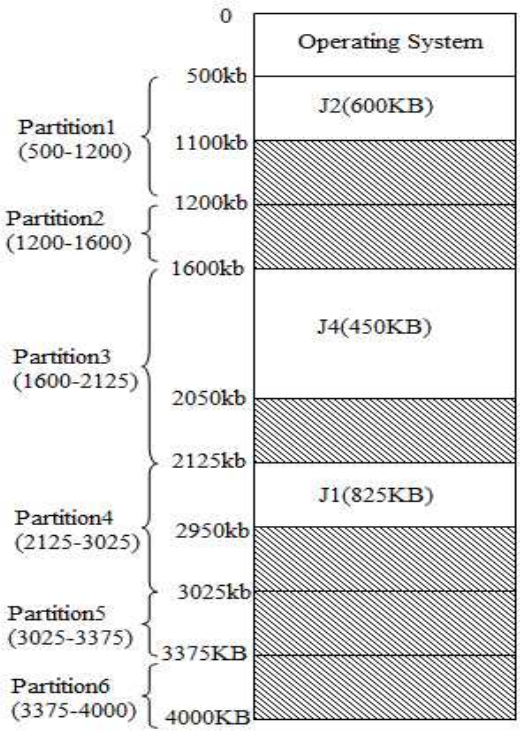


Fig.3.12: Memory allocation

Partitions P2,P5,P6 are totally free, there is no processes in these partitions. This wasted memory is said to be external fragmentation. The total external fragmentation is 1375, (400+350+625). The total internal fragmentation is (700-600) + (525-450) + (900-825) =250.

3.10.2.3 Dynamic multiple partitioning

In this method partitions are created dynamically, so that each process is loaded in to partition of exactly the same size at that process. Here the entire user space is treated as a single partition that is 'big hole'. The boundaries of partitions are dynamically changed. These boundaries are depending on the size of processes. Consider following example.

Job Queue		
Job	Size	Arrival Time
J1	825kb	10
J2	600kb	5
J3	1200kb	20
J4	450kb	30
J5	650kb	15

The job queue is

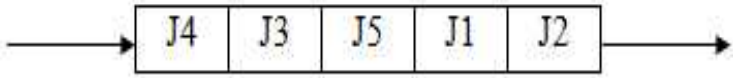
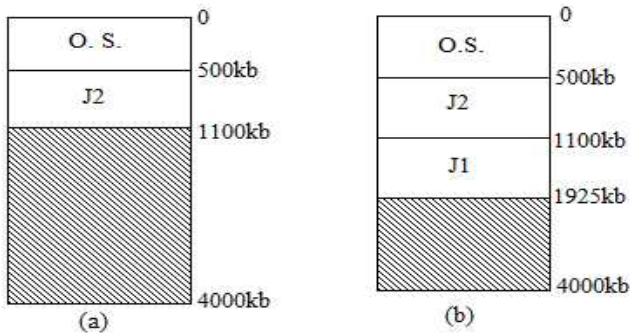


Fig. 3.13: Memory allocation

Job J2 arrives first, so load the J2 into memory first. Next J1 arrives. Load the J1 in to memory next. Then load J5,J3 and J4 into the memory. Consider following figure 3.14 for better understanding.



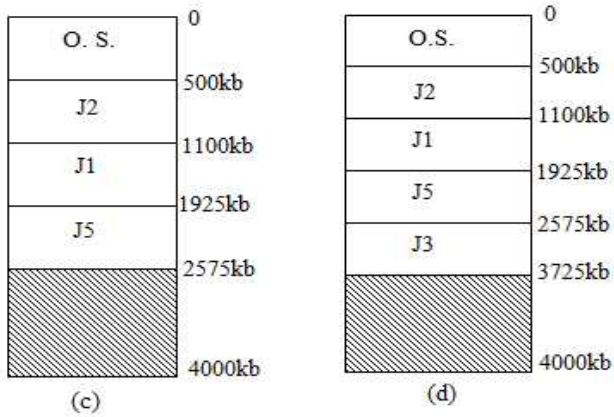


Fig.3.14: Dynamic Memory allocation

In figure (a), (b), (c), (d) jobs J2, J1, J5, and J3 are loaded. The last job is J4, the size of J4 is 450kb, but the available memory is 225kb, which is not enough to load J4, so the job J4 has to wait until the memory is free. Assume that after some time J5 has finished and it releases its memory. Then the available memory becomes 225+650=875Kb. This memory is enough to load J4. Consider the following figure 3.14 (e) and (f).

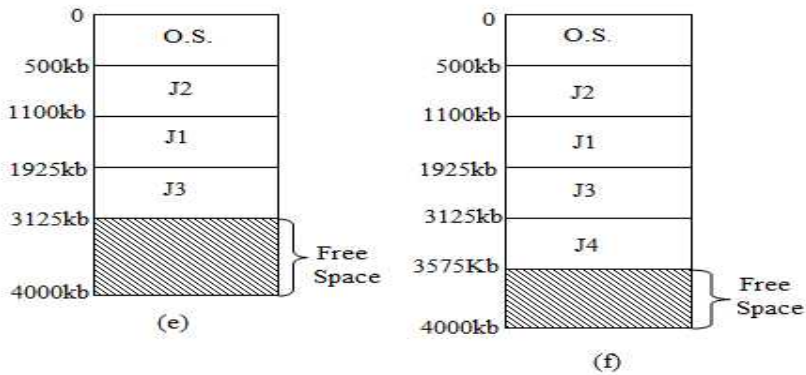


Fig. 3.14: Dynamic memory allocation

In this method partitions are changed dynamically, so it does not suffer from internal fragmentation. Efficient memory and processor utilization are possible. This scheme suffers from external fragmentation.

3.11 Compaction

Compaction is a method of collecting all the free spaces together in one block, this block can be allotted to some other job.

For example consider the example of **Fixed variable multiple partitioning method.** The total internal fragmentation is (100+75+75=250Kb). The total external fragmentation is (400+350+625=1375Kb). Collect the internal and external fragmentation together in one block (250+1375=1625Kb). This type of mechanism is said to be compaction. Now the compacted memory is 1625Kb.

Now the scheduler can load job J3 (1200Kb) in compacted memory. Thus efficient memory utilization can be possible using compaction.

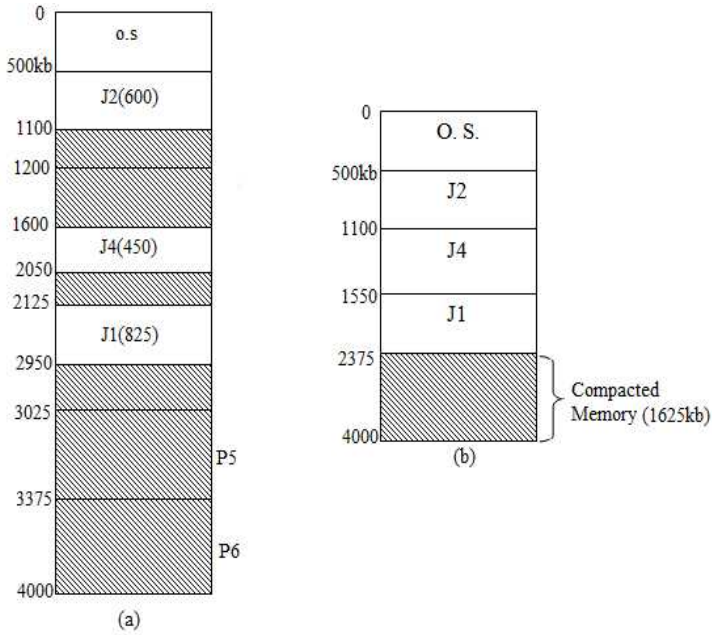


Fig. 3.15: Compaction

3.12 Paging:

The single and multiple partitioning methods supports continues memory allocation, the entire process loaded in partition. In paging the process is divided in number of small parts, these are loaded in to elsewhere in the main memory.

The physical memory is divided in to fixed sized blocks called frames; the logical address space (user Process) is divided in to fixed sized blocks called pages. The page size and the frame size must be equal. The size of page or frame is depending on the operating system. Generally the page size is 4KB.

In this method operating system maintain a data structure, called page table, it is used for mapping purpose. The page table specifies some useful information, it tells which frames are allocated and which frames are available, and how many total frames are there and so on. The general page table consisting of two fields, one is page number and other one if frame number. Each operating system has its own method for storing page table.

Every address generated by the CPU is divided into two parts; one is ‘page number’ and second is ‘page offset’ or displacement. The page number is used index in page table. Consider the following figure, the logical address space that is CPU generated address space is divided into pages, each page having the page number(P) and displacement (D) . the pages are loaded in to available free frames in the physical memory. Page table contain the base address of each page in physical memory, that address is combined with offset to find the physical address of the page.

The mapping between the page number and frame number is done by page map table. The page map table specifies which page is loaded in which frame, displacement is common.

For better understanding consider the following example.

There are two jobs in the ready queue, the size of job1 is 16kb and job 2 is 24kb. The page size is 4kb. The available main memory is 72kb i. e. 18 frames. So job 1 is divided in to 4 pages and job 2 is divided in to 6 pages. Each process maintains a program table. Consider the following figure for better understanding.

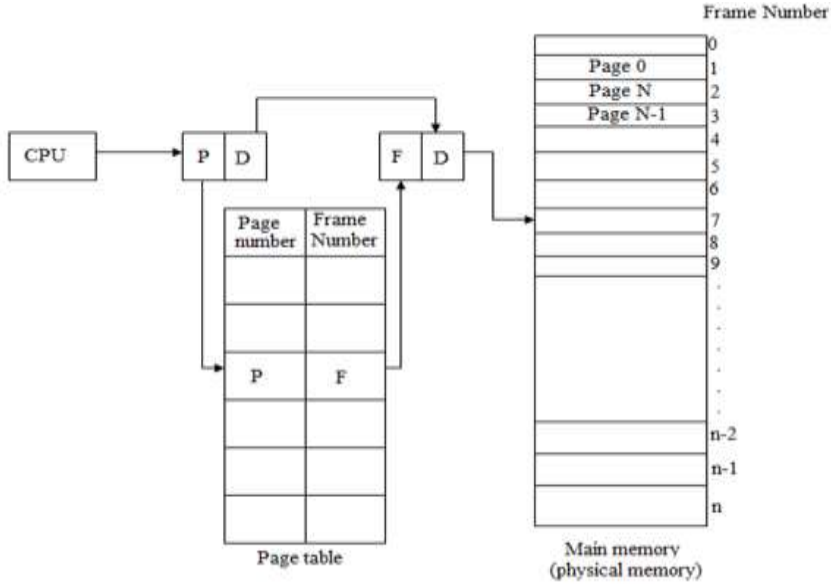


Fig.3.16: Structure of paging scheme

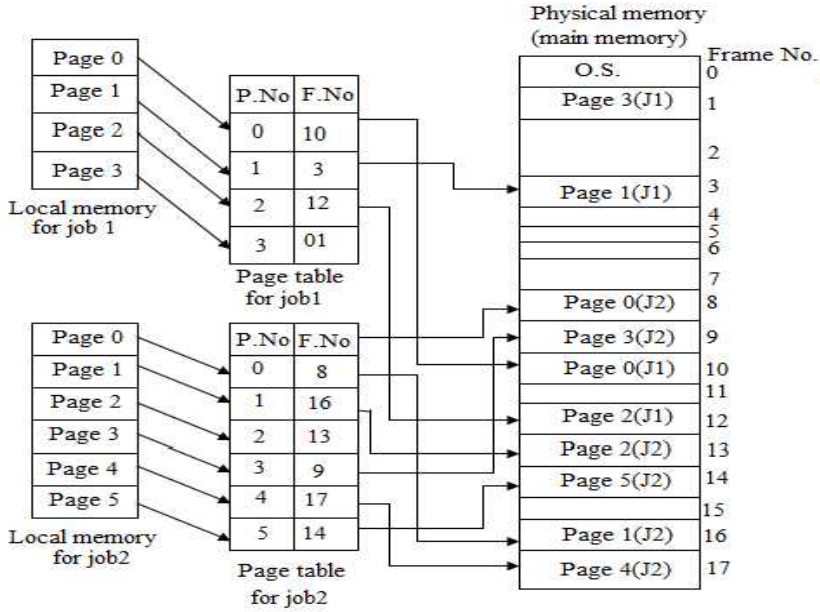


Fig.3.17: Example of paging

Four pages of job 1 are loaded in different locations in main memory. The O.S. provides a page table for each process. The page table specifies the location in main memory. The capacity of main memory in this example is 18 frames, and available jobs requires only 10 frames, so remaining 8 frames are free. These free frames can be used for some other jobs.

Advantages: Paging supports the time sharing system. It does not effect from fragmentation. It supports virtual memory.

Disadvantages: Paging may suffer 'page breaks'. For example consider a job with the logical address space 17kb, the page size is 4kb. So this job requires 5 frames. The last frame i.e. fifth frame requires only 1kb of memory, so the remaining 3kb is wasted. It is said to be page breaks.
If the number of pages is large, then it is difficult to maintain the page table.

3.12.1 Shared Pages:

In multiprogramming environment, it is possible to share the common code by number of processes at a time, instead of maintaining the number of copies of same code. The logical address is divided into pages, these pages can be shared by number of processes at a time. The pages which are shared by number of processes are called shared pages.

For example, consider a multiprogramming environment with 10 users. Out of 10 users, 3 users wish to execute a text editor; they want to take their bio-data in text editor. Assume that text editor requires 150kb and user bio-data occupies 50kb of data space. So they would need 3*(150+50)=600kb. But in shared paging the text editor is shared by all the users' jobs, so it requires 150kb and user files requires 50*3=150kb. Therefore (150+150) =300kb enough to manage these three jobs instead of 600kb. Thus shared paging saves 300kb of space.

The frame size and frame size is 50kb. So 3 frames are required for text editor and 3 frames are required for user files. Each process (P1, P2, P3) has a page table, the page table shows

the frame numbers, the first 3 frame numbers in each page table i.e. 3,8,0 are common. It means the three processes shared the three pages. The main advantage of shared pages is efficient memory utilization is possible.

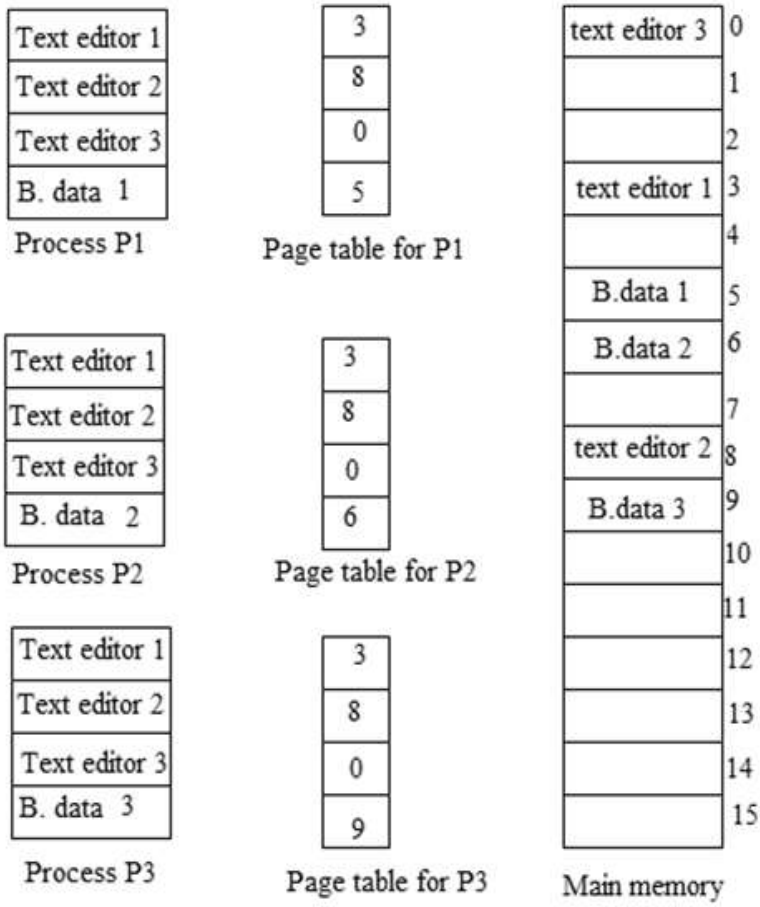


Fig.3.18: Shared Paging

3.13 Segmentation

In segmentation the instructions are logically grouped, such as a subroutines, arrays or other data areas. Every program is a collection of these segments. Segmentation is the technique for managing these segments. For example consider following figure.

Each segment has a name and a length. The address of the segment specifies both segment name and the offset within the segment. For example the length of the segment 'Main' is 100kb. 'Main' is the name of the segment. The operating system searches the entire main memory for free space to load a segment. This mapping is done by segment table. The segment table is a table having two entries segment 'Base' and segment 'Limit'

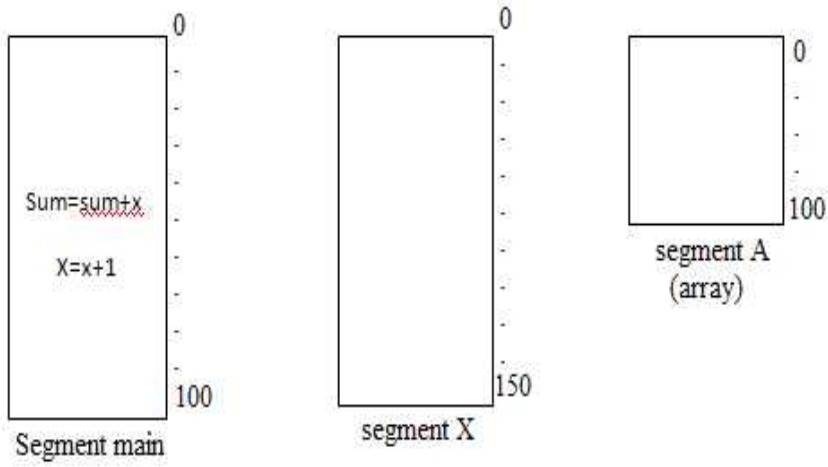


Fig.3.19: Segmented address space

The segment base contains the starting physical address where the segment resides in memory. The segment limit specifies the length of the segment. Following figure shows the basic hardware for segmentation.

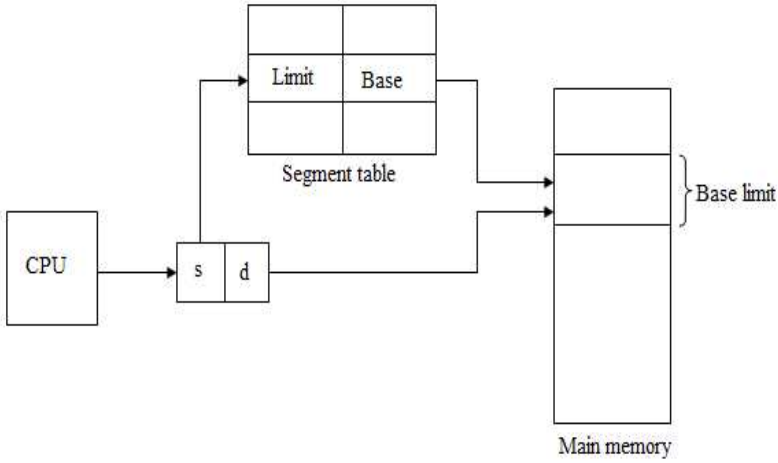


Fig.3.20: Basic segmentation hardware

The logical address consists of two parts: a segment number (s), and offset or displacement in to that segment (d). the segment number (s) is used as an index in to the segment table.

For example consider the following figure in which the logical address space is (a job) is divided in to four segments, numbered from 0 to 3. Each segment has an entry in the segment table. The limit specifies the size of the segment and the base specifies the starting address of the segment. Here segment '0' is loaded in to main memory from 1500kb to 2500kb, so 1500kb is base and 2500-1500=1000kb is the limit.

Segmentation supports virtual memory. It eliminates fragmentation by moving segments around; fragmented memory space can be combined in to a single free area.

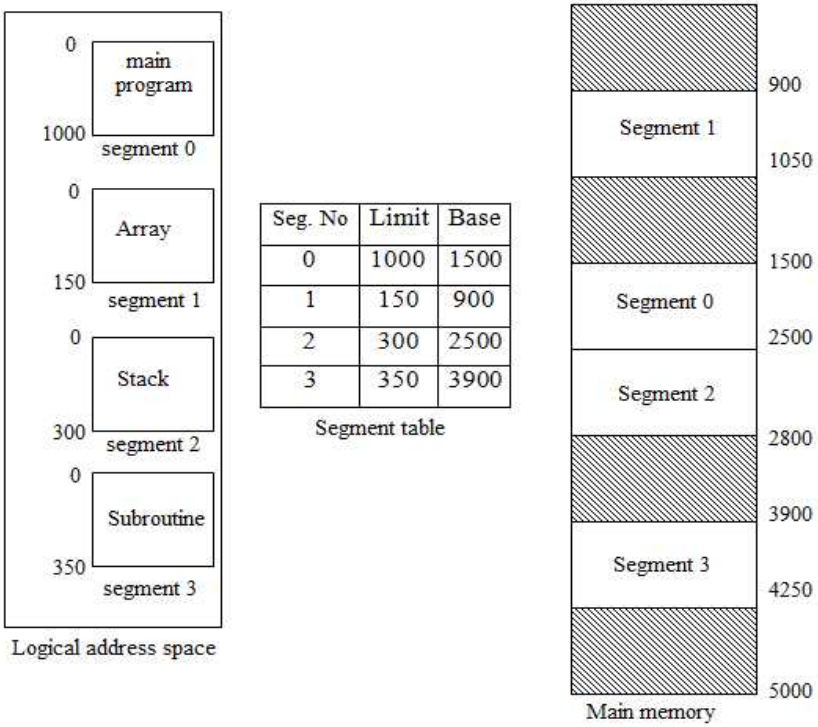


Fig.3.21: Example of segmentation

3.13.1 Segmentation with paging

In this scheme segmentation is combined with paging. The process is divided into segments, then each segment is divided into pages and each segment is maintained by page table. The logical address is divided into three parts <s,p,d>. one is the segment number(s), second is the page number(p), and third is offset or displacement(d). The basic hardware for this scheme is shown in following figure.

For example consider the figure3.23. The logical address space is divided into 3 segments numbered from 0 to 2. Each segment maintains a page table. The mapping between the page and frame is done by page table. In our example frame number 8 shows the address (1,3), 1 is a segment number and 3 is the page number. This scheme is used to avoid the fragmentation.

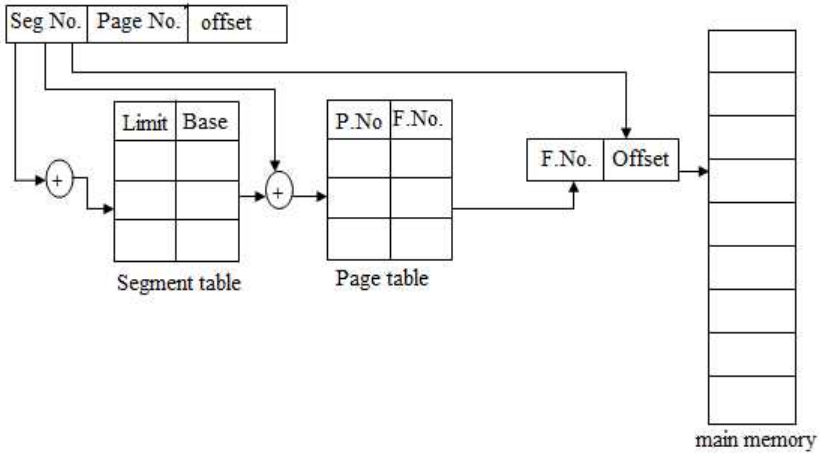


Fig.3.22: Paged segmentation memory management scheme.

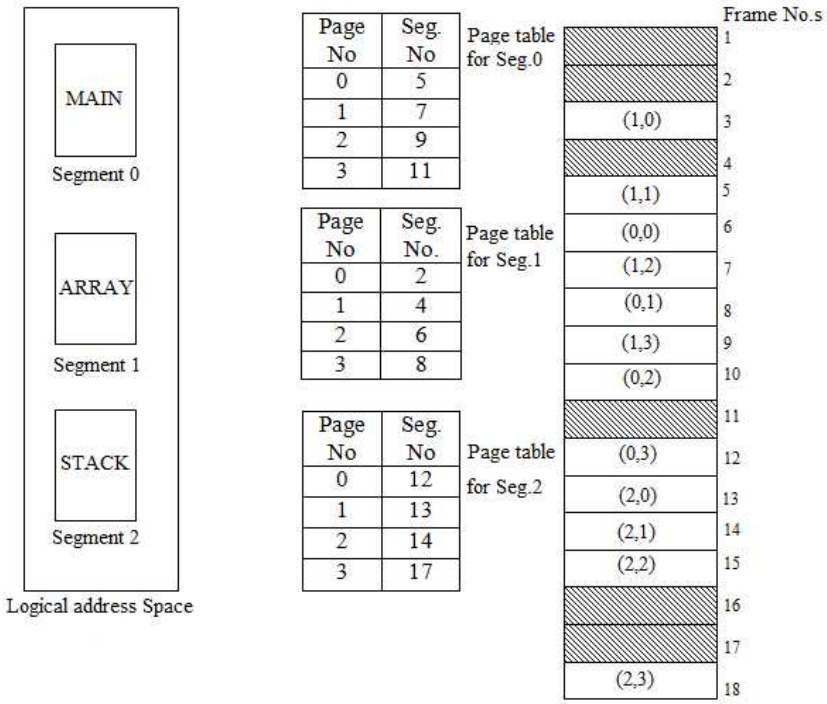


Fig.3.23: Paged Segmentation

3.14 Demand Paging

Demand paging is the application of virtual memory. Virtual memory is a technique which allows the execution of a process, even the logical address space is greater than the physical address space.

In this scheme a page is not loaded into the main memory from secondary memory, until it needed, i.e. a page is loaded into the main memory by demand. Hence this scheme is said to be "Demand paging".

For example assume that the logical address space is 72KB, the page and frame size is 8KB, so the logical address space is divided into 9 pages, numbered from 0 to 8. The available memory is 40KB, i.e. 5 frames are available, so the 5 pages are loaded into the main memory and the remaining 4 pages are loaded in the secondary storage device, when ever those pages are required, the operating system swap-in those pages into main memory.

In demand paging the page map table consist of three fields. One is page number, second is frame number, and third one is valid/invalid bit. If a page is reside in main memory the valid/invalid bit is set to 'valid'. Otherwise, if the page is reside in secondary storage the bit is set to 'invalid'.

In figure 3.24 the page numbers 1,3,4,6 are loaded in secondary memory, so those bits are set to invalid, remaining all pages reside in main memory, so those bits are set to valid.

The available free frames in main memory are 5, so 5 pages are loaded in main memory.

3.15 Page fault

When the processor need to execute a particular page, and that page is not available in main memory, this situation is said to be 'page fault'. When the page fault is happened, the page replacement will be needed. The page replacement means select a victim page in the main memory and replace that page with the required page from the backing store(disk). The victim page is selected by the "page replacement algorithm".

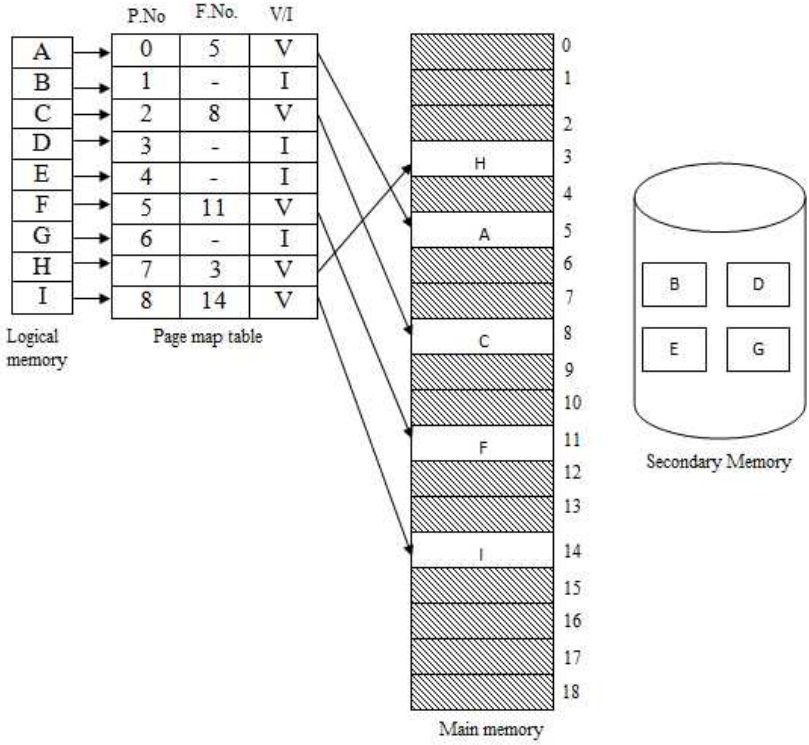


Fig.3.24: Demand Paging

3.16 Page replacement algorithm:

There are various page replacement algorithms, some of the popular page replacement algorithms are given below:

3.16.1 FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen i.e. it replaces the page that has been in memory from long time than all the other pages in the memory.

For example , consider a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0,3,2,1,2,0,1,7,0,1.

Suppose we have three frames which are initially empty. For our example reference string, our three frames are initially empty.

The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.

The first reference to 3 results in replacement of page 0, since it is now first of the three pages in memory. this means the next reference, to 0, will cause a page fault. Page 1 is then replaced by page 0. This process continues as shown in following figure. Every time a page fault occurs, we show which pages are in our three frames. There are total 15 page faults occurs for given reference string with three free frames.

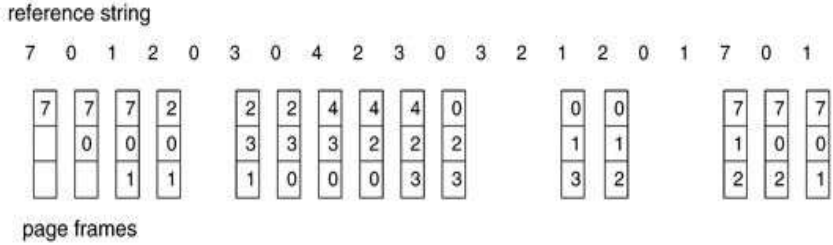


Fig.3.25 FIFO Page replacement

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. It may be possible that we have to select the active page for replacement, after replacing the active page with new one, page fault occurs immediately for the active page. Then some other page will be needed to be replaced to bring the active page in memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

We notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly:** For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving

more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

3.16.2 Optimal page-replacement algorithm:

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply this:

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in following figure. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.

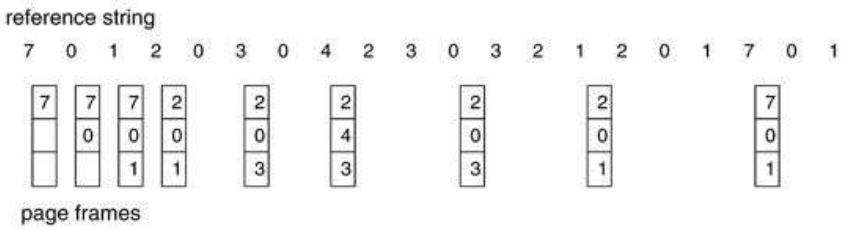


Fig.3.26: Optimum page replacement algorithm

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

3.16.3 Least recently used Page replacement:

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time .This approach is the least-recently-used (LRU) algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward.

The result of applying LRU replacement to our example reference string is shown in following figure. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15 faults.

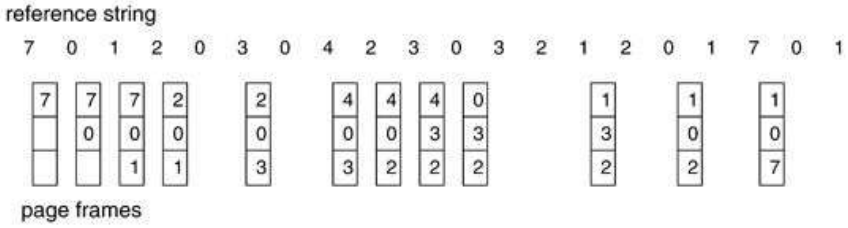
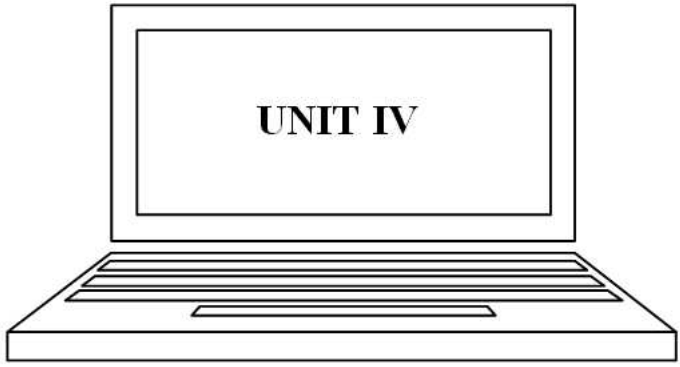


Fig.3.27: LRU replacement algorithm.



File System & Introduction To Linux Operating System

File system:

The file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of *files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system.

4.1 File Concept:

File management is one of the most visible services of an operating system. Computer can store information in several different physical forms; magnetic tape, disk, optical disk are the most common forms. Each of these devices has its own characteristics and physical organization. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file.* Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined **structure**, which depends on its type. A *text* file is a sequence of characters organized into lines (and possibly pages). A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. 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.

4.2 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *'exm.c'.* Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. The information about all files is kept in the directory structure, which also resides on secondary storage.

A file has certain other attributes, which vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non human-readable name for the file.
- **Type:** This information is needed for those systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

4.3 Operations on Files:

A file is an **abstract data type**. The basic operation performed on files are create file, write to a file, read a file, reposition a file,

delete a file. The operating system can provide system calls to perform all these operations.

- **Creating a file:**

Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file, its location in the file system, and possibly other information.

- **Writing a file:**

To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The directory entry will need to store a pointer to the current end of the file. Using this pointer the address of the next block can be computed and the information can be written. The write pointer must be updated.

- **Reading a file:**

To read from a file, we use a system call that specifies the name of the file and memory location where the next block of the file should be put. Again the directory is searched for the associated directory entry. And again the directory will need a pointer to the next block to be read. Once that block is read , the pointer is updated. A given process is usually only reading or writing a given file, and the current operation location is kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing the system complexity.

- **Repositioning within a file:**

The directory is searched for the appropriate entry, and the current-file-position pointer is set to a given value (given position). Repositioning within a file need not involve any actual I/O. This file operation is also known as files seek.

- **Deleting a file:**

To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

4.4 Types of files:

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an *extension*, usually separated by a period character. In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MSDOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the *type* of the file and the type of operations that can be done on that file. Only a file with a *.com*, *.exe*, or *.bat* extension can be *executed*, for instance. The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a batch file containing, in ASCII format, commands to the operating system. MSDOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the WordPerfect word processor expects its file to end with a *.wp* extension. Following table shows some common file types.

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

file type	usual extension	function
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	binary file containing audio or A/V information

4.5 Access Methods:

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

4.5.1 Sequential access method:

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. Reads and writes make up the bulk of the operations on a file. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward *n* records, for some integer *n*—perhaps only for *n* = 1. Sequential access, which is depicted in figure, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

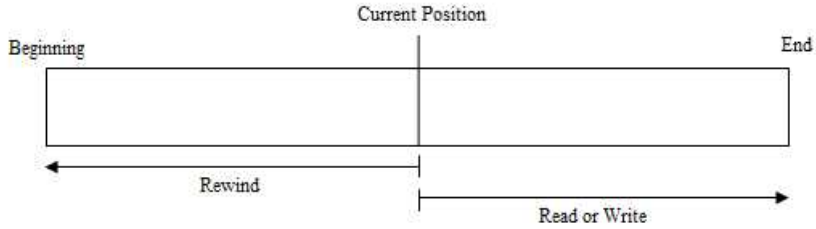


Fig.4.1: Sequential access file

4.5.2 Direct Access:

Another method is **direct access** (or **relative access**). . The direct-access method is based on a disk model of a file, since disks allow random access to any file block. A file is made up of fixed length **logical records** that allow programs to read and write records rapidly in no particular order For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows blocks to be read or written in any order. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

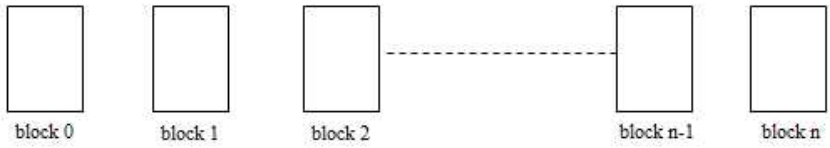


Fig.4.2: Direct access file

4.5.3 Other access methods:

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, 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.

With large files, the index file itself may become too large to kept in memory. One solution is then to create an index for the index file. The primary index file would contain pointers to secondary index files which then points to the actual data items.

For example, IBM's indexed sequential access method uses a small master index which points to disk blocks of a secondary index. The secondary index block point to the actual file blocks. Then to find a particular item, the binary search is first made of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally this block is searched sequentially. In this way, any record can be located from its key by at most two direct access reads.

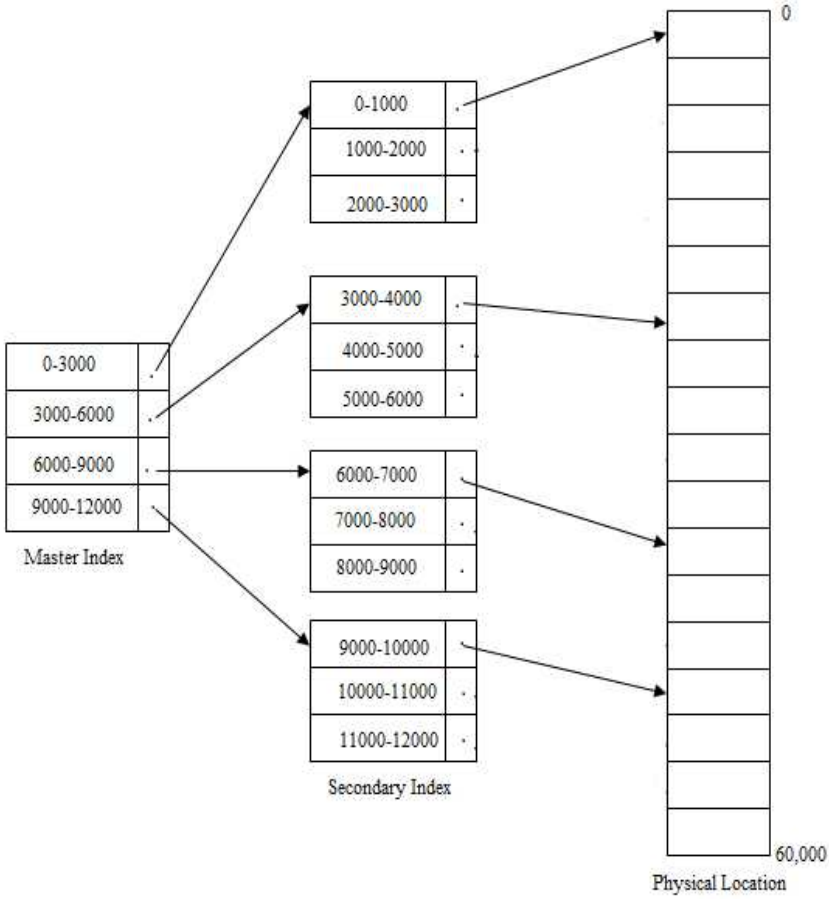


Fig.4.3: Indexed sequential file

4.6 Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we shall discuss.

4.6.1 Bit Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be

0011110011111100011000000111000000 …

The main advantage of this approach is its relative simplicity and efficiency in finding the first free block, or consecutive free blocks on the disk.

4.6.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. In our example , we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on

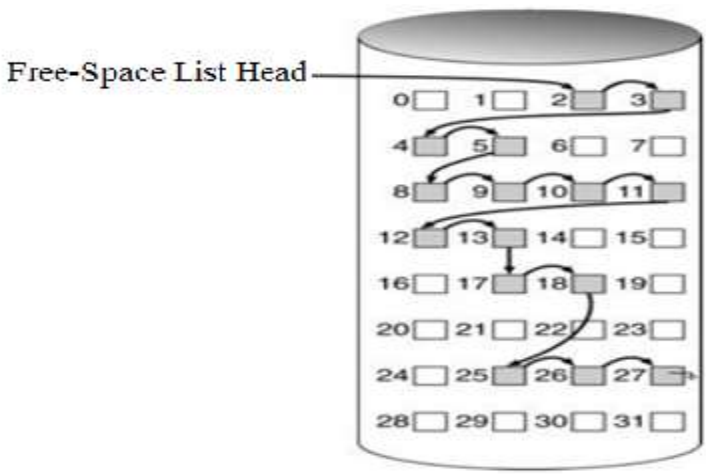


Fig.4.4: Linked free space list on disk

4.6.3 Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n−1 of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

4.6.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

4.7 Allocation methods:

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems support all three. More commonly, a system uses one method for all files within a file-system type.

4.7.1 Contiguous allocation method:

The **contiguous-allocation** method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$after block b normally requires no head movement. When head movement is

needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.

Contiguous allocation of a file is defined by the disk address of the first block and length (in block units). If the file is *n* blocks long and starts at location *b*, then it occupies blocks *b*, *b* + 1, *b* + 2, …, *b* + *n* − 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block *i* of a file that starts at block *b*, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation.

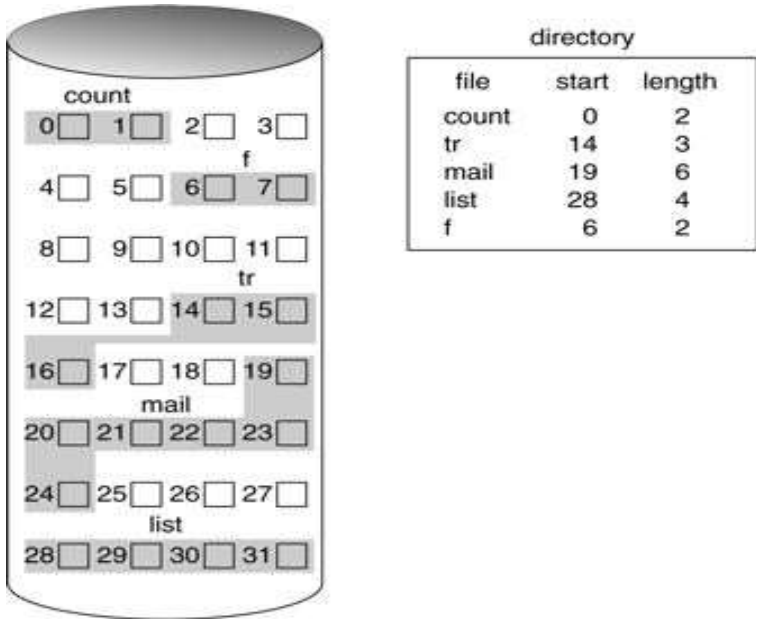


Fig.4.5: Contiguous allocation of disk space

One difficulty with contiguous allocation is finding space for a new file. To solve this problem dynamic allocation strategy is

used. Here for contiguous free blocks we use a word ‘hole’. There are three methods in this strategy.

First fit: It allocates the first hole that is big enough. Searching can be starts from the beginning of the disk, or where the last first fit search ended. We can stop searching as soon as we find a large enough free hole.
Best fit: It allocates the smallest hole, which is big enough. Searching can be starts from the beginning of the disk, it searches the entire disk and allocates the hole that is big enough for the required file.
Worst fit: It allocates the largest hole that is big enough. Searching can be started from the beginning of the disk, it searches the entire disk and allocated the largest hole that is big enough for the given file.

Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

These algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but it is not contiguous; storage is fragmented into a number of holes, no one of which is large enough to store the data.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created?

4.7.2 Linked allocation:

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For

example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25 as shown in figure. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space-management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocks are available.

The major problem with linked allocation is that it can be used effectively only for sequential-access files. To find the *i*th block of a file, we must start at the beginning of that file, and follow the pointers until we get to the *i*th block.

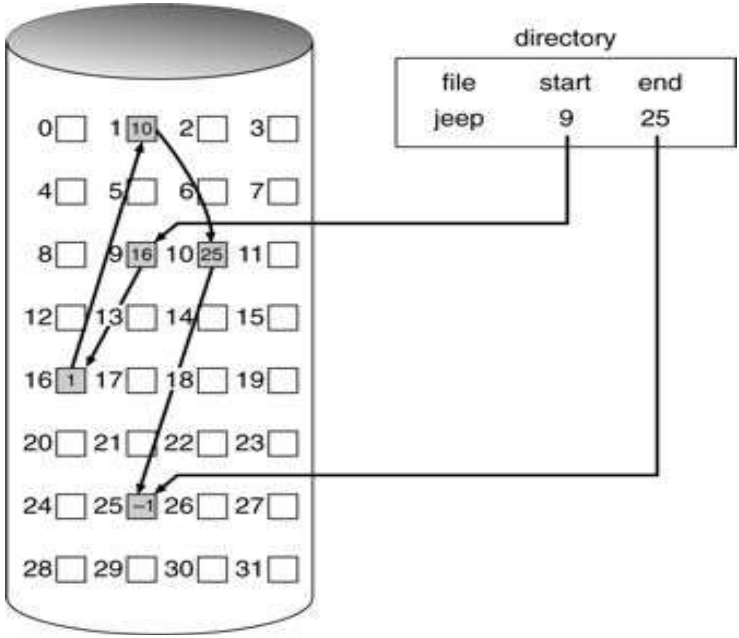


Fig.4.6: Linked allocation of disk space

Another disadvantage to linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512.byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space.

Yet another problem of linked allocation is reliability. Since the files are linked together by pointers scattered all over the disk, if a pointer were lost or damaged, a bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file.

4.7.3 Indexed allocation:

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order. **Indexed allocation** solves

this problem by bringing all the pointers together into one location: the **index block**.

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

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

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

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block (one or two pointers). With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-*nil*.

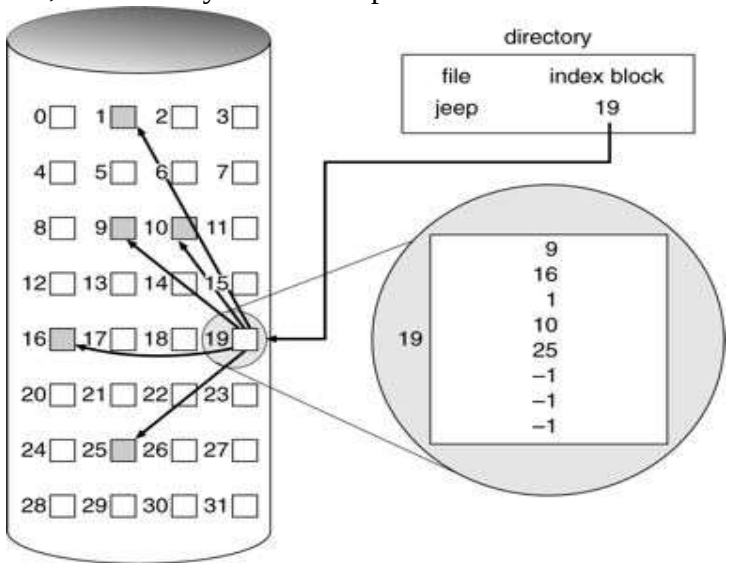


Fig.4.7: Indexed allocation of disk block

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

To allow for large files, several index files may be linked together. the linked representation is to use a first level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

4.8 Directory structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts.

First, disks are split into one or more partitions. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside.

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

The directory can be viewed as a symbol table that translates file names into their directory entries. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

When considering a particular directory structure, following are the operations that are to be performed on a directory:

- **Search for a file:** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Delete a file:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

Following are the most common schemes for defining the logical structure of a directory.

4.8.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand. A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of

files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

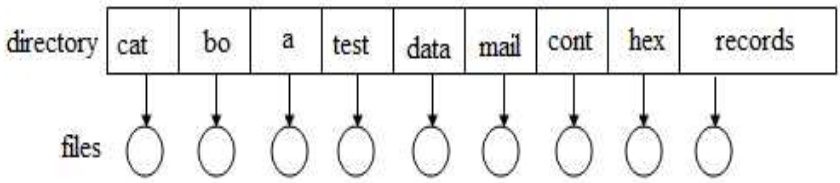


Fig.4.8 Single-level directory

4.8.2 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names between different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has their own user file directory (UFD). The UFD's have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user. When a user refers to a particular file, only her own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

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

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure

effectively isolates one user from another. This isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

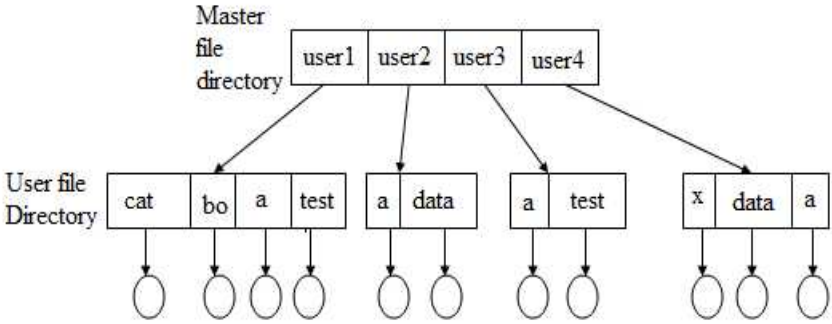


Fig.4.9 Two level directory structure.

4.8.3 Tree-Structured Directories:

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. The MS-DOS system, for instance, is structured as a tree. In fact, a tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.

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

Path names can be of two types: absolute path names or relative path names. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the

current directory. For example, in the tree-structured file system of Figure 4.9, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.

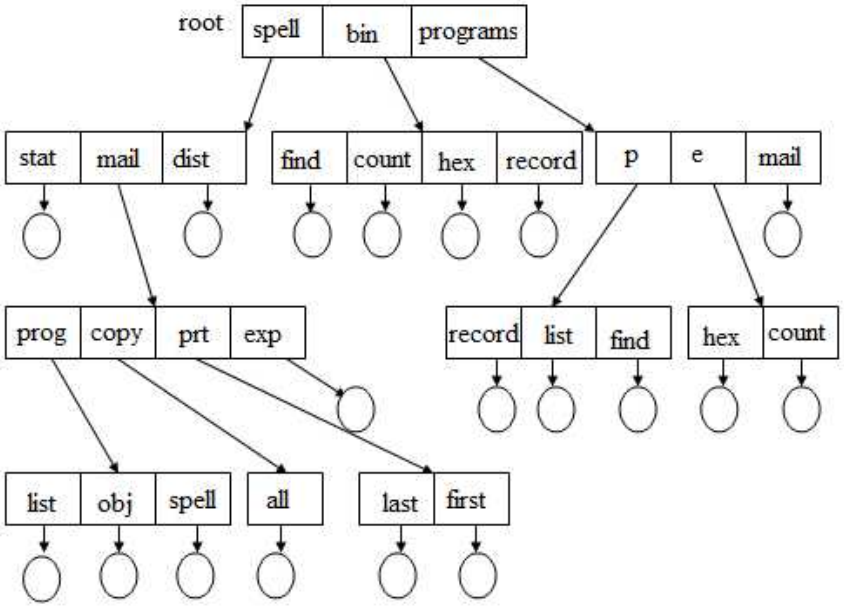


Fig.4.10: Tree-structured directory structure

4.8.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be shared. A shared directory or file will exist in the file system in two (or more) places at once.

An acyclic graph—that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 4.11). The same file or sub-directory may be in two different directories. The

acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

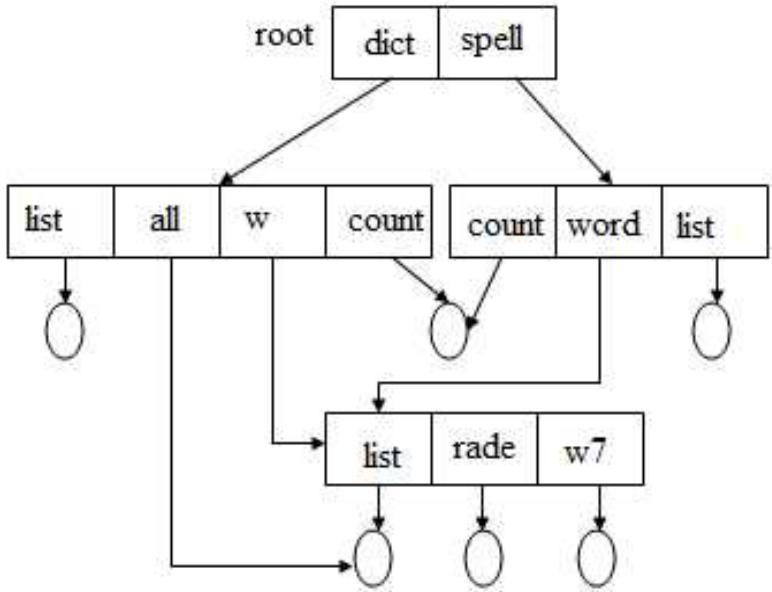


Fig.4.11: Acyclic Graph directory structure.

4.8.5 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory

preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 4.12).

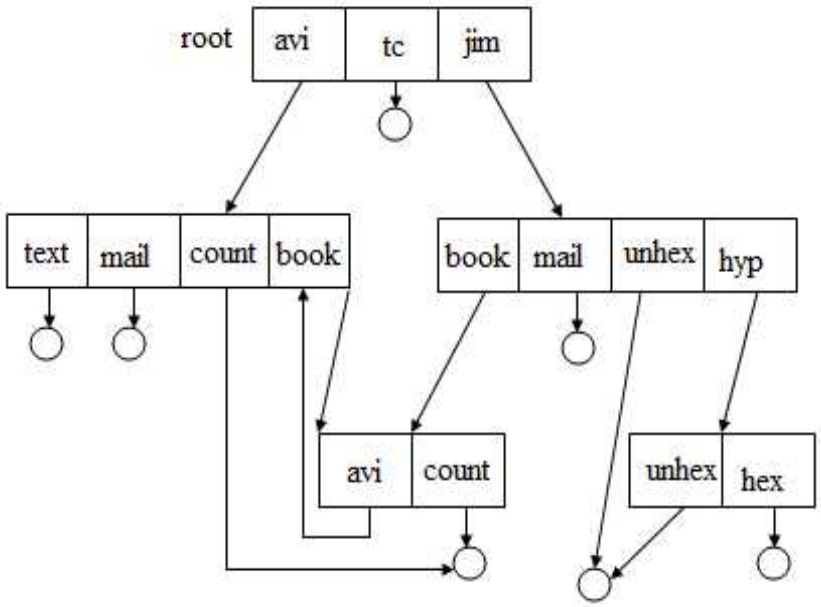


Fig.4.12 General graph directory

LINUX OPERATING SYSTEM

In 1991, Linus Torvalds wrote the first version of the Linux kernel. The first Linux systems were completed in 1992 by combining system utilities and libraries from the GNU project. Linux is used as an operating system for a wider variety of computer hardware than any other operating system including desktop computers, super computers, mainframes, and embedded devices such as cell phones.

Linux wizard is called kernel. The kernel is a file clerk that operates filing service. The primary difference between Linux and other contemporary operating systems is that the Linux kernel and other components are open source software.

Many Windows applications can be run on the Linux operating system.

4.9 Structure Of Linux Operating System:

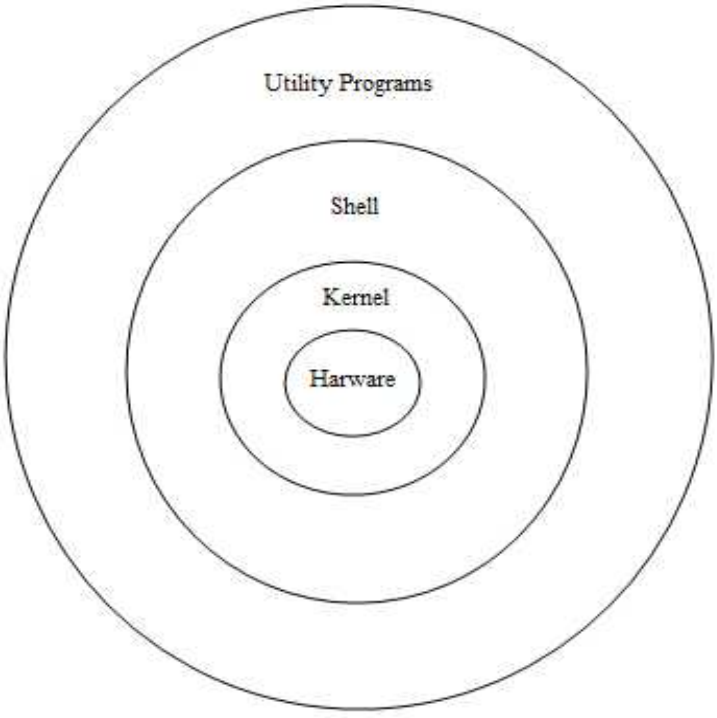


Fig.4.13: Structure of Linux operating system

The hardware is the different components which are attached to the computer.

The center of the Linux operating system is the kernel. The kernel is the piece of the software that provides an interface between user and the computer hardware and the attached peripherals. The kernel is responsible for maintaining the file system, executing commands, starting programs, timing system activities and managing system memory and other resources.

Shell is a command interpreter. It translates the user commands, given at the command prompt into kernel readable format. After that kernel can execute that commands.

Utility programs are different application programs through which users can interact with computer system, this consist of

database system, video games, business programs etc (depending on users interest).

4.10 Logging In And Logging Out:

Logging into a Linux account involves two steps,

i) Entering your user name and
ii) Entering your password.

Type the user name for your user account. If you make a mistake, you can erase characters with the BACKSPACE key. Consider the following example, the user enters the **aditi** and is then prompted to enter the password:

turtle login: aditi
password:

When user type in password, it does not appear on the screen. This is to protect password from being seen by others. If user enters either the username or the password incorrectly, the system will respond with the error message "Login incorrect" and will ask for username again, starting the login process over. User can then reenter username and password.

Once username and password entered correctly, user is logged into the system. User's command line prompt is displayed, waiting for user to enter a command. The command line prompt is a dollar sign ($) not a number sign (#).

The $ is the prompt for regular users, and the # is the prompt for the root user. The prompt is preceded by the hostname and the current user directory bounded by a set of brackets.

[turtle /home/aditi]$

To end the session logout or exit commands are used. This return to the login prompt and Linux waits for another user to log in.

[turtle /home/aditi]$ logout

Shutting Down from command line:
User can shutdown the system in either of two ways. First log in to an account and then enter the **halt** command. This command will log out and shut down the system.

$ halt

User can also use the **shutdown** command with the –h option to shut down the system. Or with the –r option, to shut down the system and then reboots. To shut down the system immediately, **+0** or **now** options are used.

shutdown –h now

4.11 Directory Structure

Linux sees each and everything as files. So it is important to understand exactly what a Linux file is and how it organizes files within the system. Every kind of file you might need is stored in Linux file system such as system files, data files, application files, utility files, driver files, configuration files and more. Each of these file has a name and a directory structure which is stored in file system.

The Linux file system is based on a tree structure like DOS, Microsoft and UNIX file system. There is a root directory, identified by '/' character, below which all other directories and subdirectories grow. The files are contained in these directories and subdirectories.

Directory names can be made up of a combination of numbers, symbols, and characters. The name of the directory includes the location of the directory i.e. the path of the directory to follow through the file system to get the directory in the file system.

Directories that are nested within other directories are called subdirectories.

The base of the directory structure is a root directory ('/'). It contains all the subdirectories and files in the Linux system.

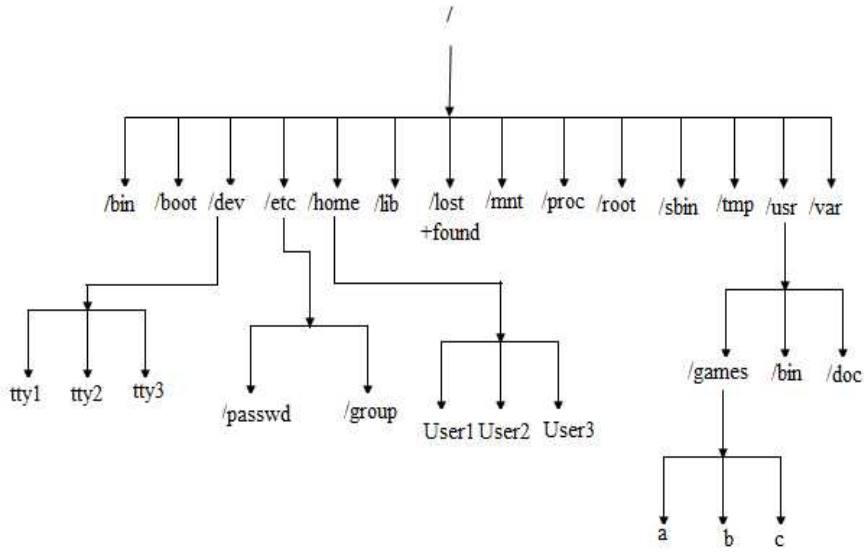


Fig 4.14.: Directory Structure in Linux

/bin: Executable files are stored in the /bin (binary) directory. In this directory Linux keeps its basic commands and programs. There are commands that starts the various shells, commands for working with files (like copying and moving), commands to change file permissions and configuration utilities.

/boot: In this directory the boot configuration files and commands are stored. This directory contains everything needed to boot the system.

/dev: In this directory all the device files (or drivers are kept for all computer hardware components. when we configure Linux to use a particular device, we have to configure particular file in this directory.

/etc: In this directory Linux keeps the system configuration files and initialization scripts. You can edit this files to add users to the

system and change their password(/etc/passwd file), create a group account for file sharing (/etc/group file).

/home: Each user on the system has a personal directory in which he can store his own created files. This directory contained in /home directory. In this directory each user has a separate subdirectory except the super user. Users home directories are for storing personal files and not accessible by other users on the system.

/lib: The /lib directory contains the libraries for c and other programming languages. It also contains the shared library images that are needed to boot the system and run commands.

/lost+found: In this directory use can look for a file if he think that Linux has lost it. If there are several partitions on the system, there is a /lost+found directory on each partition.

/mnt: This is a directory where other file system can be attached or mounted to the Linux file system. For example to view the contents of a CD-ROM, user can look in /mnt/cdrom directory. To see the contents of floppy disk, user can look in /mnt/floppy.

/proc: This directory contains virtual files that Linux uses to keep track of ongoing processes. They are virtual files and are not actually present there. They take up no space at all on the disk.

/root: The /root directory is the home directory for the super user or system administration.

/sbin: This directory contains the files for use by the super user (system administration. This directory contains commands that shut down the system, set the system clock, check the file system for errors and set up networking.

/tmp: Temporary files are located in /tmp directory. Here all users can temporarily store files so that other users can access them. All

the data stored in the /tmp directory will be lost when the system is rebooted.

/usr: This directory contains files that are not a part of the Linux file system. For example it contains applications for X Window System, Linux game collection and a long list of help files.

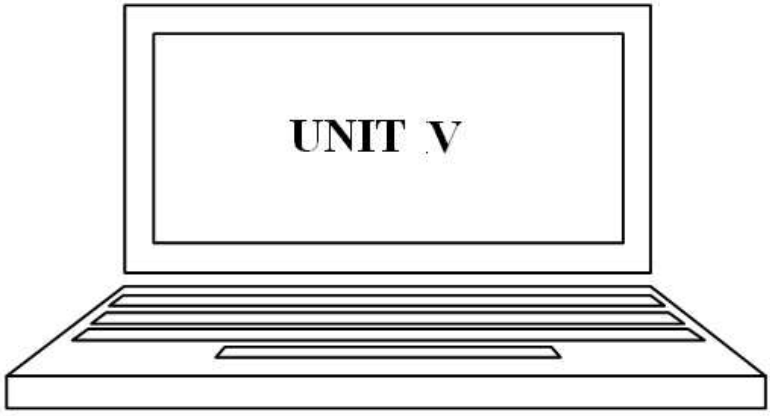
/var: This directory contains log files. The information in logs indicate what software was loaded at boot and how the system hardware is configured.

4.12 Naming Files and Directory:

Every file on the system are assigned names and a place where they resides in the system. First time when the user save a document, he need to select a directory in the file system in which to store the file and give the file name. Following are few steps to store documents in the right place and name in the file system.

- Determine the length of file name. the file name can contain as many as 256 character. File name includes the directory path that leads to the file. For example, the individual file named template.bmp located in /home/aditi/images directory is 31 characters long. The actual file name is /home/aditi/images/template.bmp.
- For naming files , you can use all the alphabets from A to Z (both lowercase and uppercase letters) or any numbers from 0 to 9. You can also use dash(-), underscores(_), and periods(.).
- Linux file names are case sensitive. An uppercase ‘A’ is not the same character as a lower case ‘a’. A file name template.bmp is a completely different file from a file named TEMPLATE.bmp.
- Avoid keyboard characters that causes problem in file names. Do not put a space in the filename and do not use following characters in file name.
  < > ‘ “ * { } [ ] ( ) ^ ! | % $? \

- To create hidden files, begin the filename with period (.). for example if you want to create a ‘file1.txt ‘ as hidden file, then give the file name as ’ .file1.txt.’
- Find the command line and change to your home directory. Type cd and press Enter to make sure that you are in your home directory.
- Create a file named firstfile. Type cat> firstfile and press Enter. The cursor will move to the next line on the screen. Type a few words and press Enter to move to the next line. Type a few words and press Ctrl+D to save and close the file.
- To read the file you just created, type cat firstfile and press Enter.



Shell and Basic Linux Commands

5.1 Shell:

The shell program is the interpreter through which we communicate with the operating system. As we type in a command at the shell prompt, it is interpreted and passed to the Linux kernel. The kernel tells the computer what to do. Shell is the command interpreter. It is the most widely used utility program in Linux. Shell offers the following features to the users.

i. Interactive Environment: The shell allows the user to create a dialogue between the user and the host system. This dialogue lasts until the user ends the session.

ii. Shell scripts: Shell script is the user interface to the computer system for the Linux operating system. The shell contains internal commands which can be utilized by the user. Shell scripts are groups of Linux commands strung together and executed as individual files

iii. Input/output redirections: Linux commands can be instructed to take their instructions from files, and not from the keyboard. The shell also allows the user to place the output of commands into a file and not on screen of the terminal. Output can also be redirected to other devices, such as to a printer or another terminal on the network.

iv. Piping Mechanism: Linux supplies 'pipe' facility that allows the output of one command to be used as input in another Linux command. The Linux file system contains many utilities that are useful for such purpose.

v. Metacharacter facility: Apart from the normal characters found on a keyboard, (letters and digits) the shell provides the use of 'metacharacters'. these allows the user to apply selection criteria when accessing files, for example the user could access all files that begin with the letter 'm'.

vi. Background processing: The true multi-tasking facilities allow the user to run commands in the background. This allows the command to be processed in background while the user can proceed with other task in foreground. When background task is completed, the user is notified.

There are many number of shells available in Linux, Bourne Shell is the oldest UNIX shell. The default shell on Linux distribution is the GNU Bourne again shell (bash). Shells available include the source version of the C Shell (tch), k shell (pdksh).

a. Bourne again Shell (bash): Bourne shell is the oldest UNIX shell. It is the most widely distributed UNIX shell. It is probably present on every existing UNIX system. The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey. The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell." The GNU/ Linux version of the Bourne (sh) shell is GNU Bourne again shell (bash). It is written by Brian Fox and Ramey. It is a feature enriched version of many of the feature found in the c shell (sh) and korn shell (ksh). The Bourne Again shell, bash, was developed as part of the GNU project and has replaced the Bourne shell, sh, for GNU-based systems like Linux. All major Linux distributions, including Red Hat, Slack ware, and Caldera, ship with bash as their sh replacement.

b. Korn Shell (ksh): Korn shell (ksh) is unix shell originally created by Dave Korn, is an also available in an open source form. The public domain free version of the shell created by Eric Gisin is called the public domain korn shell (pdksh). In march 2000, AT & T released a freely downloadable version of the original korn shell updated by Dave Korn in 1993, called 'ksh93'.

c. c shell: Bill Joy developed the C shell while he was at the University of California at Berkeley in the early 1980s. It was designed to make interactive use of the shell easier for users. Another design goal was to change the syntax of the shell from the

Bourne shell's older ALGOL style to the newer C style. The open source version of the c shell is tcsh.

5.2 Changing the running shell:

To check which shell is running , display the shell you are using. This information is found in SHELL environment variable. Type 'echo $SHELL' at the shell prompt and press Enter. If you are using bash shell system will display the following massage:

```
[aditi @localhost /aditi]$ echo $SHELL
/bin/bash
[aditi @localhost /aditi]$
```

If you are not using bash shell and want to change to bash shell, then type 'bash' at the shell prompt and press Enter. The system will display the following:

```
[aditi @localhost /aditi]$
```

Check again the shell you are using, type 'echo $SHELL' and press Enter.

To display the bash help function, type help and press Enter. The help command lists all the commands that are built into the shell.

5.3 Shell Prompt:

When you log in to your Linux account, you will see the shell prompt. It will look as follows:

```
[aditi@localhost/aditi]$
```

The first aditi is the username and it is followed by the hostname of the computer and the final aditi is the directory name that user aditi is in presently. You are automatically placed in your own user account when you first log in to the system.

When you are logged in as root, the shell prompt displayed is changes to use a pound sign (#) instead of the dollar sign ($).
[root@localhost/root]#

5.3.1 Changing the shell prompt:

The shell prompt can be changed to display information such as the username for a user, the directory in which the user is working, the current date and time or some special message. To change the shell prompt for your current login session, follow the steps given below:

1. Display the shell variables. At the shell prompt, type 'printenv' and press Enter. If you cannot read the entire screen, send the printenv command to the less pager.
(Type printenv |less)

2. Look for the PS1 variable. It may look as follows:
PS1=[\u@\r\w]\$

This variable tells the system to display the username for whoever is using the host computer and that user's current working directory inside brackets followed by the $ prompt (or # prompt for the root account).

3. Write down the information from the PS1 variable.

4. Decide how you want the prompt to look. Following table shows the characters to customize the shell prompt.

Character code	Description
\!	Display the number used in the history list to access the command that would be executed at the shell prompt.
\$	Display a $ in the prompt for regular user and a # for the super user.
\\	Display the backslash character.
\d	Display the current date
\h	Display the hostname of the computer at which you are working.
\t	Display the current time.
\u	Display the username of the user who is logged in to the system.
\w	Display the current working directory.
\xxx	Display any special comment.

Example: If you want to change the prompt so that it will display the time, the date, the current working directory and the $ prompt, then type PS1= '[\t \d \w]\$' and press Enter. The shell prompt will change to display the following: [12:18:40 Sat Jan 21 aditi]$

Once you log out of your account, the shell prompt will return to the default shell prompt.

5.4 Creating user account:

When you installed Linux, you created a superuser (root user) account. The shell loads automatically when you log in to your account. Superuser account should be used only for system administration and configuration job. To executing commands and applications and exploring Linux operating system you need to create a login for a user account. To create a login for new user 'adduser' and 'passwd' commands are used. Following are some steps to create a new user account.

1. Select a username. Each user on the system needs a unique login name to access the system and his user account. Usernames should be no more than eight characters. For the new user account the user name we have selected is ‘aditi’.

2. Choose a password. A good password should be a combination of uppercase letters, lowercase letters and numbers. A password should be at least eight characters.

3. Login as superuser. At the login prompt, type ‘root’ and press Enter. You are then prompted for password. Type the password and press Enter. The shell prompt for superuser account look like follows:

[root@localhost/root]#

4. Create the user account. At the shell prompt type ‘adduser aditi’ and press Enter. Your display should look like as follows:

[root@localhost /root]# adduser aditi
[root@localhost /root]#

5. Assign the password to the account. Type ‘passwd aditi’ and press Enter. You will see the following:

[root@localhost /root]# passwd aditi
Changing password for user aditi
New password:

Type the password and press Enter. If the system uses shadow passwords, the password does not appear on the screen. Verify the password by entering it second time and press Enter.

Retype new password:
Passwd: all authentication tokens updated successfully.
[root@localhost /root]#

6. Press Alt+F2 to display a new terminal window and login prompt is displayed as follows:

Linux Mandrake release 7.1b (hydrogen)
Kernel 2.2.15.0.25mdk on an 1586 / tty2
Localhost login:

Here tty2 is the virtual terminal that you are currently using. If you need to return to the superuser account, press Alt+F1.

7. Log in to your user account. At the login prompt, type the username and press Enter. Your shell prompt will be displayed as follows:

[aditi@localhost /aditi]$

5.5 Basic syntax for command:

Linux commands can be executed by typing its name on command line, followed by option list , and then parameters. A typical command line is as follows:

$ commandname –option [parameter]

The single line command is referred to as command line and it is typed after the shell prompt. The command name always appears first, it tells the shell WHAT to do. To execute the command press Enter. The minimum usage for most commands is the command and one or more command options.

Command options are typed after the command and are used to modify the result of the command. There must be a space between the command and the command option. Then, the dash (-) character must precede the command option. The dash tells Linux to treat each letter that follows the dash as a command option. There is no space between dash and the command option. There can be more than one command option, but do not put any space between the dash and the option.

For example:
$ls –l /usr/doc

The ls command displays a file list for the contents of the current working directory. The command option ls –l, which modifies the information displayed about each file in the listing. The parameter tells the command to list the contents of the /usr/doc directory instead of current working directory.

A command parameter can be a file, a directory, or a period of time. Parameters are not preceded by dash. Parameter specify, which file or directory a command is to be acted upon, or they can set a time limit in which the command is to be executed.

5.6 Creating alias for long command:

A command alias is a unique short name that is assigned to a command. It is possible assign a short name for a long command using alias. To create an alias for the following command

cd /user/doc/file1

Suppose ‘move’ is a sample alias name, to create this alias, type at the shell prompt:

$alias move= “cd/user/doc/file1”

The alias command is followed by the alias name. the equal sign (=)precedes the command that will be executed when the alias name is typed at the shell prompt. Do not give the space to the either side of equal sign. Now to change to ‘file1’ directory just type ‘move’ and press Enter.

5.7 Input/output Redirection:

When you execute a command, the command is read from the keyboard (the standard Input) and the result of the command(output) appears on the screen (the standard output). If you want the command output to appear in a different place, or if you want to send the output to another command for processing ,

then we need to change the output with process called Input/output redirection.

5.7.1 Redirecting Standard Output:

To redirect standard output, use '>' symbol. Placing '>' after a command (e.g. cat command) will direct its output to the filename following the symbol.

Using cat by itself simply outputs whatever you input to screen, as if it were repeating line you just typed.

Example:

```
$ cat
I am girl
I am girl
I am boy
I am boy
$
```

To redirect cat output to a file, type the following at shell prompt.

```
$ cat> file1.txt
Buy some sneakers
Then go to coffee shop
Then buy some coffee
^d
$
```

You can then use cat to read the file. At the shell prompt, type:

```
$ cat file1.txt
Buy some sneakers
Then goto coffee shop
Then buy some coffee
```

When you redirect the output to a file, you can overwrite an existing file, so that the name of the file you are creating does not match name of pre-existing file, unless you want to replace it.

Use output redirection again for another file and name it file2.txt
For example:

```
$ cat>file2.txt
Bring the coffee home
Make some coffee
Relax!
^d
$
```

Now, use cat to join file2.txt with file1.txt and redirect output of both files to a new
file file3.txt

$cat file1.txt file2.txt>file3.txt

After execution of the above command, execute the following command

$cat file3.txt
You will get the following output

```
Buy some sneakers
Then goto coffee shop
Then buy some coffee
Bring the coffee home
Make some coffee
Relax!
$
```

In this example, cat has added file1.txt and file2.txt and the output is stored to a new file file3.txt.

5.7.2 Appending standard output:

you can use output redirection to add new information to the end of existing file. Use '>>' to append data to an existing file.
When you use '>>' symbol, you are adding information to a file, rather than replacing the contents of a file entirely.
For example:
Take two already created files and join them by using the append symbol.

$cat file2.txt>>file1.txt

Here the contents of file2.txt are appended at the bottom of file1.txt.

```
$cat file1.txt
Buy some sneakers
Then goto coffee shop
Then buy some coffee
Bring the coffee home
Make some coffee
Relax!
```

5.7.3 Redirecting Standard Input:

It is possible to perform the same type of redirection with standard input. The standard input redirection symbol is '<', when you use this symbol, you are telling the shell that you want the input to be read from a file.

For example:
$cat<file1.txt

Here the output of file1.txt was read by cat command and it displays contents of the file file1.txt.

5.7.4 Pipe lines:

Pipes (| character) is used to redirect output from one command to become the input to another command. This reduces the necessity of writing new programs for complex operation. A pipe line is a mechanism, which accepts the output of a command as its input for the next command. Using the pipe operator "|" (vertical bar), the standard output of one command can be piped into the standard input of another:

command1| command2

for example:

```
$who|wc-l
13
$
```

The 'who' command produces a list of users, one user per line. Then wc command will count the number of lines and the output will appear on the screen.

Any command that writes to standard output can be used on the left hand side of pipe. Any command that reads from standard input can be used on the right hand side of pipe. Filters are commands that accepts data from standard input and write data to standard output, can be used anywhere in the pipeline. Multiple commands can be chained together with pipes.

5.7.5 Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as filters. Filters take input, change it somehow and then output it. The first one we will try is **sort**. Imagine we wanted to make a combined list of all of the executable programs in /bin and /usr/bin, put them in sorted order and view it:

```
[aditi@localhost ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (/bin and /usr/bin), the output of ls would have consisted of two sorted lists, one for each directory. By including sort in our pipeline, we changed the data to produce a single, sorted list.

uniq - Report Or Omit Repeated Lines

The uniq command is often used in conjunction with sort. uniq accepts a sorted list of data from either standard input or a single filename argument (see the uniqman page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the /bin and /usr/bin directories) we will add uniq to our pipeline:

```
[aditi@localhost ~]$ ls /bin /usr/bin | sort | uniq | less
```

In this example, we use uniq to remove any duplicates from the output of the sort command. If we want to see the list of duplicates instead, we add the "-d" option to
uniq like so:

```
[aditi@localhost ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc– Print Line, Word, And Byte Counts

The wc(word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
[aditi@localhost ~]$ wc ls-output.txt
7902 64566 503634 ls-output.txt
```

In this case it prints out three numbers: lines, words, and bytes contained in ls-output.txt. Like our previous commands, if executed without command line arguments, wc accepts standard input. The "-l" option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of programs we have in our sorted list, we can do this:

```
[aditi@localhost ~]$ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep– Print Lines Matching A Pattern

grep is a powerful program used to find text patterns within files. It's used like this:

grep pattern[file...]

When grep encounters a “pattern” in the file, it prints out the lines containing it. The patterns that grep can match can be very complex, but for now we will concentrate on simple text matches.

Let's say we want to find all the files in our list of programs that had the word “zip” embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[aditi@localhost ~]$ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for grep: “-i” which causes grep to ignore case when performing the search (normally searches are case sensitive) and “-v” which tells grep to only print lines that do not match the pattern.

head/ tail– Print First / Last Part Of Files

Sometimes you don't want all of the output from a command. You may only want the first few lines or the last few lines. The head

command prints the first ten lines of a file and the tail command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the "-n" option:

[aditi@localhost ~]$head -n 5 output.txt

Above command prints the first 5 lines of file output.txt.

[aditi@localhost ~]$tail -n 5 output.txt

The above command prints last 5 lines of file output.txt

tee– Read From Stdinput And Output To Stdoutput And Files

The tee creates a "tee" fitting on our pipe. The tee program reads standard input and copies it to both standard output and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including tee to capture the entire directory listing to the file ls.txt before grep filters the pipeline's contents:

```
[aditi@localhost ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

here the output of command 'ls/usr/bin' is stored in file 'ls.txt' after that grep searches the pattern 'zip'.

5.8 Listing files and directories: (ls command)

In UNIX there are three basic types of files:

- Ordinary Files
- Directories
- Special Files

An ordinary file is a file on the system that contains data, text, or program instructions. Directories store both special and ordinary files. Special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

ls command:

First, list the files and directories stored in the current directory. Use the following command:

$ ls

Here's a sample directory listing:

bin	hosts	lib	res.03
ch07	hw1	pub	test_results
ch07.bak	hw2	res.01	users
docs	hw3	res.02	work

This output indicates that several items are in the current directory, but this output does not tell us whether these items are files or directories. To find out which of the items are files and which are directories, specify the –F option to ls:

$ ls –F

Now the output for the directory is slightly different:

bin/	hosts	lib/
ch07	hw1	pub/
ch07.bak	hw2	res.01
docs/	hw3	res.02

As you can see, some of the items now have a /at the end: each of these items is a directory. The other items, such as hw1, have no character appended to them. This indicates that they are ordinary files. When the –F option is specified to ls, it appends a character indicating the file type of each of the items it lists.

Following table shows all the options for ls command:

Option	Function
-a	Displays all files including ., .. and hidden files
-A	Displays all but not . And ..
-C	List entries by column
-d	List directory entries instead of contents
-l	Use long listing format
-r	Sort in reverse order
-R	List subdirectories recursively
-s	Prints size of each file in blocks
-S	Sort by file size
-t	Sort by modification time,-u sort by access time

To list invisible files, specify the –a option to ls:

$ ls –a

The directory listing now looks like this:

.	.profile	docs	lib	test_results
..	.rhosts	hosts	pub	users
.emacs	bin	hw1	res.01	work
.exrc	ch07	hw2	res.02	
.kshrc	ch07.bak	hw3	res.03	

As you can see, this directory contains many invisible files. Notice that in this output, the file type information is missing. To get the file type information, specify the –F and the –a options as follows:

$ ls -a –F

The output changes to the following:

./	.profile	docs/	lib/	test_results
../	.rhosts	hosts	pub/	users
.emacs	bin/	hw1	res.01	work/
.exrc	ch07	hw2	res.02	
.kshrc	ch07.bak	hw3	res.03	

With the file type information you see that there are two hidden directories (.and ..). These two directories are special entries that are present in all directories. The first one, ., represents the current directory. The second one, .., represents the parent directory.

Option Grouping :
In the previous example, the command that you used specified the options to ls separately. These options can also be grouped together. For example, the commands

$ ls -aF
$ ls -Fa
are the same as the command
$ ls -a –F

5.9 cat command:
To view the content of a file, use the cat (short for concatenate) command. Its syntax is as follows:

$cat ‘files’

Here ‘files’ are the names of the files that you want to view. For example,

```
$ cat file1.txt
Buy some sneakers
Then go to coffee shop
Then buy some coffee
```

options:
-n : used to display numbering to the output lines. The numbered output shows us that the last line in this file is blank.

```
$ cat –n file1.txt
1       Buy some sneakers
2       Then go to coffee shop
3       Then buy some coffee
4
```

-b: skip numbering blank lines using the –b option:
In this case the output looks like the following:

```
$ cat –b file1.txt
1       Buy some sneakers
2       Then go to coffee shop
3       Then buy some coffee
```

The blank line is still there, it is no longer numbered.

5.10 wc command:
You can use the wc command to get a count of the total number of lines, words, and characters contained in a file. The basic syntax of this command is
$wc [options] files

Option	Description
-l	Counts the number of lines
-w	Counts the number of words
-c	Counts the number of characters

Files are the files you want examined.

5.11 Manipulating files and directories:

5.11.1 Copying Files (cp)

To make a copy of a file use the cp command. The basic syntax of the command is

$cp source destination

Here source is the name of the file that is copied and destination is the name of the copy. For example, the following command makes a copy of the file test_results and places the copy in a file named test_results.orig:

$ cp test_results test_results.orig

Copying Files to a Different Directory

If the destination is a directory, the copy has the same name as the source but is located in the destination directory. For example, the command

$ cp test_results work/

places a copy of the file test_results in the directory work.

5.11.2 Renaming Files & moving files and directories(mv):

To change the name of a file use the mv command. Its basic syntax is

$mv source destination

Here source is the original name of the file and destination is the new name of the file. As an example,

$ mv test_result test_result.orig

changes the name of the file test_result to test_result.orig. it is also used to move files and directories between different locations in the directory tree. The basic syntax is this:

$mv source destination
Here source is the name of the file or directory you want to move, and destination is the directory where you want the file or directory to end up. For example

$ mv /home/aditi/names /tmp

moves the file names located in the directory /home/aditi to the directory /tmp.
Moving a directory is exactly the same:

$ mv docs/ work/
moves the directory docs into the directory work.

5.11.3 Removing Files (rm):

To remove files use the rm command. The syntax is

$rm files

Here files is a list of one or more files to remove. For example, the command

$ rm res.01 res.02

removes the files res.01and res.02.

5.11.4 pwd command:

It displays the present working directory of user.
$ pwd
/home/aditi

This indicates that I am in my home directory. Your home directory is the initial directory where you start when you log in to a UNIX machine.

5.11.5 Changing Directories (cd):

You can use the cd to change to any directory by specifying a valid path. The syntax is as follows:
$cd directory
Here, directory is the name of the directory that you want to change to. For example, the command

$ cd /usr/local/bin

changes to the directory /usr/local/bin.

5.11.6 Creating Directories(mkdir):

You can create directories with the mkdir command. Its syntax is

$mkdir directory

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command

$ mkdir new
creates the directory 'new' in the current directory. Here is another example:

$ mkdir /tmp/test-dir
This command creates the directory test-dir in the /tmp directory. The mkdir command produces no output if it successfully creates the requested directory.

If you give more than one directory names on the command line, mkdir creates each of the directories. For example

$ mkdir docs pub

creates the directories docs and pub under the current directory.

5.11.7 Removing directories(rmdir):

This command is used to remove directories. Its syntax is :

$rmdir directories

Here, directories includes the names of the directories you want removed. For example, the command

$ rmdir ch01 ch02 ch03

removes the directories ch01, ch02, and ch03 if they are empty. The rmdir command produces no output if it is successful. A directory can be deleted only if it is empty(does not contain files or subdirectory). Current directory can not be deleted. Complete path names can also be specified with rmdir as shown in mkdir command.

5.12 vi Editor:

The first version of vi was written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems. vi derives its name from the word “visual,” because it was intended to allow editing on a video terminal with a moving cursor.

The vi editor is the original editor used on Unix systems. It uses the console graphics mode to emulate a text-editing window, allowing you to visually see the lines of your file, move around within the file, and insert, edit, and replace text.

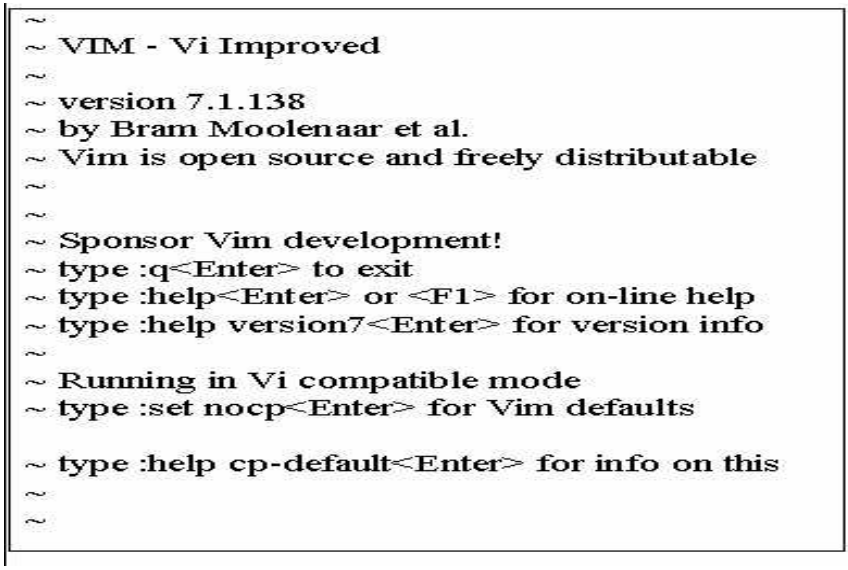
Most Linux distributions don't include real vi; rather, they ship with an enhanced replacement called vim (which is short for “vi improved”) written by Bram Moolenaar. vim is a substantial improvement over traditional Unix vi and is usually symbolically linked (or aliased) to the name “vi” on Linux systems. In the discussions that follow, we will assume that we have a program called “vi” that is really vim.

5.12.1 Starting And Stopping vi:

To start vi, we simply type the following:

[aditi@localhost/aditi]$vi

And a screen like this should appear:



To exit, we enter the following command (note that the colon character is part of the command):

:q

The shell prompt should return. If, for some reason, vi will not quit (usually because we made a change to a file that has not yet been saved), we can tell vi that we really mean it by adding an exclamation point to the command:

:q!

If you get “lost” in vi, try pressing the Esc key twice to find your way again.Try running vim instead of vi. If that works, consider adding alias vi='vim' to your shell file.

5.12.2 Editing Modes:

Let's start up vi again, this time passing to it the name of a non existing file. This means we are creating a new file with vi:

[aditi@localhost/aditi]$vi first.txt

If all goes well, we should get a screen like this:

The leading tilde characters ("~") indicate that no text exists on that line. This shows that we have an empty file. Do not type anything yet! The second most important thing to learn about vi(after learning how to exit) is that vi is a modal editor. When vi starts up, it begins in command mode. In this mode, almost every key is a command, so if we were to start typing, vi would basically go crazy and make a big mess.

5.12.3 Insert Mode(Input mode):

In order to add some text to our file, we must first enter insert mode by pressing the "i" key. We should see the following at the bottom of the screen if vim is running in its usual enhanced mode.

--INSERT--

Now we can enter some text. Try this:

This is the first day of my college.

To exit insert mode and return to command mode, press the Esc key.

5.12.4 Saving Our Work:

To save the change we just made to our file, we must enter an ex command while in command mode. This is easily done by pressing the ":" key. After doing this, a colon character should appear at the bottom of the screen:

:

To write our modified file, we follow the colon with a "w" then Enter:

:w

The file will be written to the hard drive and we should get a confirmation message at the bottom of the screen, like this:

"first.txt" [New] 1L, 36C written

Following table shows the saving and quitting commands.

Command	Action
:w	Saves files and remain in editing mode
:x	Saves files and quits editing mode
:wq	Saves files and quits editing mode
:q	Quits editing mode when no change are made to file
:sh	Escape to the shell

5.12.5 Moving The Cursor Around:

While in command mode, vi offers a large number of movement commands, some of which are given in following table:

Key	Cursor movement
l or Right Arrow	Right one character
h or Left Arrow.	Left one character
j or Down Arrow	Down one line.
k or Up Arrow	Up one line.
0 (zero)	To the beginning of the current line.
^	To the first non-white space character on the current line.
$	To the end of the current line.
w	To the beginning of the next word or punctuation character.
W	To the beginning of the next word, ignoring punctuation characters.
b	To the beginning of the previous word or punctuation character.
B	To the beginning of the previous word, ignoring punctuation characters.
Ctrl-f or Page Down	Down one page.
Ctrl-b or Page Up	Up one page.
numberG	To line number. For example, 1Gmoves to the first line of the file.

5.12.6 Adding new text in existing file:

Following keys are used to **append text** to the file in Input mode:

Key	Function
a	Append text to the right of the current cursor position.
A	Append text to the end of line.

If we move the cursor to the end of the line and type “a”, the cursor will move past the end of the line and vi will enter insert mode. This will allow us to add some more text:

This is the first day of my college. It is cool.

Now press the Esc key to exit insert mode.

Move the cursor to the beginning of the line using the “0” (zero) command.

Now we type “A” and add the following lines of text:

```
This is the first day of my college. It is cool.
Line 2
Line 3
Line 4
Line 5
```

Again, press the Esc key to exit insert mode.

Following keys are used to **open new line** in file in Input mode:

Key	Function
o	Opens line below the current cursor position.
O	Opens line above the current cursor position.

Now place the cursor on “Line 3” then press the o key.

```
This is the first day of my college. It is cool.
Line 2
Line 3

Line 4
Line 5
```

A new line was opened below the third line and we entered insert mode. Exit insert mode by pressing the Esc key. Press the u key to undo our change.

Press the O key to open the line above the cursor:

```
This is the first day of my college. It is cool.
Line 2

Line 3
Line 4
Line 5
```

5.12.7 Deleting Text:

For text deletion many commands are used, they are given in following table.

Command	Deletes
x	The current character.
3x	The current character and the next two characters.
dd	The current line.
5dd	The current line and the next four lines.
dW	From the current cursor position to the beginning of the next word.
d$	From the current cursor location to the end of the current line.
d0	From the current cursor location to the beginning of the line.
d^	From the current cursor location to the first non-white space character in the line.
dG	From the current line to the end of the file.
d20G	From the current line to the twentieth line of the file.

Place the cursor on the word “It” on the first line of our text. Press the x key repeatedly until the rest of the sentence is deleted. Next, press the u key repeatedly until the deletion is undone.

Again, move the cursor to the word “It” and press dW to delete the word:

This is the first day of my college. is cool.
Line 2
Line 3
Line 4
Line 5

5.12.8 Cutting, Copying And Pasting Text:

The d command not only deletes text, it also “cuts” text. Each time we use the d command the deletion is copied into a paste buffer (think clipboard) that we can later recall with the p command to paste the contents of the buffer after the cursor or the P command to paste the contents before the cursor.

The y command is used to “yank” (copy) text in much the same way the d command is used to cut text. Here are some examples combining the y command with various movement commands:

Command	Copies
yy	The current line.
5yy	The current line and the next four lines.
yW	From the current cursor position to the beginning of the next word.
y$	From the current cursor location to the end of the current line.
y0	From the current cursor location to the beginning of the line
y^	From the current cursor location to the first non-white pace character in the line.
yG	From the current line to the end of the file.
y20G	From the current line to the twentieth line of the file.

Place the cursor on the first line of the text and type yy to copy the current line. Next, move the cursor to the last line (G) and type p to paste the line below the current line:

```
This is the first day of my college. It is cool.
Line 2
Line 3
Line 4
Line 5
This is the first day of my college. It is cool.
```

Just as before, the u command will undo our change. With the cursor still positioned on the last line of the file, type P to paste the text above the current line:

5.13 Compressing files (gzip, gunzip commands):

The **gzip** program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding gunzip program is used to restore compressed files to their original, uncompressed form. Here is an example

```
[aditi@localhost ~]$ ls -l /etc > first.txt
[aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 15738 2008-10-14 07:15 first.txt
[aditi@localhost ~]$ gzip first.txt
 [aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 3230 2008-10-14 07:15 first.txt.gz
[aditi@localhost ~]$ gunzip first.txt
 [aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 15738 2008-10-14 07:15 first.txt
```

In this example, we create a text file named first.txt from a directory listing. Next, we run gzip, which replaces the original file with a compressed version named first.txt.gz. In the directory

listing of first.*, we see that the original file has been replaced with the compressed version, and that the compressed version about one-fifth the size of the original. We can also see that the compressed file has the same permissions and time stamp as the original. Next, we run the gunzip program to uncompress the file. Afterward, we can see that the compressed version of the file has been replaced with the original, again with the permissions and time stamp preserved.

bzip2

The bzip2 program, by Julian Seward, is similar to gzip, but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as gzip. A file compressed with bzip2 is denoted with the extension .bz2:

```
[aditi@localhost ~]$ ls -l /etc > first.txt
[aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 15738 2008-10-14 07:15 first.txt
[aditi@localhost ~]$ bzip2  first.txt
 [aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 3230 2008-10-14 07:15 first.txt. bz2
[aditi@localhost ~]$ bunzip2   first. bz2
 [aditi@localhost ~]$ ls -l first.*
-rw-r--r-- 1 aditi   aditi 15738 2008-10-14 07:15 first.bz2
```

As we can see, bzip2can be used the same way as gzip.

5.14 Archiving Files(tar):

Archiving is the process of gathering up many files and bundling them together into a single large file. Archiving is often done as a

part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

tar:

The tar program is the classic tool for archiving files. Its name, short for tape archive, reveals its roots as a tool for making backup tapes .While it is still used for that traditional task, it is equally adept on other storage devices as well. We often see filenames that end with the extension .tar or .tgz which indicate a "plain" tar archive and a gzipped archive, respectively. A tar archive can consist of a group of separate files, one or more directory hierarchies, or a mixture of both. The command syntax is:

tar mode[options] pathname...

where mode is one of the following operating modes :

Mode	Description
c	Create an archive from a list of files and/or directories.
x	Extract an archive.
r	Append specified pathnames to the end of an archive.
t	List the contents of an archive.

For creating archive of directory 'newone' which contains many file issue the following command:

[aditi@localhost ~]$ tar cf newone.tar newone

This command creates a tar archive named newone.tar that contains the entire dectory hierarchy. We can see that the mode and the f option, which is used to specify the name of the tar archive, may be joined together, and do not require a leading dash. Note, however, that the mode must always be specified first, before any other option.

To list the contents of the archive, we can do this:

[aditi@localhost ~]$ tar tf newone.tar

For a more detailed listing, we can add the v(verbose) option:

[aditi@localhost ~]$ tar tvf newone.tar

Now, let's extract the newone in a new location. We will do this by creating a new directory named foo, and changing the directory and extracting the tar archive:

```
[aditi@localhost ~]$ mkdir foo
[aditi@localhost ~]$ cd foo
[aditi@localhost ~]$ tar xf ../ newone.tar
[aditi@localhost foo]$ ls
newone.tar
```

If we examine the contents of ~/foo/newone, we see that the archive was successfully installed, creating a precise reproduction of the original files. There is one caveat, however: unless you are operating as the superuser, files and directories extracted from archives take on the ownership of the user performing the restoration, rather than the original owner.

5.15 Managing disk space: df, du

A quick way to get a summary of the available and used disk space on your Linux system is to type in the df command. The command df stands for "disk file system". With the -h option (df -h) it shows the disk space in "human readable" form, which in this case means, it gives you the units along with the numbers.

The output of the df command is a table with six columns. The first column contains the file system path, which can be a reference to a hard disk or another storage device, or a file system connected through the network. The second column shows the capacity of

that file system. The third column shows the used space, the fourth column shows the available space and the fifth column shows the used space percentage and the last column shows the path on which that file system is mounted. The mount point is the place in the directory tree where you can find and access the file system.

The *df* tool simply reports the amount of free space on each partition — how large they are, etc. It also provides information on non-local file systems (such as mounted NFS or Samba shares). In its most basic form, *df* provides the following:

```
$ df
Filesystem     1K-blocks     Used  Available     Use%  Mounted on
/dev/md2        4881472   793508   4087964      17%  /
```

However, you can add options to *df* to show the file system type and show the sizes in an easier to understand format:

```
$ df -h -T
Filesystem     Type    Size    Used   Avail   Use%  Mounted on
/dev/md2       xfs     4.7G    775M   3.9G    17%   /
```

While *df* provides an overview of entire partitions, the *du* tool will summarize the size of a given directory, broken down by subdirectories: The du command shows the disk space used by the files and directories in the current directory. Again the -h option (df -h) makes the output easier to comprehend. By default, the du command lists all subdirectories to show how much disk space each has occupied. This can be avoided with the -s option (df -h -s). This only shows a summary. Namely the combined disk space used by all subdirectories. If you want to show the disk usage of a directory (folder) other than the current directory, you simply put that directory name as the last argument. For example: du -h -s images, where "images" would be a subdirectory of the current directory.

```
$ du svn/ports
...
32      svn/ports/vnstat/.svn
48      svn/ports/vnstat
6248    svn/ports
```

Of course, to summarize the directory and all subdirectories and display size values in a human-readable format, use:

```
$ du -sh svn/ports
6.2M    svn/ports
```

5.16 Changing Your Password

To set or change a password, the passwd command is used. The command syntax looks like this:

passwd [user]

To change your password, just enter the passwd command. You will be prompted for your old password and your new password:

```
[aditi@localhost ~]$ passwd
(current) UNIX password:
New UNIX password:
```

The passwd command will try to enforce use of "strong" passwords. This means the it will refuse to accept passwords that are too short, too similar to previous passwords, are dictionary words, or too easily guessed:

```
[aditi@localhost ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

If you have superuser privileges, you can specify a user name as an argument to the passwd command to set the password for another user. There are other options available to the superuser to allow account locking, password expiration, etc.

5.17 File access permissions:

Every file on Linux has three types of permissions, read, write and execute. The command **chmod** allows to decide who can and cannot read, write and use a file. To assign these three types of permissions, the following three symbols are used.

Access type	Symbol	Meaning
Read	r	Allows displaying, copying and compiling a file.
write	w	Allows editing and deleting of file.
Execute	x	Allows execution of a file

To specify then user to whom you are grant these permissions, use these three symbols.

Symbol	Description
u	The user or owner of file or directory
g	Members of the group to which user belongs
o	All other system users

Normally when a file is created, it is created with read, write and execute permissions to the user and only read and execute permissions for group members and other system users. Write permission is not given normally to the group and other users. If it is required, the user can assign permissions by system commands. The currently existing permissions of files and directories can be determined by using long listing of directories (ls –l) command.

For example:
User is logged in with name 'funny'

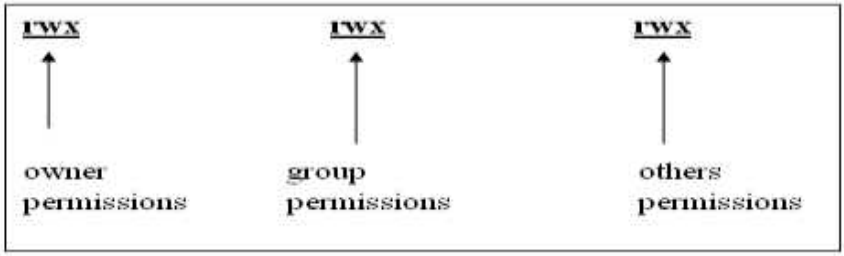
```
$ls –l
Total 3
drwx- -x- -x funny project 50 june 2 15:30 dir1
-rwx r-x r-x funny project 1120 june  1 09:50 prog1
-rwx r- - r- -funny project 2000 june 6 10:40 sample1
```

The explanation of the permissions for the prog1 and sample1 files and directory dir1 is given in following table:

Permissions	Files/directories	Remarks
-rwx r - x r - x	prog1(file)	User/owner has got all the three permissions. Group and others have got only read and execute permissions.
-rwx r- - r - -	sample1(file)	Sample1 denies write and execute permissions for group members and others.
drwx r - - r - -	dir1(directory)	dir1 directory denies permission for write and execute to group and others.

The initial character describes the file or directory.
'-' (dash) symbolizes an ordinary file and 'd' symbolizes a directory. There are nine possible permissions which can be given

or refused in combination. They are written as string of nine characters.



For example: rwx r-x r-x
In this example user(owner) has all the three permissions read (r) , write(w), execute(x) and group and other system users have read(r) and execute (x) permissions.

5.18 Granting access to files: (chmod command)

You can change existing permissions by executing chmod command. This command uses two methods:
1. Symbolic method 2. Octal method

Symbolic method:

Syntax of symbolic method is given below:

chmod who + permissions file(s)

where:
chmod -name of command.
who -one of three user groups (u, g, o or a) u=user/owner, g=group,
o=others a=all
+ -means grant permissions
- -means deny permissions
Permissions –any combination of three (r, w, x)
Files -file or directory name

For example:
$chmod u+r prog1
This command added read permission to the owner for file prog1.
$chmod u-w prog1
This command denies write permission to the owner for file prog1.
If you omit the scope(u, g or o) the permissions given apply to all the three groups.

Octal method:
This format requires to specify permissions using three octal numbers, ranging from 0 to 7. the octal code can be one to three digit long. the first, second and third digits set the user, group and other user permissions respectively. The integer used in each position dictates the permissions for the files. Following table shows the digits with meaning.

Integer	Permissions
0	No permission
1	execute
2	write
4	read
5	Execute and read
6	Write and execute
7	Read, write and execute

For example:
$chmod 750 sample1

This examples changes access permissions for the file sample1 in the current working directory. The 7 gives permissions for the owner of the file to read, write and execute that file, the 5 gives permission for group members to read and execute that file and 0 denies permissions for other users to read, write or execute that file.

5.19 Creating group account

The command **groupadd** is used to create a new group. The **groupadd** command creates a new group account using the values specified on the command line plus the default values from the system. The new group will be entered into the system files as needed.

Syntax:
groupadd [options] group

Options:

-f, **--force**

This option causes the command to simply exit with success status if the specified group already exists. When used with **-g**, and the specified GID already exists, another (unique) GID is chosen (i.e. **-g** is turned off).

-g, **--gid** GID

The numerical value of the group's ID. This value must be unique, unless the **-o** option is used. The value must be non-negative. The default is to use the smallest ID value greater than 999 and greater than every other group. Values between 0 and 999 are typically reserved for system accounts.

-h, **--help**

Display help message and exit.

5.20. Sudo command:

The administrator can configure sudo to allow an ordinary user to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. The sudo provides superuser privileges. The use of sudo does not require access to the superuser's password. To authenticate using sudo, the user uses his/her own password.

Let's say, for example, that sudo has been configured to allow us to run a fictitious

backup program called “backup_script”, which requires superuser privileges. With sudoit would be done like this:

```
[aditi @localhost ~]$ sudo backup_script
Password:
System Backup Starting...
```

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out.

5.21 chown – Change File Owner And Group

The **chown** command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of **chown** looks like this:

chown [owner][:[group]] file

chown can change the file owner and/or the file group owner depending on the first argument of the command. Here are some examples:
chown Argument Examples:

Argument	Results
bob	Changes the ownership of the file from its current owner to user bob.
bob:users	Changes the ownership of the file from its current owner to user bob and changes the file group owner to group users.
:admins	Changes the group owner to the group admins. The file owner is unchanged.
bob:	Change the file owner from the current owner to user bob and changes the group owner to the login group of user bob.

Let's say that we have two users: adi1, who has access to super-user privileges and aditi, who does not. User adi1 wants to copy a file from her home directory to the home directory of user

aditi. Since user adi1 wants aditi to be able to edit the file, adi1 changes the ownership of the copied file from adi1 to aditi:

```
[adi1 @linuxbox ~]$ sudo cp myfile.txt ~ aditi
Password:
[adi1 @linuxbox ~]$ sudo ls -l ~aditi/myfile.txt
-rw-r--r-- 1 root root 8031 2008-03-20 14:30 /home/ aditi /myfile.txt
[adi1 @linuxbox ~]$ sudo chown aditi: ~ aditi /myfile.txt
[adi1 @linuxbox ~]$ sudo ls -l ~ aditi /myfile.txt
-rw-r--r-- 1 aditi aditi 8031 2008-03-20 14:30 /home/ aditi /myfile.txt
```

Here we see user adi1 copy the file from his directory to the home directory of user aditi. Next, adi1 changes the ownership of the file from root (a result of using sudo) aditi. Using the trailing colon in the first argument, adi1 also changed the group ownership of the file to the login group of aditi, which happens to be group aditi. Notice that after the first use of sudo, adi1 was not prompted for her password? This is because sudo, in most configurations, "trusts" you for several minutes until its timer runs out.

5.22 Communication commands:

5.22.1 who

This command displays a list of all the people, or users, who are currently using the UNIX machine. The first column of the output lists the usernames of the people who are logged in.

```
$ who
atha tty1 Dec 6 19:36
yash ttyp2 Dec 6 19:38
sanu ttyp0 Dec 9 09:23
$
```

You can see that there are three users, atha, yash, and sanu. The second column lists the terminals they are logged in to, and the final column lists the time they logged in. The output varies from system to system.

5.22.2 who am i

You can use the **who** command to gather information about yourself when you execute it as follows:

```
$ who am i
atha pts/0 Dec 9 08:49
$
```

This tells me the following information:
l My username is atha.
2 I am logged in to the terminal pts/0.
3 I logged in at 8:49 on Dec 9.

The arguments **am** and **i** change the behavior of the who command to list information about you only. In UNIX, most commands accept arguments that modify their behavior.

5.22.3 mesg command:

Syntax:
mesg [y] [n]

The **mesg** command stops the incoming messages that were initiated through other UNIX communication commands.

This command allows users to set the permissions on their terminal so that read and write access is denied to all users apart from themselves i.e., the owner of the terminal. **Mesg** has two arguments: 'y' and 'n' (yes/no). **mesg** used without any arguments simply returns the state of the current setting i.e.

y -message can be sent to the terminal, or
n -message cannot be sent to the terminal
we have to specify a single argument at time for example:

$mesg n
This command disabled all the incoming messages.

$mesg y
This command allows all the incoming messages from other users through communication commands.

5.22.4 write command:

Syntax:
write <username> [ttyname]

This is two way communication command. Here <username> is the recipient user, and [ttyname], is the name of the terminal that appears in the /dev directory. With the use of write command it is possible to write a message to the other user's terminal and that recipient user can also write back to the sender's terminal and thus a two way communications link can be established between two users.

In this command a username must be specified, and an optional terminal code can be included, if required, for example when a user is logged in on more than one terminal and you want to contact to a specific terminal.

When **write** is invoked, it reads from the standard input, so you can simply type in your message, in the required format. You can end the session by pressing <ctrl-d>, and the message will be dispatched. The sender will be returned to the shell prompt. If the person whom you were contacting was in the middle of another task e.g. in an editing session etc., they will have to escape to the shell level to write back. To check how many users are currently logged in you can use the 'who' command.

For example:

```
$ who
aditi tty20 jan 20 15:10
aditi tty15 jan 20 15:20
$ write aditi tty20
Hi Aditi, How are you?
<ctrl-d>
$
```

In the above example we used both the options terminal name and username.

If we tried to the user who is not logged in we receive a following error message:

```
$ write atha
write: atha not logged in
$
```

When you receive a message from a user , Linux displays a one-line message indicating the name of the person who is contacting you, the terminal from which the message originated, and the date and time the message was sent. A small beep also accompanies the message to attract your attention:

```
$
message from aditi (tty15) Jan 20 16:20:33
Hi Aditi, How are you?
$
```

5.22.5 talk command:

The **talk** command is similar to write command except that it reserves a separate area of screen for both the sender and recipient users. The syntax is exactly the same as the write command.

Syntax:
talk <username> [ttyname]

When **talk** is activated with valid username or with a valid terminal name or both, the current screen is cleared, and the sender's screen is divided into two areas, separated by a single line. The recipient user's screen, when he or she responds to the sender, will also be divided into two separate parts. The top half of the screen is where the sender will type a message to be sent to the recipient. Assume we had typed the following command:

$ talk aditi
This command displays the following screen:

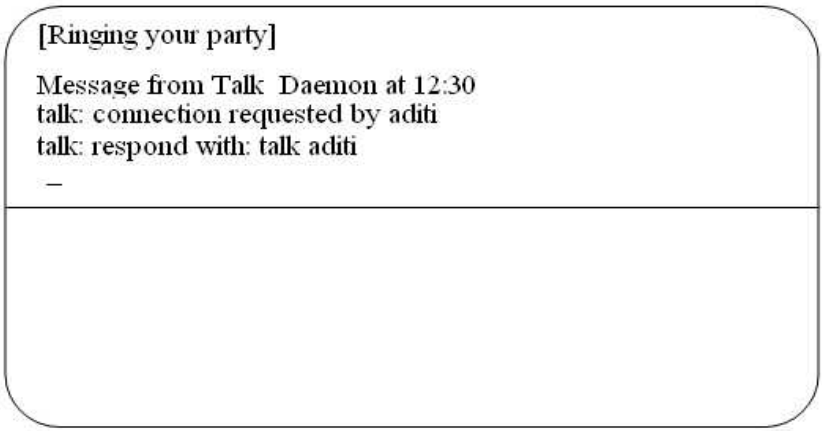


Fig 5.1: a typical screen when talk is invoked

A two screened window is opened, and **talk** contact the recipient user, in this case user aditi. When talk is initially invoked, a message is displayed on the recipient's screen.

Message from Talk Daemon at 12:30
talk: connection requested by aditi
talk: respond with: talk aditi

This message tells the recipient that somebody is trying to make contact, and that the recipient can respond with the command 'talk aditi'. It is necessary for both the sender and recipient to each invoke the talk command.

When the connection with the recipient user has been established i.e. the recipient user has invoked **talk** with the sender's recipient name, **talk** display the message 'Connection established' and both users can proceed to type their messages. Following figure illustrate a typical conversation after a connection has established between the sender and recipient party.

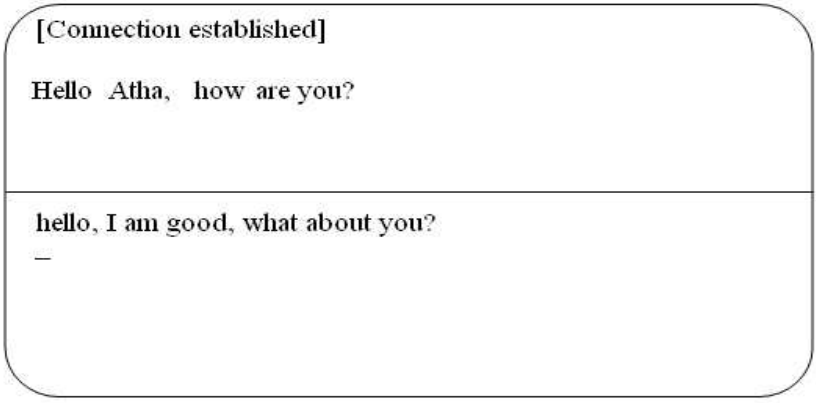


Fig.5.2: typical conversation using talk command.

To a conversation, the <ctrl-c> (interrupt) or <ctrl-d> key can be used to close the connection. Both the sender and recipient will then receive a message indicating that the connection has been terminated.

5.22.6 wall command:

Wall stands for write all. It writes a message on a recipient's terminal, reading the message from the standard input (keyboard). **Wall** delivers the message to all the users on the system. Once the command has read the message from the standard input, it proceeds to write it simultaneously on every user's terminal.

The super-user normally uses the **wall** command to deliver important system messages and can override any **mesg n** setting (since the super user can write to any terminal at any time whatever protection setting are in effect).

For example:

```
$ wall
Save and logout immediately, the system is going down in 5 minutes. Another
warning will follow in 3 minutes.
<ctrl-d>
```

When this wall message is received by the users on the network, the message may take the following form:

Broadcast message from 'root' 20 Jan 12:20:30
Save and logout immediately, the system is going down in 5 minutes. Another warning will follow in 3 minutes.

Since **wall** messages are broadcasted over an entire network, the header indicates this with the announcement 'Broadcast message'. The date and time of broadcast are commonly included in the statement.

References

1. Operating Systems by P. Balakrishna Prasad [Scitech Publication]
2. Operating System concepts : James L.Peterson Abraham Silberschatz [Addison-Wesley Publishing Company].
3. Operating System Concepts : Silbershaz [Addision Education]
4. SAMS Teach Yourself Linux by Craig and Coletta Witherspoon [Techmedia]
5. LINUX complete reference by Richard Peterson
6. Unix The complete Guide by Jason J. Manger[Galgotia Publication Pvt. ltd]
7. Operating System Concepts 8th Edition by Silberschatz , Galvin, Gagne

OPERATING SYSTEM CONCEPTS AND BASIC LINUX COMMANDS

Author of this book is self employed from last two years. She has worked as Lecturer in computer science department of polytechnic, MCA, and science colleges. She completed her bachelor degree B.Sc in computer science from Nagpur University in the year 2000, and Master in computer applications from Nagpur University in the year 2003.she started working as lecturer from the year 2004.

This book contains the introductory information about the operating system and the basics of Linux commands for graduation level studies. This book provides the concepts of operating system. It contains the fundamental concepts which are applicable for various operating systems.

EDUCREATION
PUBLISHING (Delhi)
www.educreation.in

Also available as an eBook
ACADEMIC
ISBN 978-1-5457-0850-7
9 781545 708507