Srinivas Institute of Management Studies

Pandeshwar, Mangalore – 575 001

BACKGROUND STUDY MATERIAL

OPERATING SYSTEM

BCA III Semester

Compiled by

Prof.Panchajanyeswari M Achar

SIMS, Mangalore

2015-16

CONTENTS

3.1	Basic Concept	
3.2	Factors affecting the scheduling	
3.3	Preemptive and non-preemptive scheduling	
3.3.1	Scheduling Criteria	
3.4	FCFS , SJF Priority Algorithm	
3.4.1	Problem under the Algorithm	
3.4.2	Multi level Scheduling	
3.5	Assignment Questions 3	
	UNIT II	
Chapter 4	**Process Synchronization**	
4.1	The critical section problem	
4.2	Semaphores (The Classical definition of Wait & Signal)	
4.3	Binary Semaphores	
4.4	Classical Problem of Synchronization	
4.5	Assignment Questions 4	
Chapter 5	**Deadlocks**	
5.1	Deadlock Characterization	
5.2	Necessary and Sufficient Condition for a deadlock state	
5.3	Resource Allocation Graph, Wait-for Graph	
5.4	Methods for handling Deadlocks	
5.5	Dead Lock Prevention, Deadlock Avoidance Methods	
5.6	Recovery from deadlock	
5.7	Assignment Questions 5	

	UNIT III	
Chapter 6	**Memory Management**	
6.1	Logical and Physical Address Space	
6.2	Swapping the Pages	
6.3	Contiguous Allocation (Memory Allocation , Fragmentation)	
6.4	Paging(Basic Method, Hardware Support)	
6.5	Segmentation(basic Method, Hardware)	
6.6	Assignment Questions 6	
Chapter 7	**Virtual Memory Management**	
7.1	Demand Paging	
7.2	Page Replacement	
7.3	Page Replacement Algorithms	
7.4	Allocation of Frames (Equal and Proportional Allocation)	
7.5	Thrashing(concept)	
7.6	Assignment Questions 7	
Chapter 8	**File Systems**	
8.1	File Concept, Access Methods	
8.2	Directory Structures	
8.3	File System Structure	
8.4	Allocation Methods	
8.5	Free Space Management	
8.6	Protection of File System	
8.7	Assignment Questions 8	
	UNIT IV	

Chapter 9	**Unix**	
9.1	An Introduction to Unix	
9.2	Features of Unix	
9.3	Unix System Organization	
9.4	Unix File System	
9.5	Comparison of Unix and windows operating System	
9.6	Assignments Questions 9	
Chapter 10	**Linux**	
10.1	An Introduction, reason for its popularity	
10.2	Linux file system	
10.3	Login and logout	
10.4	Assignment Questions 10	
Chapter 11	**Linux Commands**	
11.1	Command format	
11.2	Directory oriented command, wild card characters	
11.3	File oriented commands	
11.4	File Access Permissions	
11.5	Process oriented commands	
11.6	Background processing	
11.7	Communication oriented commands purpose commands	
11.8	Pipe and Filters related commands, vi editor	
11.9	shell programming	
11.10	System Administration	
11.11	Assignments Questions 11	

	Value added chapter	
Chapter 12	Protection	
12.1	Goals of Protection	
12.2	Domain Structure	
12.3	Revocation of Access Rights	
12.4	Language-Based Protection	

UNIT 1

CHAPTER 1

INTRODUCTION TO OPERATING SYSTEMS

1.1 INTRODUCTION

An operating system is a program that manages the computer hardware. It provides a basis for application programs and acts as an intermediary between the user of a computer and computer hardware. An OS is an important part of any computer system. A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users. The hardware-the central processing unit (CPU), the memory, and the input/output (I/O) devices-provides the basic computing resources for the system. The application programs-such as word processors, spreadsheets, compilers, and web browsers-define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. The components of a computer system are its hardware, software and data. The OS provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government-It does not perform any useful function itself. It simply provides an environment within which other programs can do useful work. An OS can be explored in two view points: User View and System View.

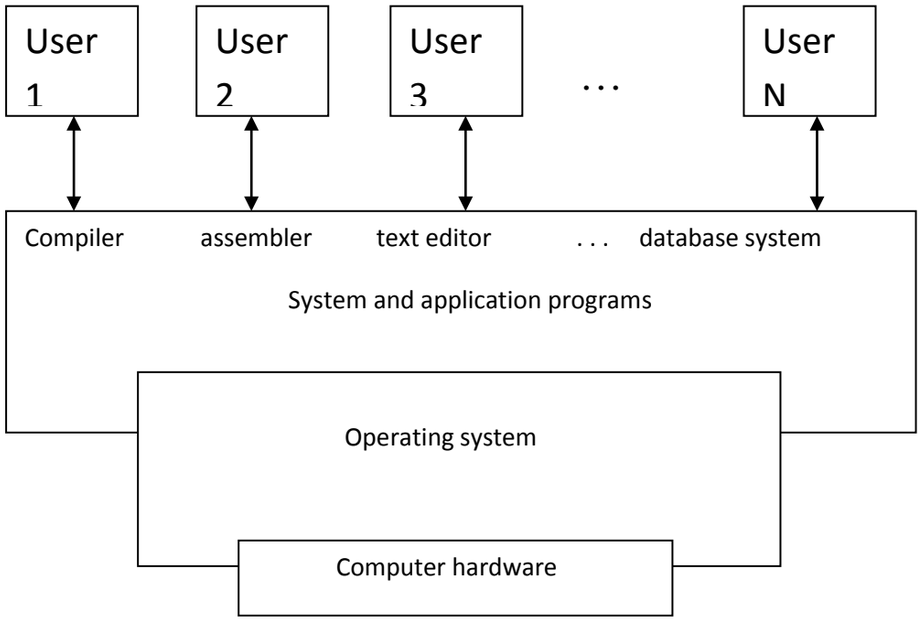


Figure:1.1 An abstract view of the components of a computer system

USER VIEW

The user view of a computer varies by the interfaces being used. In this view where the user sits in front of the PC, performance is important to the user but doesn't matter about the system being idle or waiting for the slow I/O speed of the user.

Here, the OS is designed mostly for ease of use and performance. Some users may be at terminals connected to mainframes. Others may access the same computer through other terminals. The users here share resources and exchange information. The OS in this case is designed to maximize resource utilization - to ensure that CPU time, memory and I/O are used efficiently. Some users may be at workstations connected to networks of other work stations and servers. These users have dedicated resources and at the same time share resources. The OS in this case is designed to compromise between individual usability and resource utilization. Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

SYSTEM VIEW

From computers point of view, an OS is the most intimate with the hardware. We can view OS as a resource allocator. A computer has many resources that may be required to solve a problem - CPU time, memory space and I/O devices. The OS acts as a manager of these resources. It decides on how to allocate these resources to specific programs and users so that it can operate the computer system efficiently. A slightly different view is the need to control the various I/O devices and user programs.

An OS acts as a control program. A control program manages the execution of various user programs to prevent errors and improper use of computer. Thus, the common function of controlling and allocating the resources are brought together into one piece of software called Operating Systems.

1.1 WHAT IS OPERATING SYSTEM?

A program that acts as an intermediary between a user of a computer and the computer hardware. Operating system goals Execute user programs and make solving user problems easier, Make the computer system convenient to use, Use the computer hardware in an efficient manner, An **operating system** (**OS**) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs usually require an operating system to function.

1.2 BATCH PROCESSING SYSTEMS

Early computers were enormous machines run from a console. The common input devices were card readers and tape drives. The common output devices were line printers, tape drives and card punches. The user did not interact directly with the computer. Rather, the user prepared a job which consisted of program, data and some control information and submitted to the computer operator. The job was usually in the form of punch cards. After some later time the output appeared which consisted of the result of the program as well as the dump of the final memory and register contents for debugging.

The OS in early computers was simple. The task of OS was to transfer control automatically from one job to the next. The OS was always resident in the memory. To speed up processing, operators batched together jobs with similar needs and ran them through the computer as a group. Thus, the programmers would leave their program with the operator. The operator would sort the programs into batches and as the computers became available, would run each batch. The output of each batch would be sent to the appropriate programmer. In this execution environment, the CPU is often idle because the speed of mechanical devices is slower than electronic devices. With the advent of disk technology, the OS kept all the jobs on the disk rather than in serial card reader. With direct access to several jobs on the disk, the CPU would perform job scheduling to use resources and perform tasks efficiently.

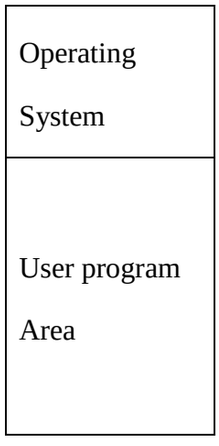


Figure:1.2 Memory layout for simple Batch System.

1.3 MULTI PROGRAMMING SYSTEMS

The most important aspect of job scheduling is the ability to multi program. A single user cannot keep either the CPU or I/O devices busy all the time. Multiprogramming increases CPU utilization by organizing jobs so that CPU always busy with executing one job. The idea is as follows - the OS keeps several jobs in memory simultaneously. The set of jobs is a subset of jobs in the job pool. The as picks and executes one of the jobs in the memory. The job may have to wait for some task such as an I/O operation to complete. In non-multi programmed environment,

the CPU would sit idle. In multi-programmed environment the OS switches to another job. When that job needs to wait, the CPU switches to another job and so on. Eventually the first job finishes waiting and gets the CPU. As long as there is at least one job to execute the CPU never sits idle. Multi-programming is the first instance where the OS has to make decisions for the user.

All the jobs that enter the system are kept in a job pool. A pool is a set of all processes residing on the disk awaiting allocation of main memory. If several jobs are ready to be brought into memory and if there is not enough space for all of them, then the system must choose from among them. This decision making capability is called job scheduling. When the OS selects a job from job pool, it loads that job into memory for execution. If several jobs re ready to run at the same time, the system must choose among them. This decision making capability of as is called CPU scheduling.

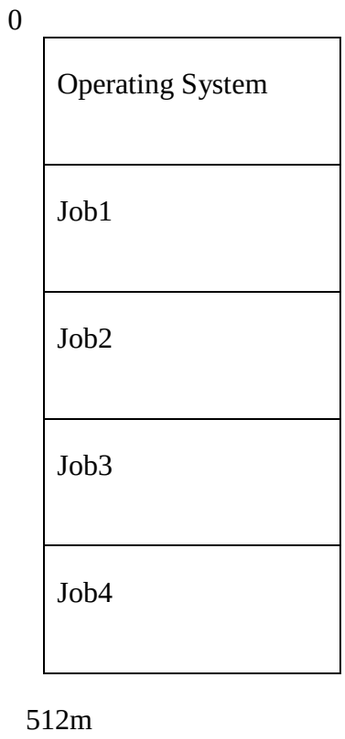


Figure: 1.3 Memory Layout for a multiprogramming System

1.4 TIME SHARING SYSTEM

Multi-programmed and batch systems provide environment for resource utilization but not for user interaction. Time sharing systems is a logical extension of multi-programming. The CPU executes multiple jobs by switching among them, but switches occur so frequently that the users can interact with the program while it is running. An interactive computer system provides direct

communication between the user and the system. The user gives instructions to the system and the system gives the result. The response time should be very short. A time shared as allows many users to share the computer system simultaneously. Since each command in a time shared system tends to be very short only little CPU time is needed for each user. As the system switches from one user to another, the user is given an impression that the entire system is dedicated to his use, even though it is shared by many users. A time shared as uses CPU scheduling and multi programming to provide each user with a small portion of time-shared computer. Each user has at least one separate user program in memory. A program that is loaded in memory and executing is commonly referred to as a process. When a process executes it is typically executed for a short time before it finishes or needs to perform an I/O. Instead of the CPU sitting idle when an I/O is performed, the as will rapidly switch the CPU to the program of some other user.

1.5 REAL TIME SYSTEMS

A real time system is used when rigid time requirements have been placed on the operation of the processor or flow of data. It is used in dedicated applications as a control device. Sensors bring data to a computer. The computer must analyze the data and possibly adjust controls to modify the sensor input. . Examples are medical imaging systems, industrial control systems. . A real time system has well defined time constraints. Processing must be done within the defined constraints or the system will fail. Real time systems can be classified as : a) hard real time systems and b) soft real time systems. A hard real time system guarantees that the critical tasks are completed on time. All delays in the system should be bounded. Delays could be due to retrieval of stored data or time for as to finish any request made of it. A soft real time system is less restrictive. A critical real time tasks gets priority over other tasks and retains that priority until it completes.

1.6 SYSTEM COMPONENTS

The various system components enc**ompassed by the OS are as follows:**

Process management

A process can be thought of a program in execution. E.g. word processing application runs by an individual, a system task of sending output to a printer. A process needs certain resources - CPU time, memory, files and I/O devices to complete a task. These resources are given to it when the process is created or while it is running. A program is a passive entity such as contents of a file stored on a disk. A process is an active entity with a program counter specifying the next instruction to be executed. The execution of a process is sequential. The CPU executes one instruction of a process after another until the process is complete. Thus, a system consists of a collection of processes that include as processes and user processes. The OS is responsible for the following activities with respect to process management.

- Creating and deleting user and system processes

- Suspending and resuming processes
- Providing mechanism for process synchronization
- Providing mechanism for process communication
- Providing mechanism for deadlock handling

MAIN MEMORY MANAGEMENT

Main memory is a large array of words/ bytes. Each word has its own address. It is a repository of quickly accessible data shared by CPU and I/O devices. The CPU reads the instruction from main memory during instruction fetch cycle and reads and writes data from main memory during the data-fetch cycle. The OS is responsible for the following main memory activities Keeping track of which part of main memory is in use by whom Deciding which process to be loaded into main memory when memory space becomes available Allocating/de-allocating memory space as needed

FILE MANAGEMENT

File management is one of the visible components of OS. Computers store information on several different types of physical media – magnetic tape, magnetic disk and optical disks. Each of these media has its own characteristics and physical organization. A file is a collection of related information defined by its creator. Commonly file represents programs or data. Data files may be alphabetic or alpha numeric. A file may be free-form or may be formatted rigidly. A file consists of a sequence of bits, bytes, lines or records whose meaning is defined by its creators

Files are organized into directories for ease of use. When multiple users have access to files, we may have to control by whom and in what ways a file may be accessed. The OS is responsible for the following file management activities

- Creating and deleting files/directories
- Supporting primitives for manipulating files/ directories
- Mapping files on secondary storage
- Backing files on stable storage media

I/O SYSTEM MANAGEMENT

The peculiarities of I/O devices are hidden from the bulk of OS by I/O subsystem. . The I/O subsystem consists of

- Memory management component that includes buffering, caching and spooling

- General device driver interface
- Drivers for specific hardware devices

Secondary Storage Management

Secondary storage is used to store large volumes of data. Most of the programs including compilers, editors - are stored on the disk until they are loaded into memory. Then, they use the disk both as source and destination for their processing. The OS is responsible for the following activities in connection with disk management

- Free space management
- Storage allocation
- Disk scheduling

Networking

A distributed system is a collection of processors that do not share memory or peripheral devices. Each processor has its own local memory and processors communicate with one another through communication lines such as high speed buses or networks. A distributed system collects physically separate, possibly heterogeneous systems into a single coherent system providing the user with access to the various resources it maintains. The File Transfer Protocol and Network File System eliminate the need for the user to log in before he is allowed to use a remote resource. HTTP is used in communication between the web server and the web browser. A web browser needs to send only a request for the information to the remote machines' web server and the information is returned.

Protection

Protection is any mechanism for controlling the access of programs, processes or users to the resources defined by the computer. Protection can improve reliability by detecting latent errors at interfaces between the component sub systems. A protection oriented sub system provides means to distinguish between authorized and unauthorized usage.

Command Line Interpreter

Command interpreter is an interface between the user and OS. Many commands are given to as by control statements. When a new job is started in a batch or when a user logs into time shared system, a program that reads and interprets control statements are execute automatically. This program is called command line interpreter or control card interpreter and is often known as shell. Its function is simple - to get the next command statement and execute it.

1.7 OPERATING SYSTEM SERVICES

An as provides an environment for the execution of program. It provides certain services to the program and the user of the program. The services are provided for the convenience of the programmer to make programming task easier. The services provided by the as are

Program execution: The system must be able to load a program into memory and to run the program. The program must be able to end its execution either normally or abnormally (by indicating an error)

I/O operations: A running program may require I/O. This I/O may be a file or an I/O device. Users cannot control I/O devices directly. Hence, an OS must provide a means for I/O operation.

File system manipulation: The OS must provide means to manage and work with file.

Communication: One process needs to exchange information with another process. This could take place between the processes within the same computer or between different computers tied through a network. Communication can be implemented by shared memory or by message passing, in which packets of information are moved between processes by OS

Error detection: Errors may occur in CPU, memory hardware, I/O devices or user programs. For each type of error, the OS should take appropriate action to ensure correct and consistent computing.

Assignment 1:

1. List the advantages of time sharing systems over multiprogramming system?
2. An operating system is a control program. Justify your answer?
3. Write a note on real time operating system?
4. Explain any five services of an OS?
5. Explain any five major activities of an operating system regard to process management?
6. What is operating system? Explain any two?
7. List the advantages of time shared systems over multi programmed systems
8. Explain Time sharing operating system.
9. Write a note on multiprogramming system.
10. Explain about all operating system components?

CHAPTER 2

PROCESSES

2.1 THE PROCESS CONCEPT

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. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a data section, which contains global variables. We emphasize that a program by itself is not a process; a program is a passive entity, such as the contents of a file 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.

Process State

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.

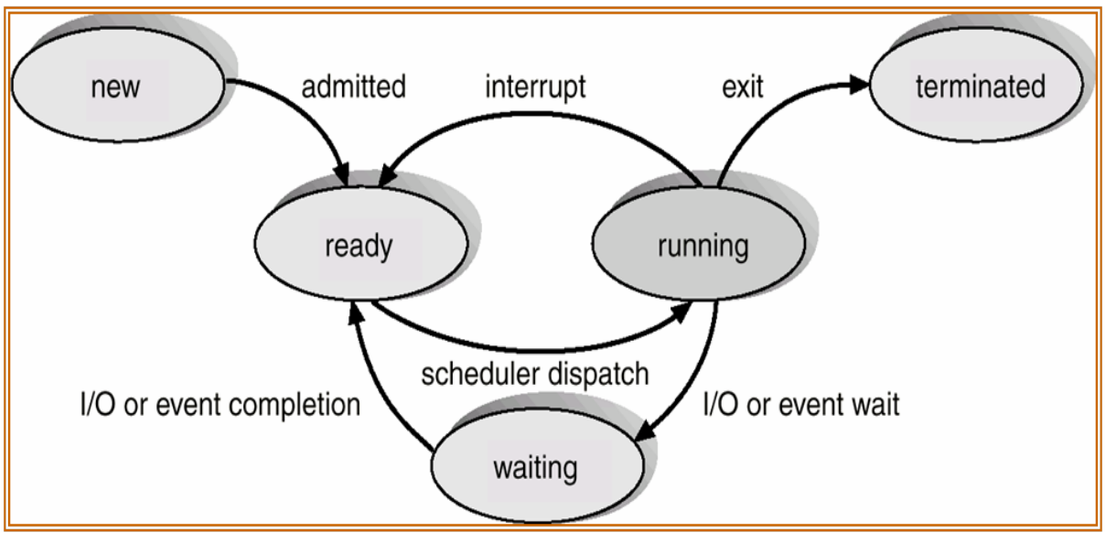


Figure: 2.1 Diagram of Process State

Process Control Block

Each process is represented in the operating system by a process control block (PCB)-also called a task control block.

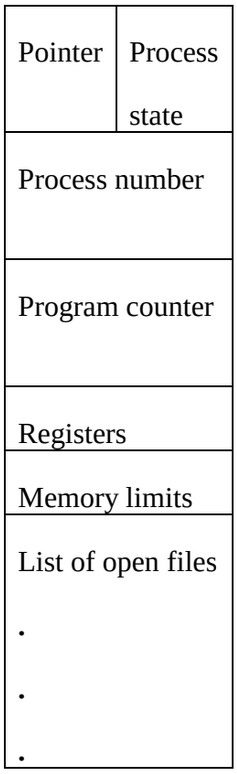


Figure:2.2 Process control block (PCB)

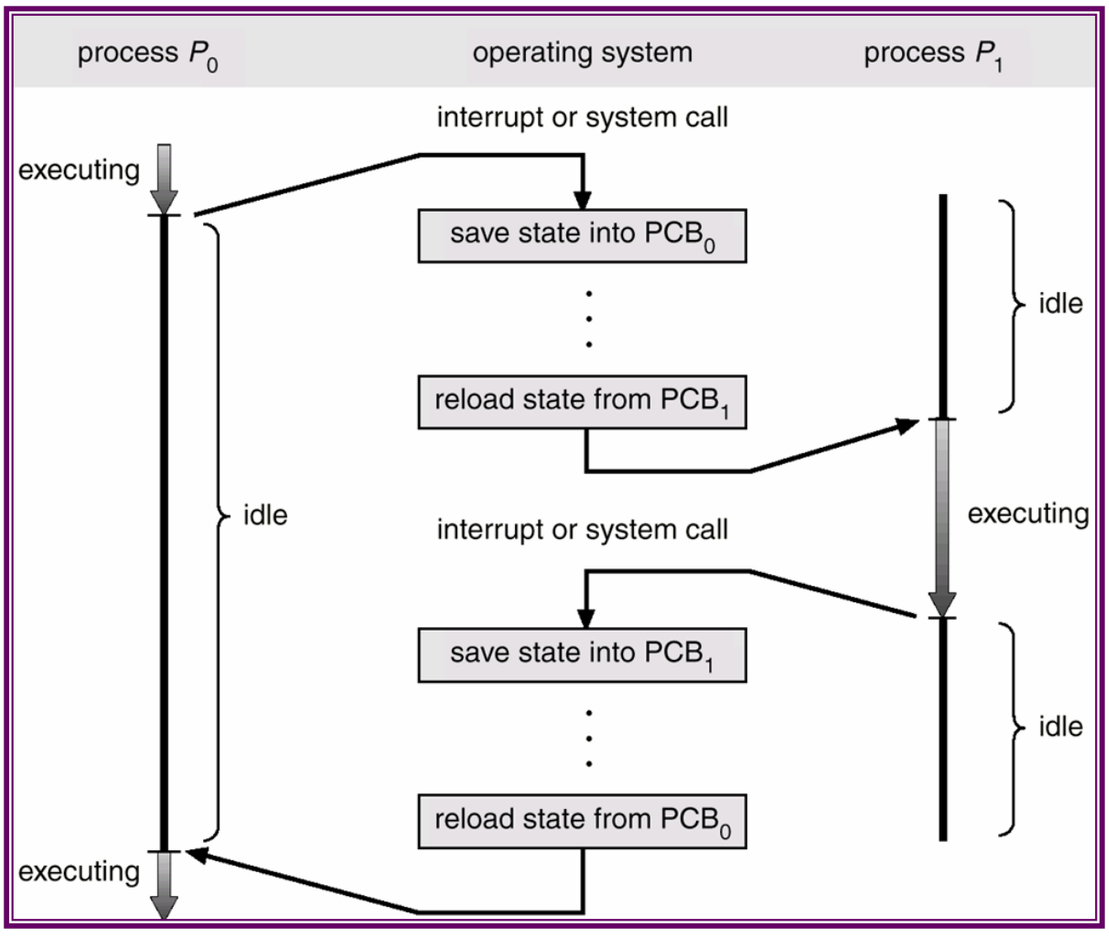


Figure: 2.3 Diagram showing CPU switch from process to process

2.2 PROCESS SCHEDULING

The objective of multiprogramming is to have some process running at all times, so as 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. A uniprocessor system can have only one running process. If more processes exist, the rest must wait until the CPU is free and can be rescheduled.

Scheduling Queues

As processes enter the system, they are put into a job queue. This queue 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. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

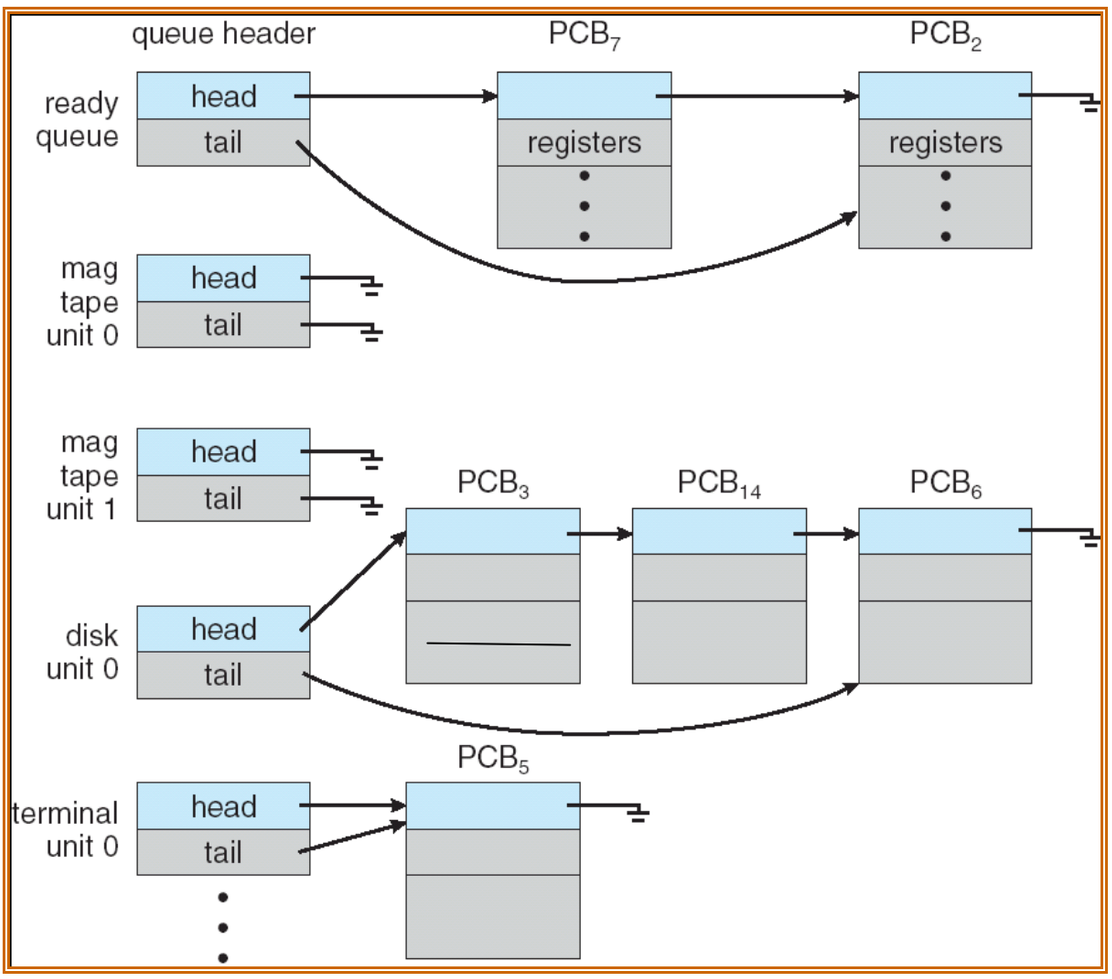


Figure: 2.4The ready queue and various device queues

The operating system also has 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. In the case of an I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since the system has many processes, 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.

A common representation of process scheduling is a queuing diagram. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.

The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

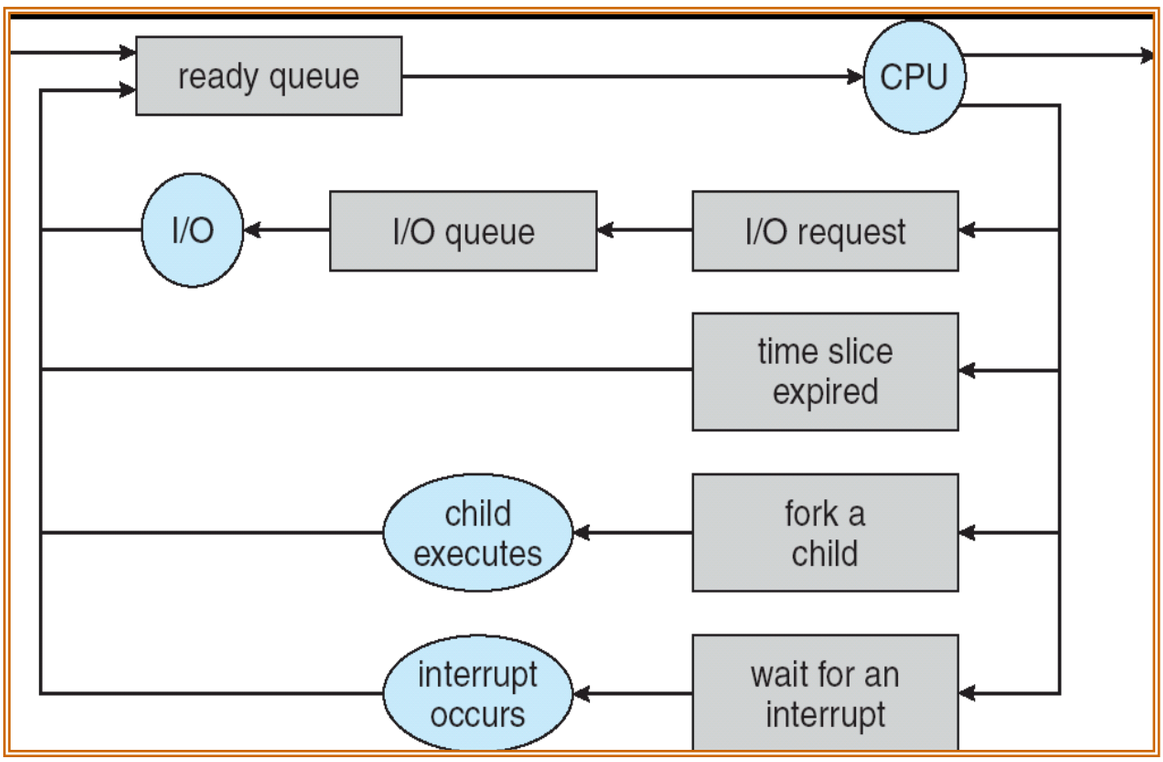


Figure:2.5 Queuing-diagram representation of process scheduling

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

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

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

2.3 TYPES OF SCHEDULERS

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

In a batch system, often more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept fur later execution. The long-term scheduler, or Job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them.

The primary distinction between these two schedulers is the frequency of their execution. 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 brief time between executions, the short-term scheduler must be fast. If it takes 10milliseconds to decide to execute a process for 100milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (or wasted) simply for scheduling the work.

The long-term scheduler, on the other hand, executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the degree of multiprogramming-the number of processes in memory. The long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An I/O-bound process spends more of its time doing I/O than it spends doing computations. A CPU-bound process, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses. The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes. Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler removes processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler.

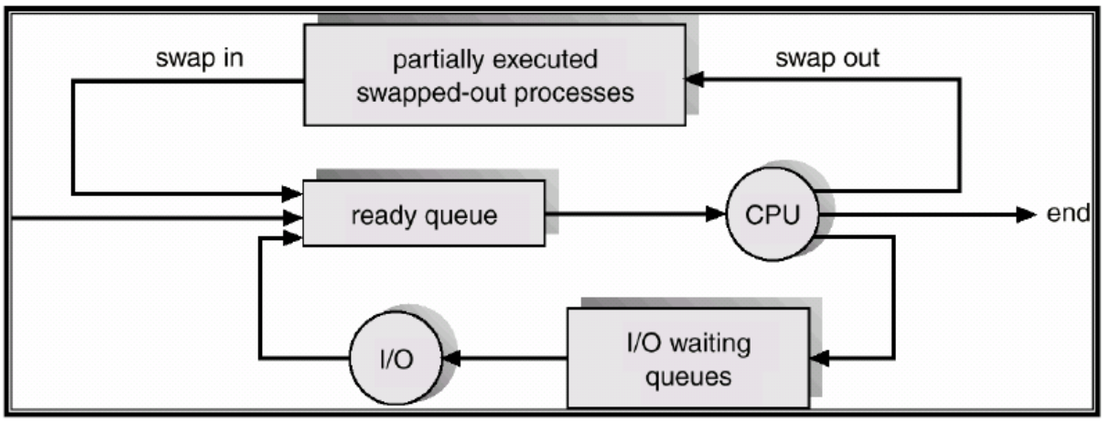


Figure:2.6 Addition of medium term scheduling to the queuing diagram

Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. The context of a process is represented in the PCB of a process. It includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the kernel 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. Typical speeds range from 1 to 1000microseconds. Context-switch times are highly dependent on hardware support.

2.4 CO-OPERATING PROCESSES

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Clearly, any process that does not share any data (temporary or persistent) with any other process is independent. On the other hand, a process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.We may want to provide an environment that allows process cooperation for several reasons:

Information sharing: Since several users may be interested in the same piece of information. We must provide an environment to allow concurrent access to these types of resources.

Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements.

Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Convenience: Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.

2.5 THREADS

2.5.1 Threads concepts

A thread, sometimes called a lightweight process (LWP), is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time.

Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network. A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

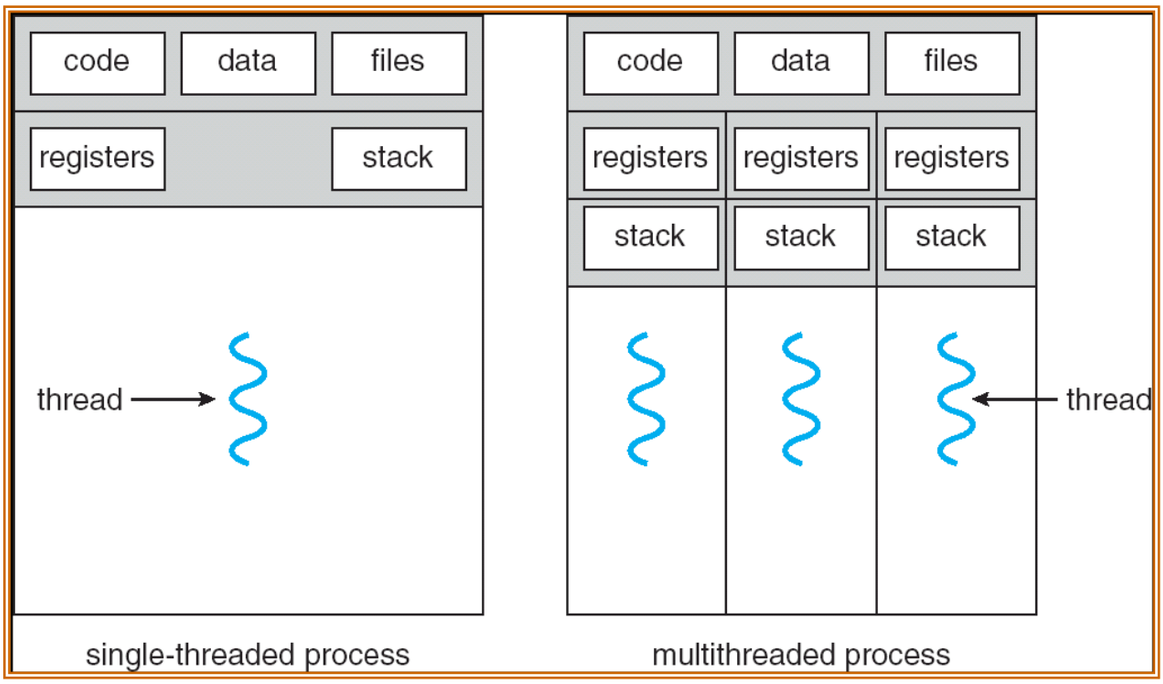


Figure: 2.7 Single and multithread process

2.5.2 SINGLE AND MULTIPLE THREADS BENEFITS

Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation thereby increasing responsiveness to the user.

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

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

Utilization of multiprocessor architectures: the benefits of multithreading can be greatly increased in a multithreading architecture, where each thread may be running in parallel on a processor.

Assignment-2

1. Define process? Explain different states of process with the help of a diagram?
2. Describe the difference between short-term, medium-term and long-term schedulers.
3. Write a note on co-operating processes.
4. With a neat diagram explain process scheduling using a queueing diagram?
5. What is scheduler? Explain different types of schedulers?
6. Compare and contrast thread and process?
7. Write a short note on thread?
8. Write a short note on process control block
9. Write a short note on scheduling queues
10. Write a short note on context switch

CHAPTER 3

CPU SCHEDULING

3.1 BASIC CONCEPT

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes must wait until the CPU is free and can be rescheduled. The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

3.2 FACTORS AFFECTING THE SCHEDULING

CPU-I/O Burst Cycle: 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, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst

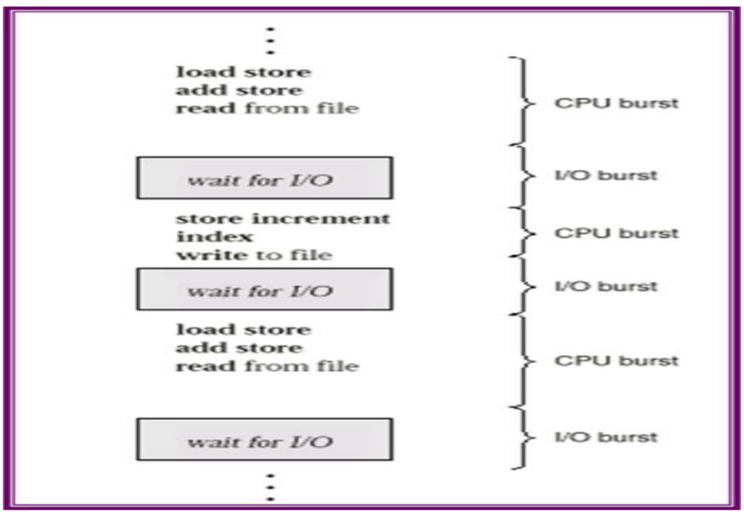


Figure:3.1 Alternating sequence of CPU and I/O bursts

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. The ready queue is not necessarily a first-in, first-out (FIFO) queue. A ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, completion of I/O)

4. When process terminates.

3.3 PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING

Scheduling can be of two types:

a) Non-preemptive scheduling: In this scheme, once a CPU is allocates to process, the process keeps the CPU until it releases the CPU either by terminating or by switching to waiting state. Circumstances 1 and 4 come under this approach.

b) Preemptive scheduling: In this scheme, once a CPU is allocates to process, the process can be forcibly be taken away from the CPU can be allocated to the new process. Circumstances 2 and 3 come under this approach.

Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

3.3.1 SCHEDULING CRITERIA

Different CPU-scheduling algorithms have different properties and 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. The criteria include the following:

- CPU utilization: We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100percent.
- Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput.
- Turnaround time: The interval from the time of submission of a process to the time of completion 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 amount of time that a process spends waiting in the ready queue is called waiting time. Waiting time is the sum of the periods spent waiting in the ready queue.
- Response time: The time interval between submission of a request and the first response is called Response time. It is the amount of time it takes to start responding, but not the time that it takes to output that response.

We want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, in some circumstances we want to optimize the minimum or maximum values, rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

3.4 SCHEDULING ALGORITHMS

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

3.4.1 FIRST-COME, FIRST-SERVED SCHEDULING (FCFS)

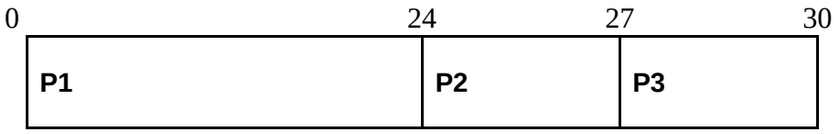
By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm). With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process

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

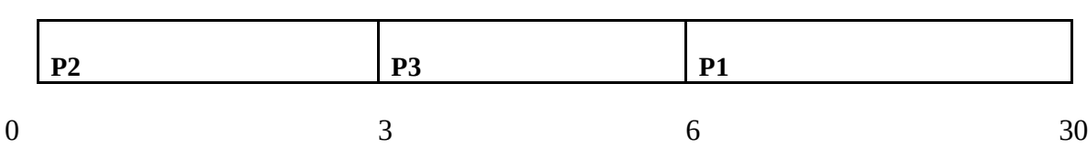
Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart:



The waiting time is 0 milliseconds for process P_1, 24milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 +27)/3 = 17milliseconds. The turn around time is 24 milliseconds for process P_1, 27milliseconds for process P_2, and 30 milliseconds for process P_3. Thus, the average turn around time is (24 + 27 +30)/3 = 27milliseconds.

If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. The average turn-around time is (3 + 6 + 30)/3 = 13. This reduction is substantial. Thus, the average waiting time under a FCFS policy is generally not minimal, and may vary substantially if the process CPU-burst times vary greatly.

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/O. The FCFS algorithm is particularly troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

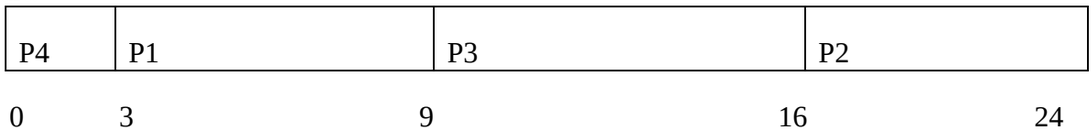
3.4.2 SHORTEST-JOB-FIRST SCHEDULING

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that a more appropriate term would be the shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks refer to this type of scheduling discipline as SJF.

As an example, consider the following set of processes, with the length of the burst time given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

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



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. The turn around time is 1milliseconds for process P_1, 5 milliseconds for process P_2, 10 milliseconds for process P_3, and 16 milliseconds for process P_4. Thus, the average waiting time is (1 + 5 + 10 + 16)/4 = 8 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time could be 10.25 milliseconds.

The SJF scheduling algorithm is probably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases 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. For long-term (or job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value.

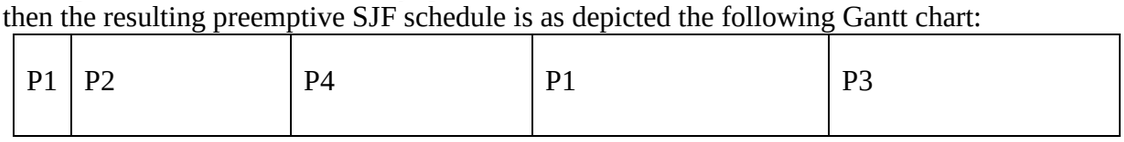
The SJF algorithm may be either preemptive or non preemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new

process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas 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 time 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 times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted the following Gantt chart:



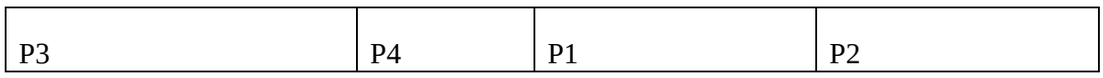
0 1 5 10 17 26

Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4= 6.5 milliseconds. A non preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

The turn around time for process is calculated as (The time at which process completed its CPU burst). The Average Turn around time is (17+5+26+10)/4=14.5 milliseconds.

Example: Non-Pre emptive SJF (varied arrival times)

Process	Arrival Time	Burst Time
P_1	2	6
P_2	1	8
P_3	0	7
P_4	3	3



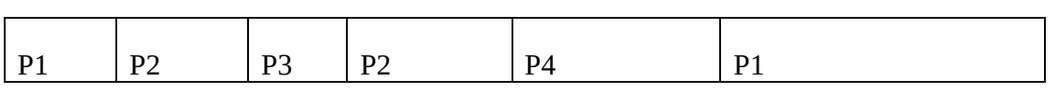
0 7 10 16 24

Average waiting time = ((10–2)+(16–1)+(0–0)+(7–3))/4 = (8 + 15 + 0 + 4)/4 = 6.75

Average turn-around time = ((16–2)+(24–1)+(7-0)+(10–3))/4 =(14 + 23 + 7 + 7)/4 = 12.75

Example: Pre emptive SJF (Shortest-remaining-time-first)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



0 2 4 5 7 11 16

Average waiting time = ([(0 – 0) + (11 - 2)] + [(2 – 2) + (5 – 4)] + (4 - 4) + (7 – 5))/4
=(9+1+0+2)/4
= 3

Average turn-around time = (16 + 7 + 5 + 11)/4 = 9.75

3.4.3 Priority Scheduling

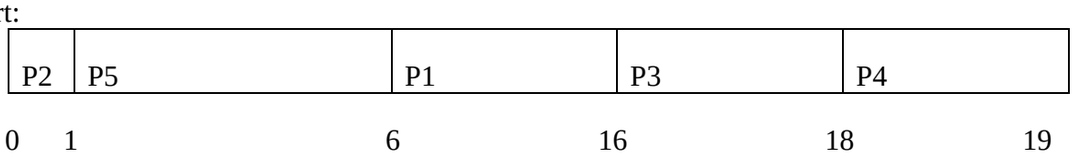
The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where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

Priorities are generally 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 we use low numbers to represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2…..P_5 with the length of the CPU-burst time 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 is 8.2 milliseconds (i.e. (6+0+16+18+1)/5). Average Turn around time is 12 milliseconds (i.e.(16+1+18+19+6)/5).

Priority scheduling can be either preemptive or non preemptive. When a process arrives at the ready queue, its priority is compared with the priority of 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 lacking the CPU can be considered blocked-waiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. 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.

3.4.4 ROUND-ROBIN SCHEDULING

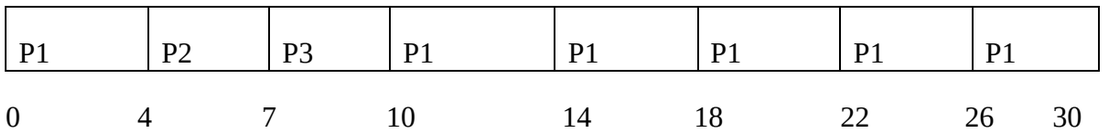
The round-robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice),is defined. A time quantum is generally from 10 to 100 milliseconds. 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 itself 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, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

If we use a time quantum of 4 milliseconds, then process PI 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 P_2 Since process P_2 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 P_1 for an additional time quantum. The resulting RR schedule is



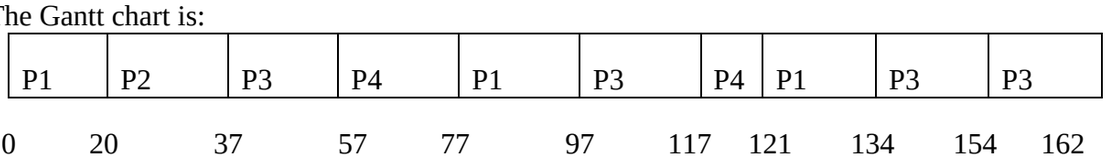
The average waiting time is 17/3 = 5.66milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive. The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), theRR policy is the same as the FCFS policy.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

The Gantt chart is:



Typically, higher average turnaround than SJF, but better *response time*

Average waiting time = ([(0 – 0) + (77 - 20) + (121 – 97)] + (20 – 0) + [(37 – 0) + (97 - 57) + (134–117)]+[(57–0)+(117–77)])/4 = (0 + 57 + 24) + 20 + (37 + 40 + 17) + (57 + 40)) / 4
=(81+20+94+97)/4
= 292 / 4 = 73

Average turn-around time = 134 + 37 + 162 + 121) / 4 = 113.5

3.4.5 MULTILEVEL QUEUE SCHEDULING

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between

foreground (or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority (or externally defined) over background processes.

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

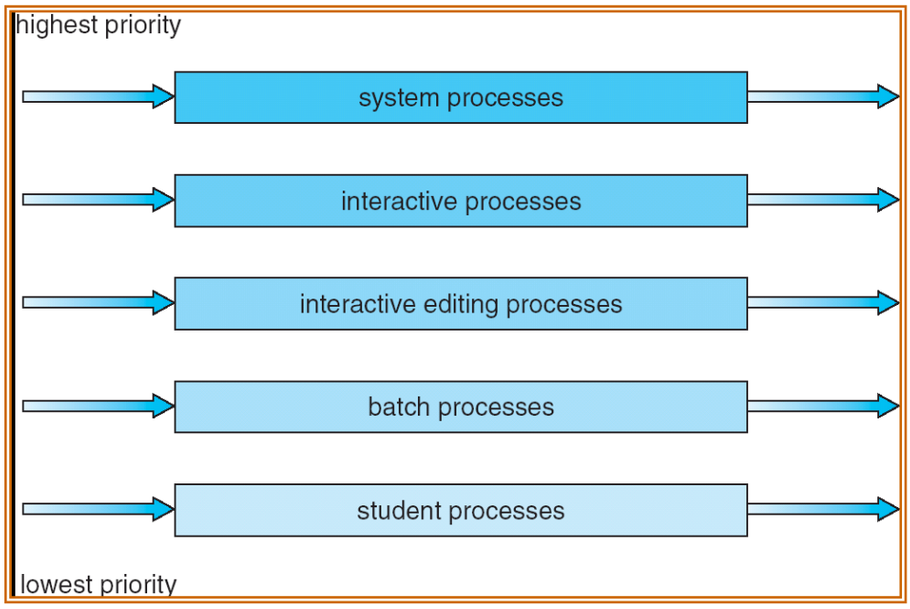


Figure: 3.2 Multilevel Queue Scheduling

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Solaris 2 uses a form of this algorithm.

3.4.2 MULTILEVEL FEEDBACK QUEUE SCHEDULING

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. Similarly, a process that waits too long in a lower priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue O. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process that arrives for queue a will, in turn, preempt a process in queue 1

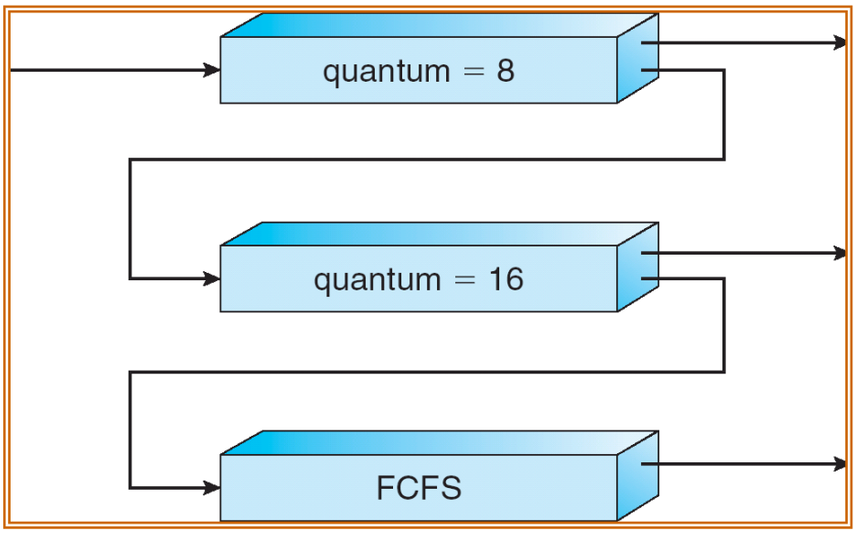


Figure:3.3 Multilevel feed back queues

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higher priority queue

• The method used to determine when to demote a process to a lower priority queue

• The method used to determine which queue a process will enter when that process needs service

Multiple-Processor Scheduling

In a homogeneous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor; otherwise the device would not be available.

When several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, however, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this

situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. If we have multiple processors trying to access and update a common data structure, each processor must be programmed very carefully. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

This asymmetric multiprocessing is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, allows the need for data sharing. However, it is also not as efficient. I/O bound processes may bottleneck on the one CPU that is performing all of the operations.

Algorithm Evaluation

To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as

Maximize CPU utilization under the constraint that the maximum response time is 1 second.

Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Different techniques for algorithm evolution are

- Deterministic modelling
- Queueing models
- Simulation
- Implementation

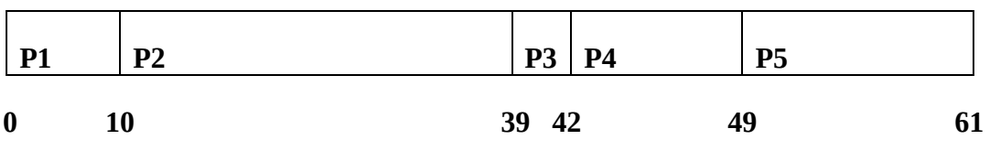
Deterministic Modeling

One major class of evaluation methods is called analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload. One type of analytic evaluation is deterministic modeling. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

For example, assume that we have the workload shown. All five processes arrive at time 0, in the order given, with the length of the CPU-burst time given in milliseconds.

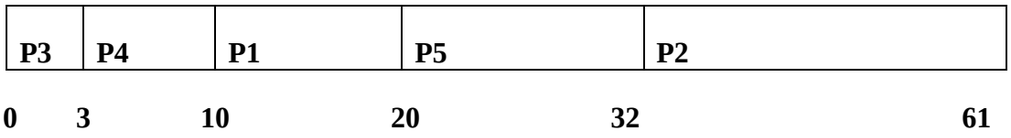
Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. We have to know which algorithm would give the minimum average waiting time. For the FCFS algorithm, we would execute the processes as



The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for process P_3, 42 milliseconds for process P_4, and 49 milliseconds for process P_5. Thus, the average waiting time is (0 + 10 + 39 + 42+ 49)/5 = 28 milliseconds.

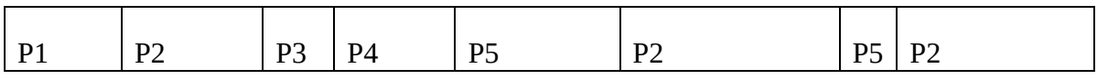
With non preemptive SJF scheduling, we execute the processes as



0 3 10 20 32 61

The waiting time is 10 milliseconds for process P_1, 32 milliseconds for process P_2, 0milliseconds for process P_3, 3 milliseconds for process P_4, and 20 milliseconds for process Ps. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm we execute the process as



0 10 20 23 30 40 50 52 61

The waiting time is 0 milliseconds for process P_1, 32 milliseconds for process P_2, 20 milliseconds for process P_3, 23 milliseconds for process P_4, and 40 milliseconds for process P_5. Thus, the average waiting time is (0 + 32 + 20+ 23 + 40)/5 = 23milliseconds.

We see that, in this case, the SJF policy results in less than one-half the average waiting time obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives exact numbers, allowing the algorithms to be compared. However, it requires exact numbers for input, and its answers apply to only those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing example

Queueing Models

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on.

This area of study is called queueing-network analysis.

As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). Then, we expect that during the time W that a process waits, λ

X W new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation is known as Little's formula. Little's formula is particularly useful because it is valid for any scheduling algorithm and arrival distribution. Queueing models are often only approximations of real systems. The classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Arrival and service distributions are often defined in unrealistic, but mathematically tractable ways.

Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Simulations involve programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

Simulations can be expensive, however, often requiring hours of compute time. A more detailed simulation provides more accurate results, but also requires more computer time. In addition, trace tapes can require large amount of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

Implementation

The completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

The major difficulty with this approach is high cost. The algorithm has to be coded and the operating system has to be modified to support it. The users must tolerate a constantly changing operating system that greatly affects job completion time. Another difficulty is that the environment in which the algorithm is used will change. New programs will be written and new kinds of problems with be handled

The environment will change as a result of performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive

processes are given priority over non-interactive processes, then users may switch to interactive use.

The most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. For example, a workstation that performs high-end graphical applications may have scheduling needs different from those of a web server or file server. Another approach is to use APIs (POSIX, WinAPI, Java) that modify the priority of a process or thread. The downfall of this approach is that performance tuning a specific system or application most often does not result in improved performance in more general situations

Assignment-3

1. Consider the following set of processes that arrive at time 0 with the length of the CPU-burst time given in milliseconds.

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Draw Gantt chart and find average waiting time using SJF scheduling

2. What are the criteria for CPU scheduling? Explain in detail?
3. Discuss about FCFS and SJF CPU scheduling policies with examples and also compare the same.
4. Explain multilevel feedback scheduling.
5. Explain priority scheduling algorithm with one example?
6. Consider the following set of process, with the length of CPU-burst time given in milliseconds

Process	Arrival Time	Burst Time
P_1	2	6
P_2	1	8
P_3	0	7

P_4	3	3

Draw Gantt chart and find average waiting time using SJF scheduling.

7. Explain round robin scheduling algorithm with an example?
8. Explain multilevel feed back queue scheduling?
9. Explain two types of scheduling?
10. Explain FCFS scheduling with an example?
11. Explain SJF scheduling with an example?

UNIT-II

CHAPTER 4

PROCESS SYNCHRONIZATION

INTRODUCTION

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files. Concurrent access to shared data may result in data inconsistency. There are different mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

4.1 CRITICAL SECTION PROBLEM

Consider a system consisting of n processes $\{P_0, P_1,\ldots, P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

```
do {

    Entry section

    Critical section

    Exit section

    Remainder section

} while (1);
```

Figure:4.1 General structure of a typical process Pi

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

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

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

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

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n process When presenting an algorithm, for critical section solution we define only the variables used for synchronization purposes, and describe only a typical process P_i. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j==1- i.

Algorithm 1

Our first approach is to let the processes share a common integer variable turn initialized to 0 (or 1). If turn== i, then process P_i is allowed to execute in its critical section. The structure of process Pi is shown below

```
do{

While(turn !=i);

Critical section

Turn =j;
```

Remainder section

} while (1) ;

Figure:4.2 The structure of process P_i in algorithm 1.

This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and P_1 is ready to enter its critical section, P_1 cannot do so, even though P_0 may be in its remainder section.

Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable turn with the following array:

boolean flag[2];

The elements of the array are initialized to false. If flag [i] is true, this value Indicates that P_i is ready to enter the critical section. Then P_i checks to verify that process P_j is not also ready to enter its critical section. If P_j were ready, then P_i would wait until P_j had indicated that it no longer needed to be in the critical section (that is, until flag[j] was false). At this point, Pi would enter the critical section.

On exiting the critical section, P_i would set flag [i] to be false, allowing the other process (if it is waiting) to enter its critical section. In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

T0: p_0 sets flag[0]=true

T1:p_1 sets flag[1]=true

Now P_0 and P_1 are looping forever in their respective while statements.

do{

Flag[1] = true

While(flag [1]);

Critical section

Flag[i] = false

Remainder Section

}while(1);

Figure: 4.3 The structure of process Pi in algorithm 2

Note that switching the order of the instructions for setting flag [i], and testing the value of a flag [j] ,will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section at the same time, violating the mutual-exclusion requirement.

Algorithm 3

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables

boolean flag[2];

int trun;

Initially flag [0] =flag [1] = false, and the value of turn is immaterial (but is either 0 or 1). The structure of process P_i is shown below.

Do{

flag [i]=true;

turn = j;

while (flag [i] && turn==j);

Critical section

flag[i] = false

Remainder section

} while (1);

Figure: 4.4 The structure of process P_i in algorithm 3

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, there by asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same

time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that

1. Mutual exclusion is preserved,

2. The progress requirement is satisfied,

3. The bounded-waiting requirement is met.

To prove property 1, we note that each Pi enters its critical section only if either flag [j] == false or turn==i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] ==flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both. Hence, one of the processes-say P_j-must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn == j"). However, since, at that time, flag[j] == true, and turn == j, and this condition will persist as long as Pj is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] ==true and turn == j; this loop is the only one. If P_j is not ready to enter the critical section, then flag [j] == false and Pi can enter its critical section. If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag [j] to false, allowing P_i to enter its critical section. If P_j resets flag [j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Multiple-Process Solution

This algorithm is used for solving the critical-section problem for n processes. This algorithm is also known as the bakery algorithm, and it is based on a scheduling algorithm commonly used in bakeries, ice-cream, stores, deli counters, motor-vehicle registries, and other locations where order must be made out of chaos. On entering the store, each customer receives a number. The customer with the lowest number is served next. Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) do not receive the same number. In the case of a tie, the process with the lowest name is served first. That is, if P_i and P_j receive the same number and if i < j, then P_i is served first. Since process names are unique and totally ordered, our algorithm is completely deterministic. The common data structures are

boolean choosing [n];

int number [n];

Initially, these data structures are initialized to false and 0, respectively. For convenience, we define the following notation:

- (a, b) < (c, d) if a < c or if a==c and b < d
- Max (a_0…..a_{n-1} is a number , k , such that k > = a; for i = 0 ….b

The structure of process P_i used in the bakery algorithm, is shown below

```
do{
choosing [i] = true ;
number [i] = max (number) [0], number [1] …..number [n-1] + 1;
choosing [i] = false ;
for (j=0: j< n: j++)
{
While (choosing [j]);
While ((number [j] !=0 ) && (number [j , j] < numbers[ i , i]));
}
Critical section

Number [i] = 0 ;

Remainder section
}While (1);
```

Figure: 4.5 The structure of process Pi in the bakery algorithm

To prove that the bakery algorithm is correct, we need first to show that, if Pi is in its critical section and P_k (k != i) has already chosen its number k!= 0, then (number[i] ,i) < (number [k] ,k). It is now simple to show that mutual exclusion is observed. Indeed, consider Pi in its critical

section and P_k trying to enter the P_k critical section. When process P_k executes the second while statement for j ==i, it finds that

- Number [i] !=0
- Number[i] ,i) < (number [k] , k)

Thus, it continues looping in the while statement until P_i leaves the P_i critical section. The progress and bounded-waiting requirements are preserved, and that the algorithm ensures fairness, it is sufficient to observe that the processes enter their critical section on a first-come, first-served basis.

Synchronization Hardware

Synchronization hardware can be used effectively in solving the critical-section problem. The critical-section problem could be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

This solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically-that is, as one uninterruptible unit. The Test And Set instruction can be defined as shown below

```
Boolean Test And Set(boolean  &targrt){

Boolean rv  = taget;

Target = true;

Return rv;

}
```

Figure 4.6: The definition of Test And Set Instruction

The important characteristic is that this instruction is executed atomically. Thus, if two Test And Set instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

If the machine supports the Test And Set instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i is shown below

```
do{

While (Test And Set(lock));

Critical section

Lock = false

Remainder section

}While(1);
```

Figure 4.7: Mutual Exclusion implementation with Test And Set

The Swap instruction operates on the contents of two words; like the Test And Set instruction, it is executed atomically.

```
Void swap (boolean &a, boolean &b){

Boolean  temp = a;

a= b;

b= temp ;

}
```

Figure 4.8: The definition of Swap Instruction

If the machine supports the Swap instruction, then mutual exclusion can be provided. A global Boolean variable lock is declared and is initialized to false. In addition, each process also has a local Boolean variable key.

```
do{

Key = true ;

While (key = = true)
```

```
Swap (lock, key);

Critical section

Lock = false;

Remainder section

}while(1);
```

Figure 4.9: Mutual-exclusion implementation with the Swap Instruction

These algorithms do not satisfy the bounded-waiting requirement. Another algorithm is developed that satisfies all the critical-section requirements. The common data structures are

Boolean waiting[n];

Boolean lock;

These data structures are initialized to false.

```
do{
  waiting [i] = true;
  key = true;
  while (waiting (i) && key)
  key = Test AndSet(lock);
  waiting [i] = false;
  critical section
  j = (i+ 1) % n;
  while (( j ! = i) && ! waiting [j] )
```

```
j = (j + 1) %  n;
if (j == i)
lock  = false ;
else
waiting [j] = false
remainder  Section
} while (1);
```

Figure 4.10: Bounded waiting mutual exclusion with Test And Test

To prove that the mutual exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the Test And Set is executed. The first process to execute the Test And Set will find key == false; all others must wait. The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual exclusion requirement.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false, or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n - 1,0, ... , i-I). It designates the first process in this ordering that is in the entry section (waiting [j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns. Unfortunately for hardware designers, implementing atomic Test And Set instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

4.2 Semaphores

The generalized solution for complex critical section problems can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard wait and signal. These operations were originally termed P (for wait; from The Dutch proberen, to test) and V (for signal; from verhogen, to increment). The classical definition of wait in pseudocode is

```
wait(S) {
while (S ≤ 0)
;/ / no-op
S--;
}
```

The classical definition of signal in pseudocode is

```
signal(S) {
S++;
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S ($S \leq 0$), and its possible modification (S--), must also be executed without interruption.

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting PI and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
S1;
Signal (synch);
In process PI, and the statements
Wait (Synch);
```

S2;

In process P2. Because synch is initialized to 0, P2 will execute S2 only after PI has invoked signal (synch), which is after s1.

```
Do{

Wait (mutex);

Critical section

Signal (mutex);

 Remainder section

}while(1);
```

Figure 4.11: Mutual-exclusion implementation with semaphores.

The main disadvantage of the mutual-exclusion solutions semaphore definition is that they all require busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock (because the process "spins" while waiting for the lock).

To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. The block operation suspends the process that invokes it. The wakeup (P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute wait and signal operations on the same semaphore at the same time.

4.3 Binary Semaphores

There are two types of semaphores. They are counting semaphore and binary semaphore. In counting semaphore its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. The counting semaphore can be implemented using binary semaphores.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
Binary-semaphore s1, s2;
Int c ;
```

Initially S1 = 1, S2 = 0, and the value of integer C is set to the initial value of the counting semaphore S

The wait operation on the counting semaphore S can be implemented as follows:

```
Wait (s1);
C--;
If(c < 0){
Signal (s1);
Wait (s2);
}
Signal (s1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
Wait (s1);
```

```
C++;
If(c  <= 0)
Signal (s2);
else
Signal (s1);
```

4.4 Classic Problems of Synchronization

The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The -empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. We can interpret this as the producer producing full buffers for the consumer, or as the consumer producing empty buffer for the producer.

The Readers- Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers- writers' problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

The Dining-Philosophers Problem

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

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5];

Where all the elements of chopstick are initialized to 1.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.When each philosopher tries to grab her right chopstick, she will be delayed forever.

We present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

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

Assignment-4

1. What is critical section? What are the requirements for a solution to critical section problem?
2. Explain the two process solution to critical section problems?
3. Write a short note on semaphores

4. Write a short note on bounded buffer problem
5. Write a short note on Dining-philosopher problem
6. Write a short note on Readers- Writers Problem
7. Explain binary semaphores
8. Explain n process solutions to critical section Problem

CHAPTER 5

DEADLOCK

INTRODUCTION

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

System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and 1/0 devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances. If a process requests an instance of a resource type, the allocation of an instance of the type will satisfy the request.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3**. Release:** The process releases the resource.

A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an even that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example files, semaphores, and monitors).

To illustrate a deadlock state, we consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

5.1 DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting.

5.2 NECESSARY ANY SUFFICIENT CONDITION FOR A DEADLOCK STATE

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 model 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 and 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 $\{P_0, P_1,\ldots,P_n\}$ of waiting processes must exist such that P0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n 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.

5.3 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 = {P_1, P_2,... Pn}, the set consisting of all the active processes in the system, and R = {R_1, R_2, ..., R_m}, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process Pi is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

Pictorially, we represent each process Pi as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a

dot within the square. Note that a request edge points to only the square R_j, whereas an assignment edge must also designate one of the dots in the square. When process Pi requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.

The resource-allocation graph depicts the following situation

- The sets P R and E
  - P={p1,p2,p3}
  - R={R1,R2,R3,R4}
  - E={P1→ R1, P2→R3,R1→P2, R2→P2, R2→P1,R3→P3}

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

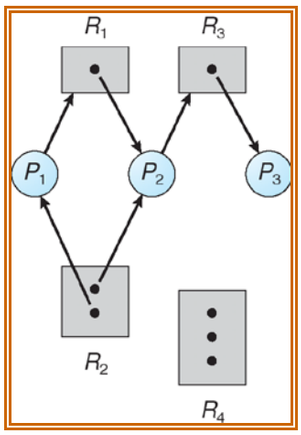


Figure 5.1: Resource allocation graph

Process states:

- Process P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.
- ProcessP_2 is holding an instance of R_1 and R_2, and is waiting for an instance of resource type R_3.
- Process P_3 is holding an instance of R_3

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.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

For example in above resource allocation graph suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge P_3 → R_2 is added to the graph. At this point, two minimal cycles exist in the system:

P1→ R1 → p2 →R3 →p3 → R2 –p1P2→ R3 → p3—R2 → p2

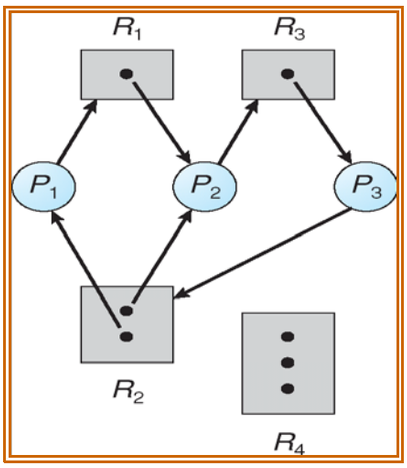


Figure 5.2 : Resource-allocation graph with a deadlock

Now consider another resource-allocation graph as follows

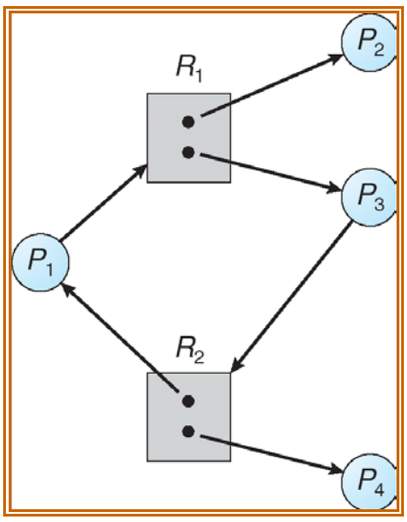


Figure 5.3 : Resource-allocation graph with a cycle but no deadlock.

In above graph we have cycle. i.e.

P1→ R1 → P3 → R2 → P1

However there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system mayor may not be in a deadlock state.

5.4 METHODS FOR HANDLING DEADLOCKS

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, and then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock.

If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

In this case, the undetected deadlock will result in the deterioration of the system performance, because resources are being held by processes that cannot run, and because more and more processes as they make requests for resources, enter a deadlock state. Eventually, the system will stop functioning and will need to be restarted manually.

5.5 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.

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, on the other hand, 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 atterp.pt 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: Some resources are intrinsically non sharable.

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 the process 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 tape drive to a disk file, sorts the disk 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 tape 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 tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape 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.

These protocols have two main disadvantages. First, resource utilization may be low, since many of the 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.

No Preemption

The third necessary condition 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 resources for which the 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 available, 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 not either available or 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.

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 impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let R = $\{R_1, R_2, ..., R_m\}$ 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. Formally, we define a one-to-one function F: R → 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

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, say R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. 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. 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 R_j, it has released any resources R_i such that $F(R_i) \geq F(R_j)$. If these two protocols are used, then the circular-wait condition cannot hold.

deadlock avoidance

Possible side effects of preventing deadlocks by deadlock prevention method however, are low device utilization and reduced system throughput. An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This 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.

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. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_2,...,P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j 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 P_i terminates, P_{i+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 deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

Maximum Needs Current Needs

P0	10	5
P2	4	2
P3	9	2

At time to, the system is in a safe state. The sequence <P_1, P_0, P_2> satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will have all the 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time h, process P_2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are un available, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

Our mistake was in granting the request from process P_2 for 1 more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

We can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

5.5.1 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined can be used for deadlock avoidance.

In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by Pi, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of process in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found; then the allocation will put the system in an unsafe state. Therefore, process P_i will have to wait for its requests to be satisfied.

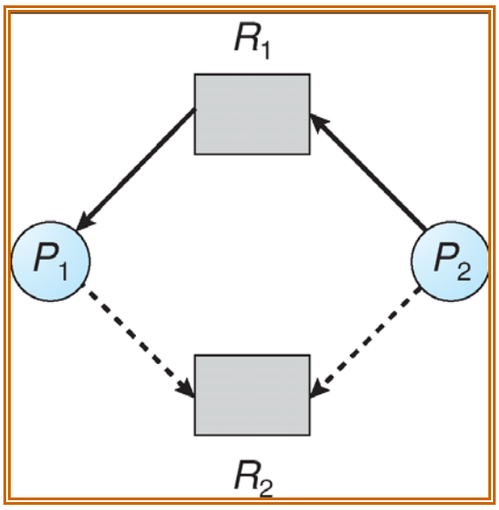


Figure 5.4 : Resource-allocation graph for deadlock avoidance.

To illustrate this algorithm, we consider the resource-allocation graph of above. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph as shown below.

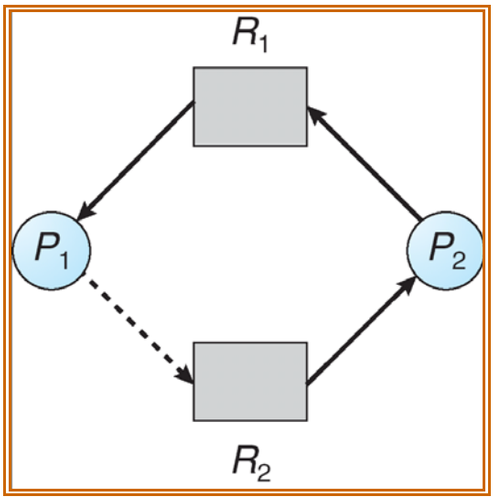


Figure5.6: An unsafe state in resource allocation graph

A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The Banker's algorithm is applicable to system with multiple instances of each resource type, but is less efficient than the resource-allocation graph scheme. The name was chosen because this algorithm could be used in a banking system to ensure that that bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

- 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 R_i available.
- Max: An n x m matrix defines the maximum demand of each process. If Max[i,j]=k, then process P_i may request at most k instances of resource type R_j.

- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.

- Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process P_i may need k more instances of resource type R_j to complete its task. Note that Need[i,j]= Max[i,j] - Allocation [i,j].

To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and Y are vectors of length n. We say that $X \leq Y$ if and only if Xli] $\leq$ Y[i] for all i = 1, 2, ..., 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 $Y \neq X$.

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 = 1,2, ...,n.

2. Find an i such that both

   a. Finish[i] =false

   b. $Need_i \leq Work$.

   If no such i exists, go to step 4.

3. Work:= Work + $Allocation_i$

   Finish[i] := true

   go to step 2.

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

This algorithm may require an order of m x n2 operations to decide whether state is safe.

Resource-Request Algorithm

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

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i 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 - $Request_i$;

$Allocation_i$: = $Allocation_i$ + $Request_i$;

$Need_i$: = $Need_i$ - $Request_i$;

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

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, 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

We elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. A detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process Pi to release a resource that Pi needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_i$ for

some resource R_q. We present a resource-allocation graph and the corresponding wait-for graph as follows.

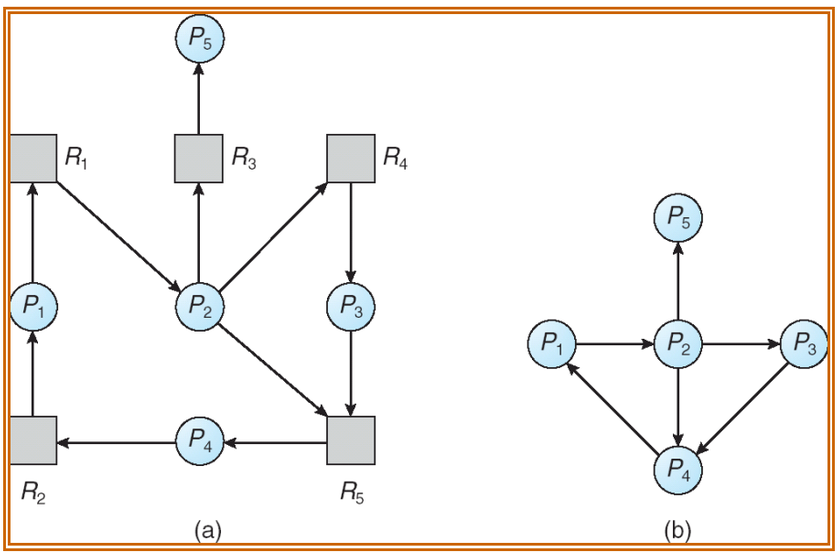


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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 to invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

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. The deadlock-detection is applicable to such a system. The algorithm employs several time-varying data structures.

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

Let X and Y are vectors of length n. We say that $X \leq Y$ if and only if Xli] $\leq$ Y[i] for all i = 1, 2, ..., 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 shall again treat the rows in the matrices Allocation and Request as vectors, and shall refer to them as Allocationi and $Request_i$, respectively. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

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

   Work := Available. For i = I, 2, ..., n, if Allocationi ≠0, then Finish[i]:=false

   otherwise, Finish[i] := true.

2. Find an index i such that both

   a. Finish[i]=false.

   b. $Request_i$:= Work.

   If no such i exists, go to step 4.

3. Work:= Work + $Allocation_i$

   Finish[i] := true

   go to step 2.

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

   Finish[i] = false, then process P_i is deadlocked.

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

We reclaim the resources of process P_i (in step 3) as soon as we determine that $Request_i \leq$ work (in step 2b). We know that P_i is currently not involved in a deadlock (since $Request_i \leq$ Work). Thus, we take an optimistic attitude, and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time that the deadlock-detection algorithm is invoked.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking CI, deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a 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 dead lock cycle, but at a great expense; these processes may have computed for long time, and the results of these partial computations must be discarded and probably recomputed later.

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**.**

If the partial termination method is used, then, given a set of deadlock processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. We should abort those processes the termination of which will incur the minimum cost.

1. What the priority of the process is
2. how long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

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

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim: As in process termination, we must determine the order of pre emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
2. Rollback: when we preempt a resource from a process, 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 total roll back. Abort the process and then restart it. However, it is more effective to rollback the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. 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.

Assignment-5

1. Explain resource-allocation graph
2. Explain the methods used to recover from deadlock
3. Explain wait-for graph with an example
4. Explain the method used to recover from deadlock
5. Illustrate the use of wait-for graph in deadlock detection
6. Explain the data structures involved in banker's algorithm?

8. List out and explain necessary and sufficient condition for deadlock
9 . Explain the deadlock avoidance methods
10. Explain recovery from deadlock?
11. Write a short note on deadlock detection.

UNIT-III

CHAPTER 6

MEMORY MANAGEMENT

6.1 LOGICAL VS PHYSICAL ADDRESS SPACE

An address generated by the CPU is referred to as logical address. An address seen by the memory unit i.e. the one loaded into the memory address register of the memory is referred to as the physical address. The compile time and load time address binding methods generate identical logical and physical addresses. The execution time address binding scheme results in differing logical and physical addresses. Here, we refer to logical address as virtual address. The set of all logical addresses generated by the program is called logical address space and the set of all physical addresses corresponding to these logical addresses is called physical address space. The run time mapping from virtual address to physical addresses is done by a hardware device called

the Memory Management Unit (MMU). The user program supplies the logical addresses, these logical addresses must be mapped into physical addresses before they are used.

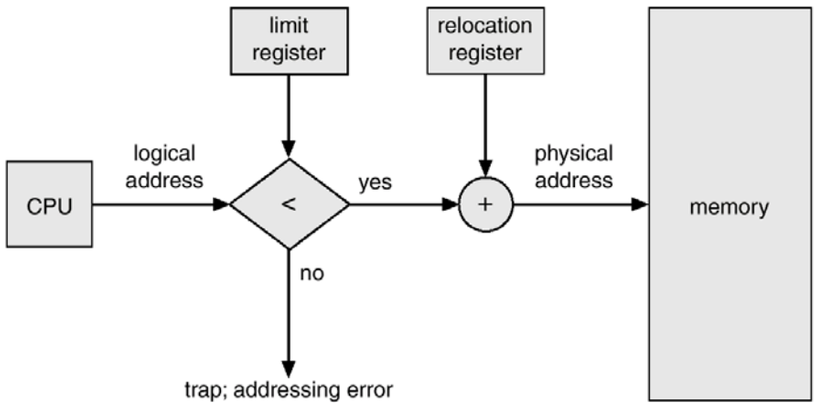


Figure :6.1 Dynamic relocation using relocation register

The base register is called as 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. The MS-DOS operating system running the Intel 80x 86 families of processors uses four relocation registers when loading and running processes.

6.2 SWAPPING

A process needs to be in memory to be executed. A process can be swapped temporarily out of memory to a backing store and then brought back to memory for continued execution. For example: assume a multi programmed environment with round robin CPU scheduling. When a quantum expires the memory manager will start to swap out the process that just finished and swap in another process to the memory space that has just been freed. In the mean time the CPU will allocate time slice to some other process in the memory. A variant of this policy is used in priority based scheduling algorithm. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute high priority process. When a higher priority process finishes, the lower priority process can be swapped in and continued. This variant of swapping is called roll-out roll-in.

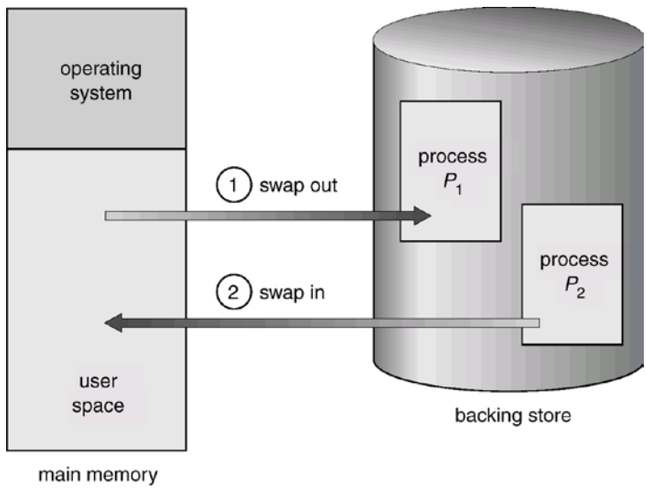


Figure:6.2 swapping of two processes using a disk as backing store

Normally, a process that is swapped out will be swapped back into the same memory space that it occupied previously. If the binding is done at assembly/ load time, the process cannot be moved to different locations. If execution time binding is being used, then a process can be swapped into different memory space because physical addresses are computed during execution time. Swapping requires a backing store. Backing store is usually a fast disk. It must be large enough to accommodate copies of all memory images for all users. It must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If there is no free memory region the dispatcher swaps out a process currently in memory and swaps in a desired process. It then reloads the registers as normal and transfers control to the selected process.

Limitations:

- Context switch time is high.
- Major part of swap time is transfer time.

6.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. Hence, we need to allocate different parts of main memory in an efficient way. The memory is divided into two parts: one for resident operating system and one for the user processes. We may place the operating system either at low memory or high memory. Since the interrupt vector is in the low memory the operating system is also placed in the low memory. Contiguous memory allocation deals with each process contained in the single contiguous section of memory.

Memory Allocation

One simple approach is to divide the memory into fixed sized partitions. Each partition may contain exactly one process. The degree of multiprogramming is bounded by the number of partitions. In multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes free for another process. The operating system keeps a table indicating which parts of the memory are available and which are occupied. Initially all memory is available for user processes and is considers as one large block of available memory - a hole. When a process arrives and needs memory, we search for a hole that is large enough for the process. If we find one, we allocate only as much memory as needed, keeping the rest to satisfy future requirements. As the processes enter the system, they are put in an input queue.

The operating system takes into account the memory requirement of each process and the amount of memory space available.

When a process is allocated space, it is loaded into memory and it can then, compete for CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue. At any given point of time, we have a list of block sizes and input queue.

The operating system can order the input queue according to any scheduling algorithm. Memory is allocated to processes until the memory requirements of the next process cannot be satisfied or no available block of memory is large enough to hold that process. The operating system can then wait until a large block that is big enough is available or it can skip down the input queue to see whether smaller memory requirement processes can be met. A set of holes of various sizes is scattered in the memory at any given point of time.

When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: one part is allocated to the arriving process and the other is returned to the set of holes. If a new hole is adjacent to other holes these adjacent holes are merged to form a larger hole. At this point of time, the system needs to check whether there are processes waiting for memory and whether their requirements can be satisfied. This is an instance of dynamic storage allocation problem - how to satisfy a request of size n from a list of free holes. The set of holes is searched to determine which of the hole is best to allocate. The most common strategies are

- First fit - Allocate the hole that is large enough. Searching can start at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching once we find a free hole that is large enough.

- Best fit - Allocate the smallest hole that is big enough. We must search the entire list unless the list is kept ordered by size. This strategy produces smallest leftover holes.

- Worst fit - Allocate the largest hole. We must search the entire list. This strategy produces the largest leftover holes which may be more useful than the smaller leftover holes from the best fit approach.

Fragmentation

Memory fragmentation could be internal as well as external. . As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request but it is not contiguous. Storage is fragmented into a large number of small holes. Physical memory is broken into fixed sized blocks and allocated memory in unit of block sizes. In this approach, memory allocated to a process will be slightly larger than the requested memory. The difference in memory space between the allocated and requested memory is called internal fragmentation - memory that is internal to a partition but not being used. One solution to the problem of external fragmentation is compaction. The goal here is to shuffle the memory contents to place all the free memory together in one large block. Compaction is possible only if the relocation is dynamic. The simplest compaction algorithm is simply to move all processes to one end of memory and all holes in the other direction, producing one large hole of available memory.

Another solution is to permit logical address space of a process to be non-contiguous thus allowing a process to be allocated physical memory where it is available. The two approaches to implement this strategy are paging and segmentation.

6.4 PAGING

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids problems of fitting varying-sized memory chunks on to the backing store.

Basic Method

Physical memory is divided into fixed-sized blocks called frames. Logical memory is broken into blocks of same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed sized blocks that are of the same size as memory frames. Every address that is generated by the CPU is divided into two parts: a page number (P) and a page offset (d). The page number is used as an index to the page table. The page table contains the base address of each page in the physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

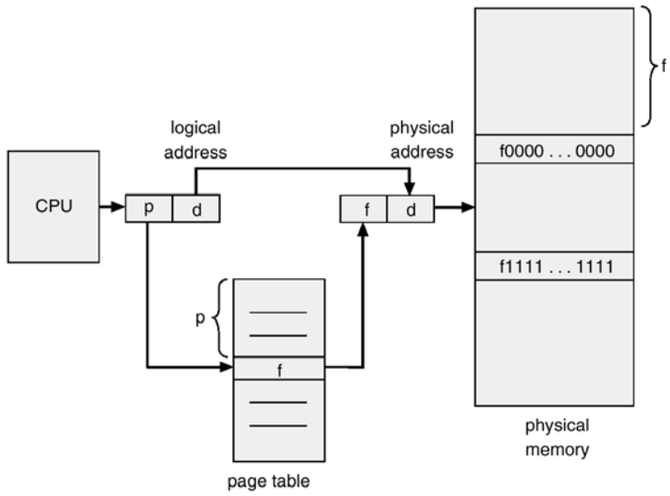


Figure:6.3 paging Hardware

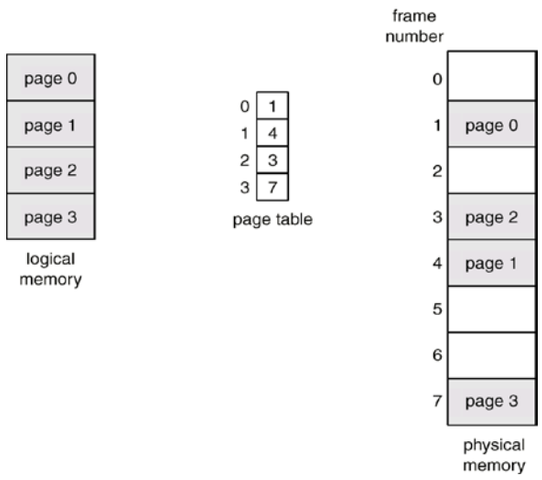


Figure: 6.4 Paging model of logical and physical memory

The page size is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 64 MB/page depending on the architecture. The selection of power of 2 as page size makes it easy to translate the logical address space to page number and offset. If the size of the logical address space is 2mand page size is 2n then the higher order m-n bits of logical address designate page number and the lower order bits designate the page offset.

Thus the logical address is as follows:

Page number page offset

F d

m-n n

Where, p - index to page table

d - displacement within the page.

Consider an example for 32 byte memory with 4 byte pages. Using page size as 4 bytes and the physical memory of 32 bytes (8 pages), illustrate how user view of memory can be mapped into physical memory.

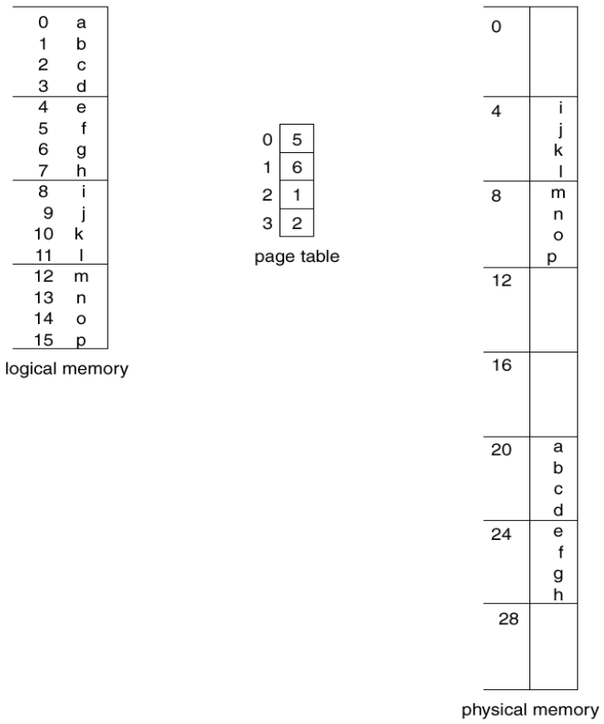


Figure:6.5 paging example for a 32-byte memory with 4 byte pages

Logical address 0 is page 0 offset O. Indexing into page table, we find that page 0 is in frame 5, Thus logical address 0 maps to physical address (5x4)+0=20. Logical address 3(page 0, offset 3) maps to physical address (5x4)+3=23. Logical address 4 is page 1 offset O. According to the page table, page 1 is mapped to frame 6. Logical address 4 maps on to physical address (6x4)+0=24. Paging is itself a form of dynamic relocation. Every logical address is bound by paging hardware to some physical address. Using paging, we have no external fragmentation. Any free frame can be allocated to a process that needs it.

However, internal fragmentation may occur. If the memory requirement of a process does not happen to fall on page boundaries, the last fame allocated may need not be completely full.

For example, if each page can hold 2048 bytes and the process requires 72766 bytes, it would need 35 pages plus 1086 bytes. It would be allocating 36 frames, resulting in an internal fragmentation of 2048-1 086=962 bytes. When a process enters a system to be executed. Its size in terms of pages is examined.

Each page of a process needs one frame. Thus, if a process requires n pages, at least n frames must be available in memory. If n frames are available in memory, they are allocated to the arriving process.

The first page of the process is loaded into one of the allocated frames and the frame number is put in the page table for this process. An important aspect of paging is the clear separation between the user memory and the actual physical memory. The user program views memory as a single contiguous space containing only this single program. But, the user's program is scattered throughout the physical memory that also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address translation hardware. The logical addresses are translated into the physical address. This aping is hidden from the user and is controlled by the operating system.

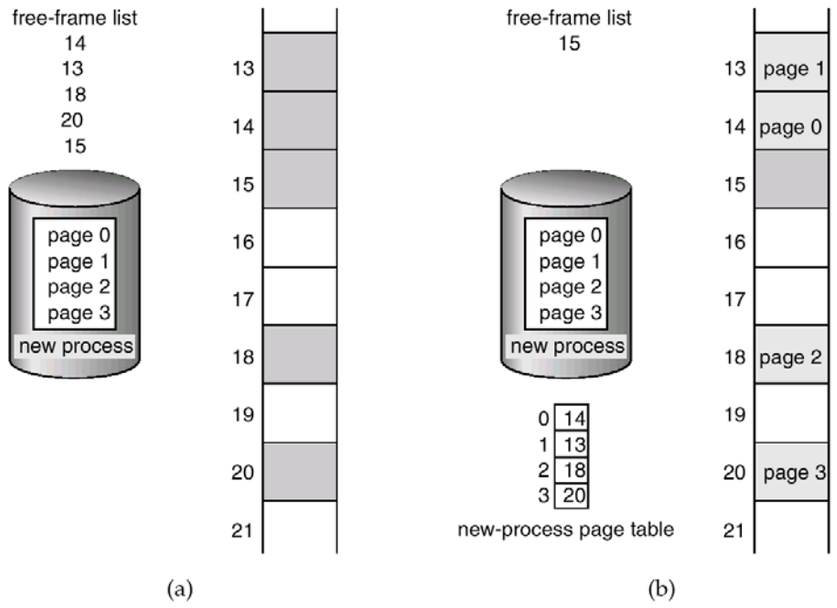


Figure:6.6 free frames (a) before allocation (b) after allocation

Since the operating system is managing the physical memory, it must be aware of the allocation details of the physical memory - which frames are available, which frames are allocated. Total frames and so on. This information is kept in a data structure called the frame table. The frame table has one entry per for each physical page frame indicating whether the latter is free or allocated. If it is allocated, to which page of which process, the operating system also maintains a copy of page table for each process in order to translate logical address to physical address. Paging increases the context switch time.

Hardware support

Each operating system has its own method of storing page tables. Most allocate each page table for each process. A pointer to the page table is stored with other register values in the process control block. When a dispatcher is told to start a process, it must reload the user registers and define correct hardware page table values from the stored user page table. In the simplest case, the page table can be implemented as a set of dedicated registers. These registers are built with high speed logic to make paging address translation efficient. Some machines keep the page table in main memory rather than in registers. In such cases, a page-table base registers (PTBR) points to a page table. Changing the page table requires only changing this one register. The problem with this is the time required to access the user memory location. A standard solution is to use a special small fast look up hardware cache called translation look aside buffer (TLB). It is an associative high speed memory. Each entry in the TLB consists of 2 parts: a tag and a value. When the associative memory is given an item, it is compared with all the keys simultaneously. If the item is found then the corresponding value field is returned. The percentage of times a particular page is found in the TLB is called the hit ratio.

6.5 SEGMENTATION

It is a memory management scheme that supports user view of memory. A logical address space is a collection of segments. Each segment has a name and length. The addresses specify both the segment name and the offset within the segment. Hence, the user specifies each address by a segment name and an offset. Thus, the logical address of a segment consists of <segment number, offset>

For example a user's view of a program may consist of subroutine, stack, symbol table, Sqrt, main program as segments.

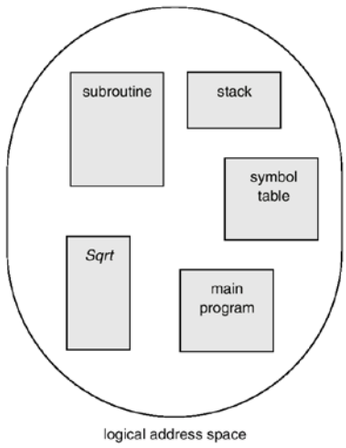


Figure:6.7 Segmentation

Segmentation Hardware

The user refers objects in the program by a two-dimensional address whereas the physical address is one dimension sequence of bytes. We now define an implementation to map 2D user defined address into one dimensional physical address. This mapping is affected by the segment table. Each entry of a segment table has a segment base and a segment limit. The segment base consists of the starting physical address where the segment resides in the memory whereas the segment limit specifies the length of the segment. A logical address consists of two parts: a segment number s and an offset into the segment d.

The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If the offset is legal, it is added to the segment base to produce the address of the physical memory.

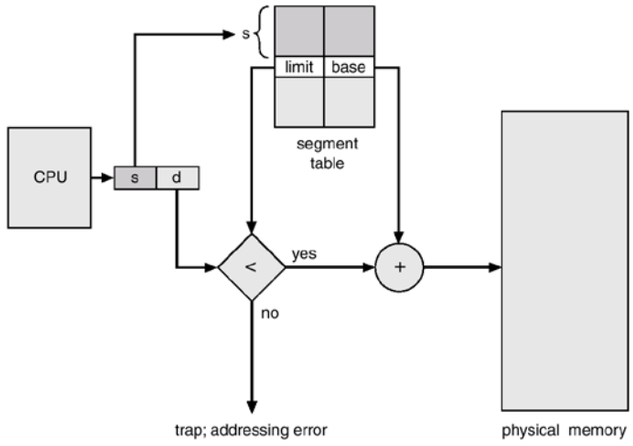


Figure:6.8 Segmentation Hardware

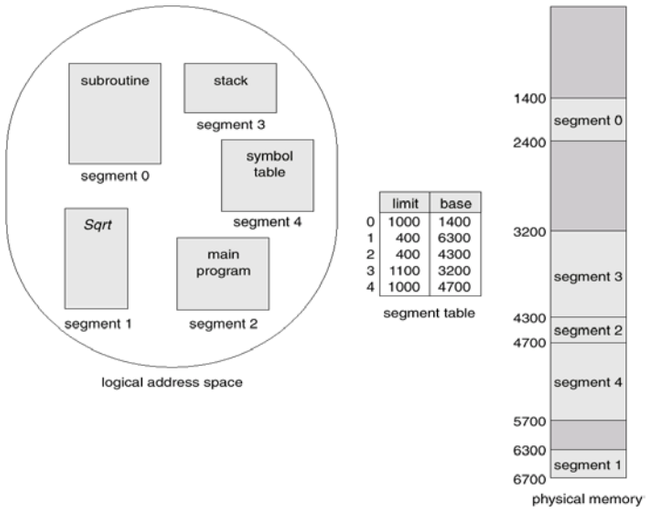


Figure:6.9 Examples of segmentation

Assignment 6

1. Explain the concept of ‘swapping’ in detail
2. Briefly explain segmentation hardware?
3. Write a short note on fragmentation.
4. Explain the segmentation with an example

5. Write a short note on following

   (i) Internal fragmentation (ii) External Fragmentation

6. Explain briefly the concept of paging

7. With a neat diagram explain paging hardware with TLB

CHAPTER 7

VIRTUAL MEMORY

INTRODUCTION

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than the physical memory. It allows processes to easily share files and address spaces. Virtual memory is the separation of user logical memory from physical memory. Virtual memory makes the task of programming much easier because the programmer no longer needs to worry about the amount of physical memory available. Virtual memory is implemented using demand paging, segmentation and demand segmentation.

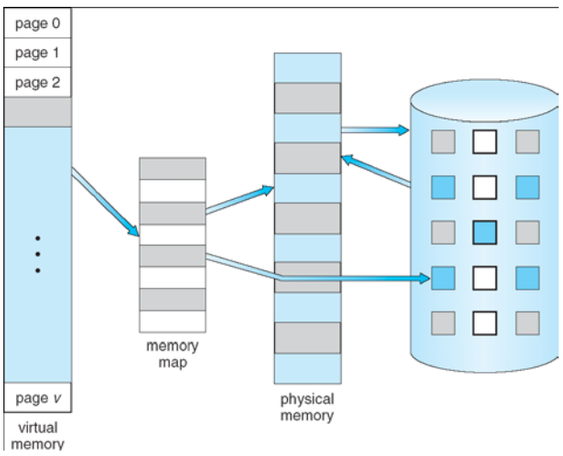


Figure:7.1 Diagram showing virtual memory that is larger than physical memory

7.1DEMAND PAGING

A demand paging system is similar to paging system with swapping. Processes reside on the secondary memory. When we want to execute a process, we swap it in to the memory. Here, we use a lazy swapper. A lazy swapper never swaps a page into memory unless it is needed. A swapper manipulates the entire process whereas a pager is concerned with individual pages of a process.

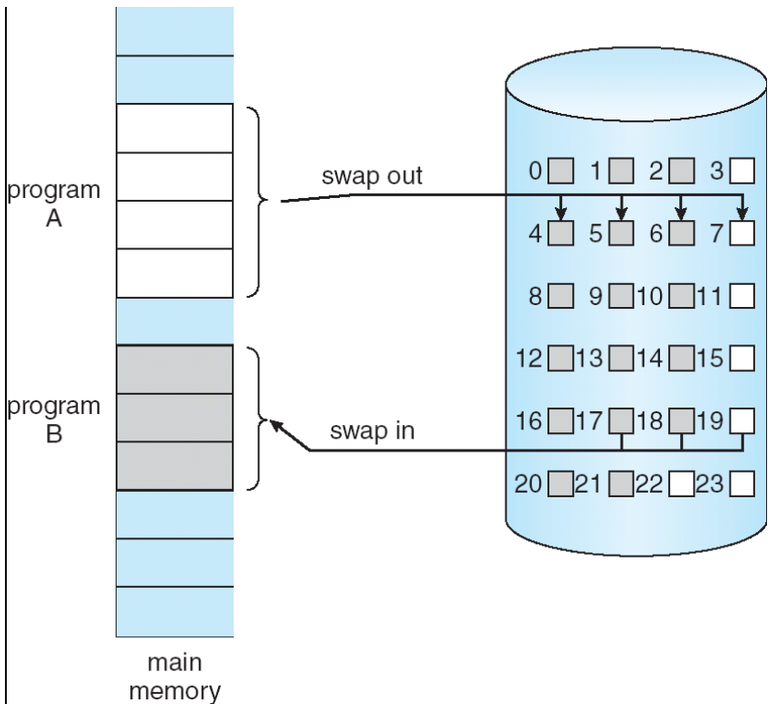


Figure:7.2 Transfer of a paged memory to contiguous disk space

Basic Concept

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. The pager then brings only the necessary pages into the memory.

Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed. The valid-invalid bit is used to distinguish between those pages that are in memory and those that are on the disk. When this bit is set to "valid" it indicates that the associated page is both legal and in memory. If the bit is set to "invalid" it indicates that the page is not in the logical address space of the process or is on the disk. The page table entry for a page that is currently not in memory is marked as invalid or it contains the address of the page on the disk. If the required number of pages are brought into the memory then, the execution proceeds normally. If the process tries to access a page that was marked as invalid, then a page-fault trap occurs. The trap is a result of the operating system not able to bring the desired page into the memory.

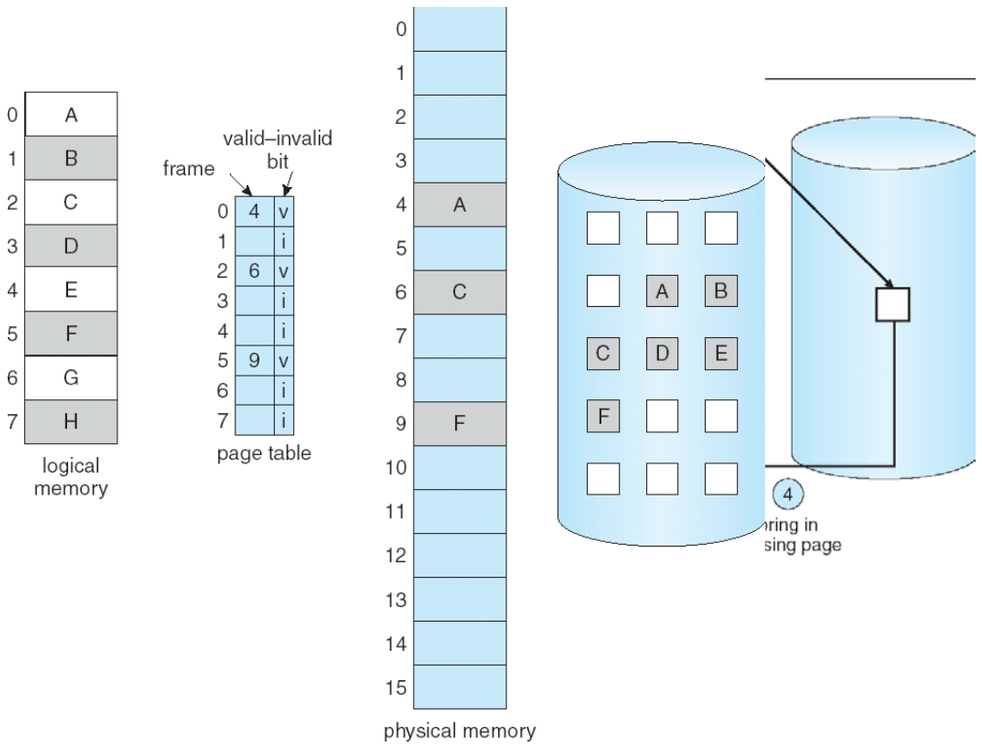


Figure:7.3 Page table when some pages are not in main memory

The procedure to handle the trap is as follows.

1. We check the internal table in the PCB to determine whether the reference is valid or invalid memory access.
2. If the reference is invalid terminate the process. If it was valid but we have not yet brought in that page we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table and page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

Figure:7.4 Steps in handling a page fault

In the extreme case, we could start executing a process with no pages in the memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory resident page, the process immediately faults for the page. After this page is brought into the memory, the process continues to execute, faulting as necessary until every page it needs is in memory. At that point, it can execute with no more faults. This scheme is called pure demand paging: Never bring a page into memory until it is required.

The hardware to support demand paging are

a) Page table - This table has the ability to mark an entry invalid though a valid-invalid bit or special value of protection bits.

b) Secondary memory - This memory holds those pages that are not present in the main memory.

The secondary memory is a high speed disk. It is known as swap device and the section of disk used for this purpose is called swap space

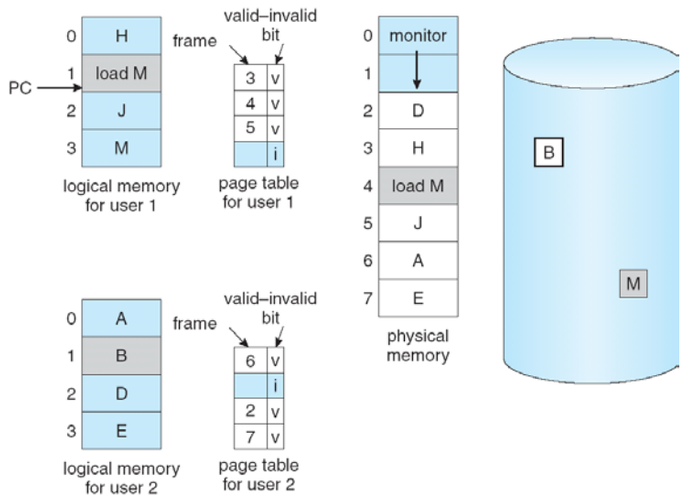


Figure:7.5 Need for page replacement

7.2PAGE REPLACEMENT

Page replacement takes the following approach - if no frame is free we find the one that is not currently being used and free it. We can free a frame by writing its contents to a swap space and changing the page table to indicate that the page is no longer in memory.

Now the freed frame can be used to hold a page for which the process faulted.

The page fault service routine is modified as follows:

1. Find a location of desired page on the disk.

2. Find a free frame

   a) If there is a free frame use it.

   b) If there is no free frame, use a page replacement algorithm to select a victim frame.

   c) Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the new free frame. Change the page and frame tables.

4. Restart the user process.

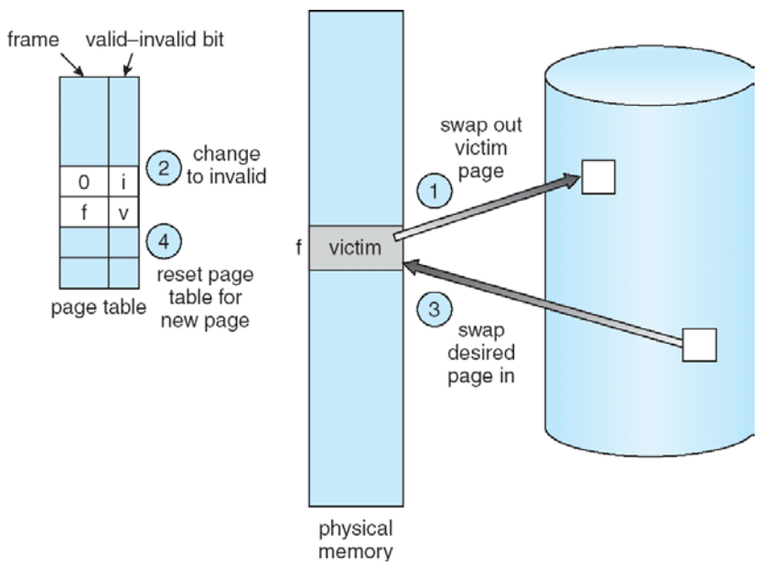


Figure: 7.6 Page Replacement

Each page/frame may have a modify bit associated with it in the hardware. The modify bit in the page is set whenever a word/byte is written into it indicating the page has been modified.

When a page is selected for replacement, the modify bit is examined. If it is set, we know that the page has been modified since it was read from the disk. In this case the page must be written to the disk.

If the bit is not set, the page has not been modified since it was read into the memory.

In order to implement demand paging we need to develop

a) Frame allocation algorithm

b) Page replacement algorithm

The string of memory references is called the reference string**.** Reference string can be generated artificially using random number generator or we can trace the given system and record the address of each memory reference. As the number of frames increase, the number of page faults decreases

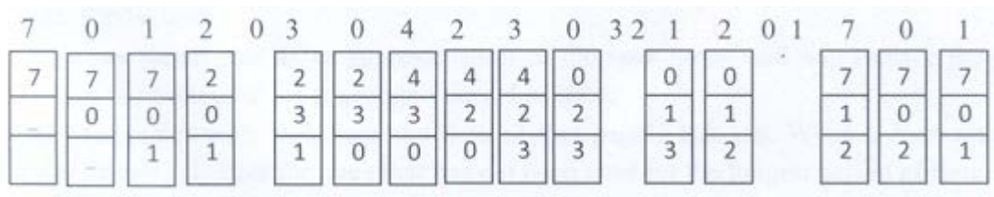
FIFO page Replacement

It is the simplest page replacement algorithm. This algorithm associates with each page the time when it was brought into memory. When a page is to be replaced the oldest page is chosen. We can create a FIFO queue to hold all the pages in memory. We can replace the page at the head of the queue. When a page is brought into memory, we insert it into the tail of the queue. Let us assume our reference string to be

7012030423032120170 1

Initially the first three frames are empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces 7 because, 7 was brought in first. Since 0 is the next reference and 0 is already in the memory, we have no page fault for this reference. The first reference to 3 results in a page 0 being replaced since it was the first of three pages in memory (0, 1, 2). The process continues and there are altogether 15page faults.

Reference string



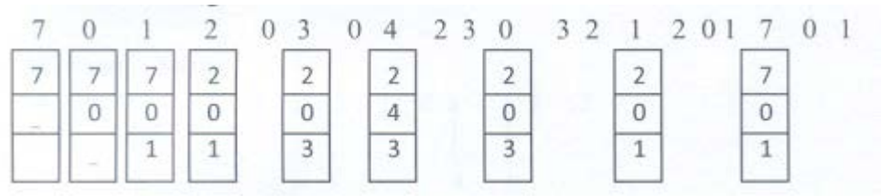
FIFO replacement algorithm is easy to understand and to program. The performance is not always good. If we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve an active page. Some other page will need to be replaced to bring the active page back to memory. Thus, a bad replacement choice increases the page fault rate and decreases the process execution speed. For some page replacement algorithm the page fault rate may increase as the number of allocated frames increase. This phenomenon is called Belady's anomaly.

PAGE REPLACEMENT ALGORITHM

This algorithm has the lowest page fault rate of all algorithms. The theme of this algorithm is- Replace the page that will not be used for the longest period of time. For the above reference

string, the first three references cause faults that will fill the three empty frames. The reference to page 2 replaces page 7 because page 7 will not be used until reference 18, whereas page 0 will be used at 5 and page 1 at 14. Reference to page 3 replaces page 1 as page 1 will be the last of three pages in the memory to be referenced again. This algorithm requires 9 page faults.

Reference string



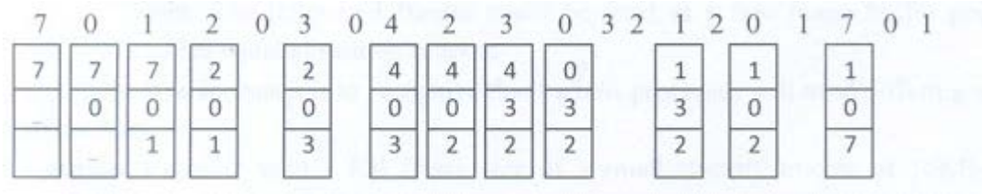
This algorithm is difficult to implement, because it requires a future knowledge of reference string.

Least Recently Used (LRU) Replacement algorithm

The distinction between FIFO and Optimal replacement is that FIFO uses the time when a page was brought into memory; optimal replacement algorithm uses the time when a page is to be used. LRU uses recent past as an approximation of the near future and will replace the page that has not been used for the longest period of time.

LRU 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 optimal page replacement algorithm looking backward in time. This algorithm produces 12 page faults for the given reference string.

Reference string



This algorithm requires substantial hardware assistance in order to be implemented.

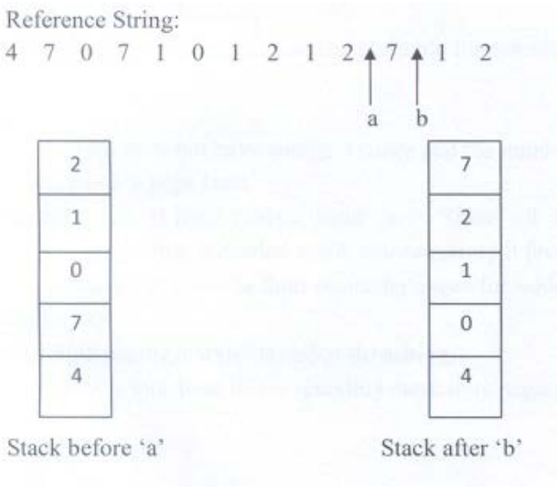
Two implementations are used to determine the order for frames defined by the tie of last use. They are:

a. Counters - We associate each page table entry a time of use field and add to the CPU a logical clock/counter. The clock is incremented for every memory reference.

- Whenever a reference to a page is made, the contents of the clock register are copied to the time of use field in the page table entry for that page.
- We replace the page with the smallest time value.

b. Stack - Another method to keep track of LRU is to keep a stack of the page numbers.

Whenever a page is referenced it is removed from the stack and put on the top. In this way the top of the stack is always the most recently used and the bottom of the stack is least recently used.



7.4 ALLOCATION OF FRAMES

The easiest way to split m frames among n processes is to give every process an equal share of m/n frames. For example if there are 93 frames and 5 processes, each process will get 18 frames. The leftover 3 frames could be used as a free frame buffer pool. This scheme is called equal allocation scheme. An alternative approach is to recognize that various processes will need differing amounts of memory. Consider a system with 1 KB frame size. If a small student process of 10KB and an interactive database of 127KB are the only 2 processes running on the system with 62 frames, it doesn't make much sense to give each process 31 frames. The student process needs only 10 frames and 21 frames are wasted. In such situations, we use proportional allocation. We allocate memory to each process depending on its size.

Let the virtual memory for process P_i be S_i and define $S=\sum S_i$. If the total number of available frames is m, we allocate a_i frames to process P_i where a_i is approximately

$a_i = (s_i/S) * m$

For student process:

10 / 137 * 62 =~ 4 frames

For iateractive database process:

127 / 127 * 62 =~ 57 frames

Hence both the processes share the available frames according to their needs.

7.5 THRASHING

If a process does not have enough frames and the number of allocated frames is minimum, it will quickly page fault. At this point, it must replace some page. Since, all the pages are in active use, it will replace a page that is needed again. Consequently it faults again and again. The process continues to fault replacing pages for which it then faults and brings back in right away. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

7.6 Assignment-7

1. Explain the concept of demand paging?
2. Explain FIFO page replacement algorithm with an example
3. Consider the following page reference string: (M.U. Oct/Nov. 2009)

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

How many page faults would occur for the following replacement algorithm assuming three frames?

(i)LRU algorithm

(ii) Optimal replacement algorithm

4. What are the limitations of fixed partition allocation method for memory management?
5. Explain the concept on multiple-partition allocation
6. When do page fault occur? Explain the FIFO page replacement algorithm using the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 assuming four frames.
7. Consider the following reference string. Assuming three frames find the number of Page faults using FIFO page replacement algorithm.

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

UNIT IV

CHAPTER 8

FILE SYSTEMS

8.1 Introduction

The file system consists of

- Collection of files, each storing related data
- Directory structure which organizes and provides information about all files in a system
- Partitions which are used to separate physically or logically large collections of directories

8.2 File Concept

- A file is a logical unit of storage in a computer
- Files are mapped onto physical storage devices by the OS
- A file is a named collection of related information that is recorded on a secondary storage
- A file has a defined structure according to its type. A text file is a sequence of characters organized into lines. A source file is a sequence of functions and sub routines. Functions can be organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the systems' linker. An executable file is a series of code sections that the loader can bring into memory and execute.

8.2.1 File Attributes:

- Every file has a name by which it is referred. The attributes of a file include
- Name – symbolic file name is the only information that is in human readable form.
- Identifier – This is a unique tag, usually a number. It identifies the file within the file system. It is a non-human readable name of the file.
- Type – this information is needed for those systems that support different types of files
- Location – this is a pointer to a device and to the location of the file on that device.
- Size – the current size of the file
- Protection – access control information that determines who can read, write or execute the file
- Time, date and user identification – this information may be kept for creation, last modification and last use. This data can be useful for protection, security and usage monitoring

8.2.2 File Operations:

- A file is an abstract data type. The basic operations on files are

    a) Creating a file – Two steps are necessary to create a file. First, required space must be found in the file system. Second, an entry for the new file must be made in the directory. The directory records the name of the file, location in the file system and other information.
    b) 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 on to the file. Given the name of the file, the system searches for the directory to locate the file. The system must keep a write pointer to the location in a file where the next write is to take place. The write pointer must be updated whenever a write occurs
    c) Reading a file – To read from a file, we use a system call that specifies the name of the file and where the next block of file should be kept in memory. The system needs to keep a read pointer at a location in a file where the next read is to take place. Once the read has taken place, the read pointer is updated. Both read and write use the same pointer thus saving space and reducing system complexity
    d) Repositioning within a file – The directory is searched for appropriate entry and the current-file-position is set to a given value. This operation does not involve actual I/O. This file operation is also called **file seek**.
    e) Deleting a file – To delete a file, search the directory for a file. Having found the file, we release all the file space so that it can be reused by other files and erase that entry.
    f) Truncating a file – The user may want to erase the contents of a file. Rather than forcing the user to delete the file and recreate it, this function allows all attributes to remain unchanged except the length. It resets the file length to zero and releases the file space

- Open file table: The OS keeps a small table containing information of all open files in a open file table. This is done to facilitate easy searching of files. When a file operation is requested, the file is specified via an index into this table. Hence no searching is required. When the file is no longer actively used, it is close by the process and the OS removes its entry from the open file table
- File pointer: The system must keep track of the last read/write location as a current file position pointer. This pointer is unique to each process operating on a file
- File open count: This counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry from the open file table.

8.3 File Types:

- It is necessary that the OS should recognize and support various file types. If an OS recognizes the type of the file only then it can operate on the file in a reasonable ways. The type of the file is included as a part of its name.
- The name of the file contains 2 parts – file name and its extension. They are separated by a period(.)
- For example in MS DOS, name can contain up to 8 characters followed by a period followed by the extension of 3characters. The system uses this extension to indicate the type of the file and the operations that can be performed on it.

File type	Extension	Function
Executable	exe, bin, com	Read to run machine language program
Source code	c, cpp, java	Source code in various languages
Text	txt, doc	Textual data, documents
Multimedia	mpeg, mov, rm	Binary file containing audio/video information

8.4 File Structure:

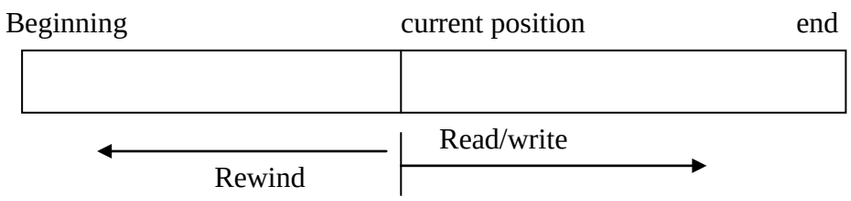
- File types are used to indicate the internal structure of a file. The file structure depends on the OS being used.
- All disk I/O are performed in units of one block and all blocks are of same size.
- It is unlikely that the physical block size and the logical record size will match.
- Logical records may vary in length
- **Packing** is a technique where a number of logical records are made into physical blocks.
- The record size, the physical block size and the packing technique determines how many logical records can be in each physical block. Packing can be done using user's application program or by the OS
- Disk space is always allocated in blocks. Some portion of the block of each file is generally wasted.
- For example: if the block size is 512 bytes and the file size is 1949 bytes, the number of blocks needed is 1949/512 = 4(approx).But, 4x512=2048 bytes out of which only 1949 bytes are used. i.e. 2048-1949=99 bytes are wasted.

- This wasted bytes allocated to keep everything in units of blocks is called **internal fragmentation**.
- All file systems suffer from internal fragmentation. The larger the block size the greater the internal fragmentation.

8.5 Access Methods:

a) Sequential Access:

- It is the simplest access method. Information in a file is processed one after the other.
- This mode of access is used by compilers and editors
- The operation that can be performed are read/ write
- A read operation reads the next portion of the file and automatically advances the file pointer.
- A write operation appends to the end of the file and advances to the new end of the file
- Such files can be reset to the beginning of the file and also skip forward/backward n records.
- It works well on sequential access and random access devices



b) Direct Access:

- This method is also known as relative access
- A file is made up of fixed length logical records that allow a program to read/write records rapidly in no particular order.
- In this mode, a file is viewed as a numbered sequence of blocks or records. A direct access file allows arbitrary blocks to be read or written
- Databases are of this type.
- The file operations must be modified to include block number as a parameter. Thus, the operations on this file are

  Read n for read next

  Write n for write next , where n is the block number
- The block number provided by the user to the OS is a relative block number. A relative block number is an index relative to the beginning of the file. Thus, relative bock of the file zero even if the absolute disk address begins at say 4152

- Given a logical record length L, a request for record N is turned into an I/O request for L bytes starting at location L*(N-1) assuming that the first record is N=1

Sequential access	Implementation for Direct access
reset	cp=0
Read next	Read cp; cp=cp+1
Write next	Write cp; cp=cp+1

c) Other Access Methods:

- Other access methods can be built on top of direct access methods
- These methods involve construction of index for the file.
- The index contains pointers to various blocks. To search a record in a file, we first find the index and then use the pointer to access the file directly and find the required record.

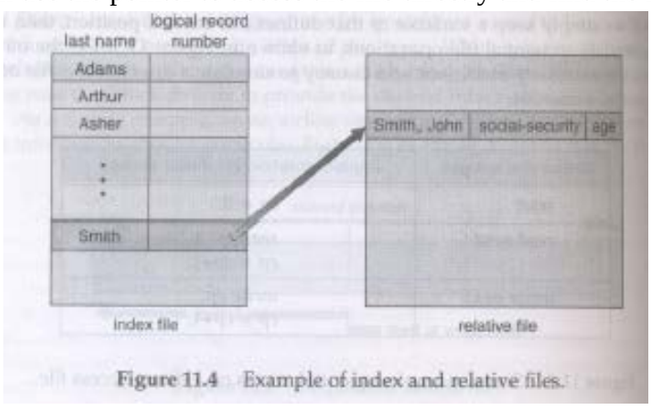


Figure 11.4 Example of index and relative files.

8.6 Directory Structure:

- Computer systems store large volumes of data. To manage these data, we need to organize them. This organization is done in two parts

1) Disks are split into one or more partitions called minidisks or volumes.
   - Each disk on the system contains at least one partition, which is a low level structure in which files and directories reside. Partitions are used to provide separate areas within one disk, each treated as a separate storage device
   - The user needs to be concerned only about the logical directory and file structure
2) Each partition contains information about the files in it.
   - Each partition contains information about files within it.
   - This information is kept in entries in a device directory or volume table of contents

- The directory records all information like name, size, location, type for all files on that partition.

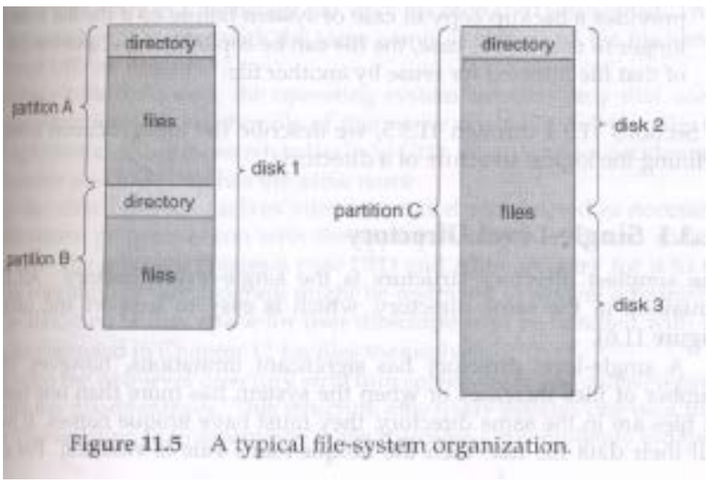


Figure 11.5 A typical file-system organization.

8.7 Operations on Directories:

- Search for a file – We should be able to search a directory to find entry for a particular file. In addition, we must also be able to find all files whose name match a particular pattern
- Create a file – new files should be created and added to the directory
- Delete a file – we should be able to remove a file from a directory
- List a directory – we should be able to list all the contents of a directory
- Rename a file – allow to change the name of an existing file
- Traverse a file system – we should be able to access every directory and also each and every file within the directory

8.8 Logical Structure of a Directory:

a) Single level directory :
   - It is the simplest directory structure
   - All the files are contained in the same directory, which is easy to support and understand

Limitations:

- When the number of files increase or when the system has more than one user, it is difficult to manage the files

- Since all the files are in the same directory, they must have unique names. If two users give the same name to different kinds of files, then, the unique name rule is violated

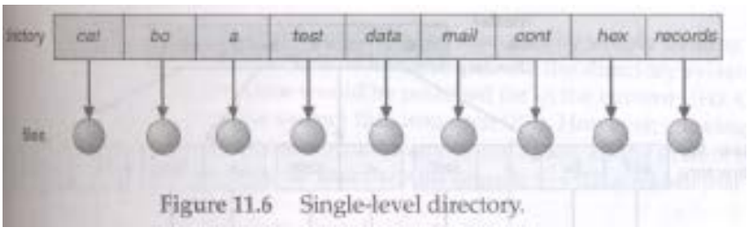


Figure 11.6 Single-level directory.

b) Two-level directory:

- In order to overcome the limitation of single level directory, here, we create a separate directory for each user
- Each user has his own user file directory(UFD)
- Each UFD has a similar structure and lists files only of a single user.
- When a user logs in the Master File Directory is searched. MFD is indexed by user name and each entry points to the UFD of that user
- When a user refers to a particular file, only his own UFD is searched
- The system administrator is responsible for creation/deletion of user directories

Limitations:

- It isolates one user from the other. The isolation is an advantage when users are completely independent, but is a disadvantage when users want to cooperate on some task and to access one another's file

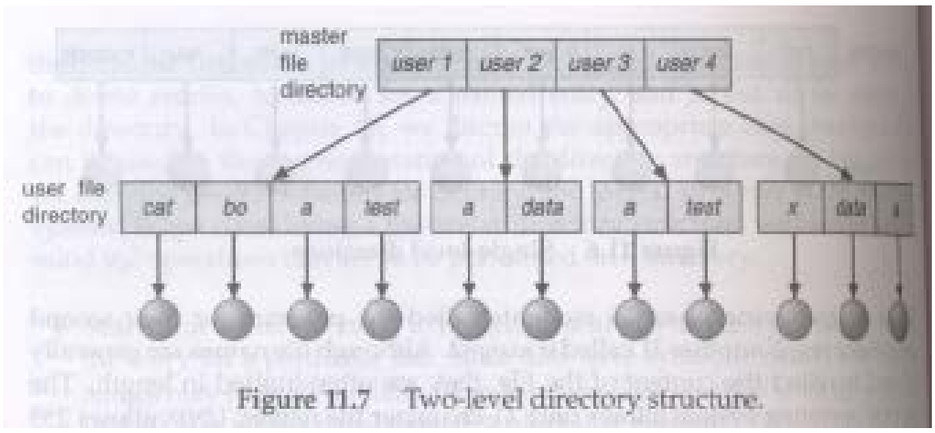


Figure 11.7 Two-level directory structure.

c) Tree structured directory:

- A two level directory is a two level tree
- The generalization of two level directory structure to a tree of arbitrary height where the user can create his own sub directories and files is called a tree-structure

- It is the most commonly used approach in many OS
- MS-DOS is structured as a tree.
- The tree has a root directory. Every file in the system has a unique path name. A path name. a path name is a path from the root to the specified file through sub directories
- A directory may contain set of files or sub directories. All directories have the same internal format.
- One bit in each directory entry defines the entry either as a file(0) or as a sub directory(1)
- Each user has a current directory. A current directory refers to the most commonly referred directory by the user.
- The user can change the current directory whenever he desires to.
- A directory can be deleted only if it is empty

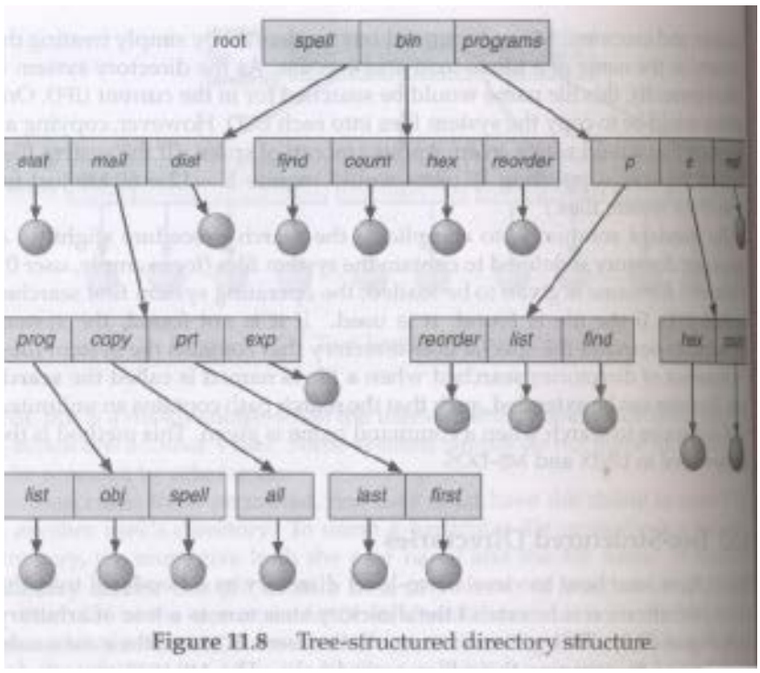


Figure 11.8 Tree-structured directory structure.

- Path names are of two types:
  1) Absolute path names: It begins with the root and follows a path down to the specified file giving the directory names on the path
  2) Relative path names: It defines a path from the current directory

  Example: If the current directory is root/test/test2, the absolute path is root/test/test2/prt/x and the relative path is prt/x.

d) Acyclic graph directory:

- A tree structure prohibits sharing of files and directories

- An acyclic graph structure allows directories to have shared sub directories and files. An acyclic graph is a graph with no cycles
- With a shared file only actual file exists and any changes made to this file are visible to others who share that file.

Limitations:

- A file may have multiple absolute path names
- Deletion is also complex task because files are shared

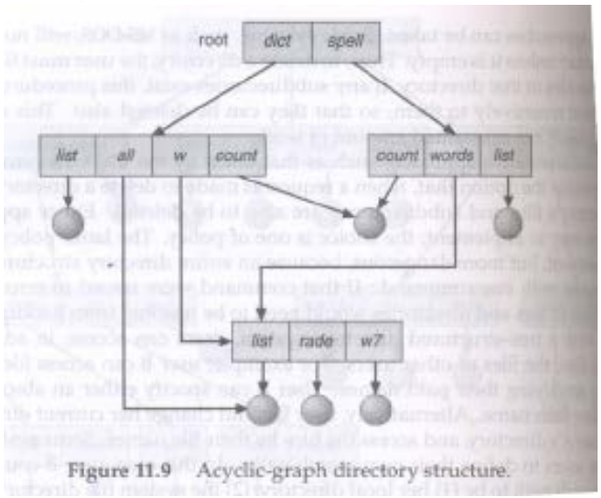


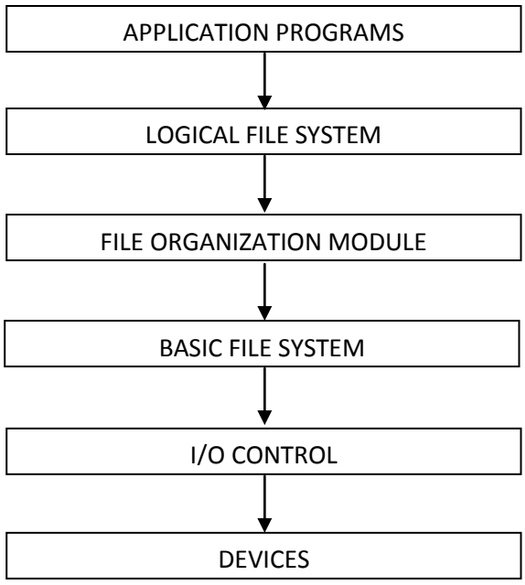
Figure 11.9 Acyclic-graph directory structure.

e) General graph directory:

- One major problem of acyclic graph is to ensure that there are no cycles.
- The primary advantage of acyclic graph structure is the simplicity with which algorithms traverse the graph an determine when there are no more references to a file.
- If cycles are allowed to exist, the algorithm may result in an infinite loop continually through the cycle without terminating.
- In an acyclic graph structure, a reference count 0 indicates there are no references to a file or directory and the file can thus be deleted.
- When a cycle exists, reference count is non-zero even when there is no longer a possibility to refer a file or directory.
- In this case, we use garbage collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.
- Garbage collection involves traversing the entire file system and marking everything that can be accessed.
- A second pass collects everything that is not marked onto a list of free space.

8.10 File system Structure:

- Disks provide the bulk of secondary storage on which a file system is maintained
- The two characteristics that make them convenient for use are
  a) They can be rewritten in place i.e. it is possible to read a block from the disk, modify the block and write it back into the same place
  b) They can directly access any block of information. It is simple to access any file sequentially or randomly. Switching from one file to another requires only moving the read/write head or waiting for the disk to rotate



- The file system is composed of different levels
- The lowest level, the I/O control consists of device drivers and interrupt handlers to transfer information between main memory and the disk system
- The device drivers can be thought of as translators. The input to it is a high level command. The output of the device driver is a low-level hardware specific instruction that is used by the hardware controller, which interfaces I/O with the rest of the system
- The basic file system needs to issue generic commands to appropriate device drivers to read and write physical blocks on the disks.
- The file organization module knows about the files, their logical blocks as well as physical blocks. By knowing the type of file allocation, this module translates logical block address to physical block addresses. It also consists of free space

manager, which tracks unallocated blocks and provides these blocks to file organization when requested

- Logical file system manages the meta-data information. It manages the directory structure to provide file organization module with the information the latter needs, given the symbolic file names. It maintains the file structure via File Control Block (FCB). FCB contains information about the file including ownership, permissions and location of the file

8.11 Allocation Methods:

- Three major methods of allocating disk spaces are as follows:
  a) Contiguous allocation
  b) Linked allocation
  c) Indexed allocation

a. Contiguous allocation:

- This method requires each file to occupy a set of contiguous blocks on the disk.
- Disk addresses define linear ordering on the disk. With his ordering, it is possible to access b+1 block after b block with no head movement at all
- Hence, the number of disk seeks in this method is minimal
- Contiguous allocation of a file is defined by disk address and the length of the first block,
- 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 the file.
- Accessing the file using this method is very 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
- One limitation here is to find free space for a new file. This is generally seen in dynamic storage allocation. First fit and the best fit are the most common strategies used to select a free hole from a set of available blocks
- The above approach suffers from external fragmentation. As files are allocated and deleted, free disk space is broken into pieces. External fragmentation exists when free space is broken into chunks. When the largest contiguous chunk is insufficient for a request, storage is fragmented into a number of holes, none of which is large enough to store data.

- Another problem is to determine the space needed for a file

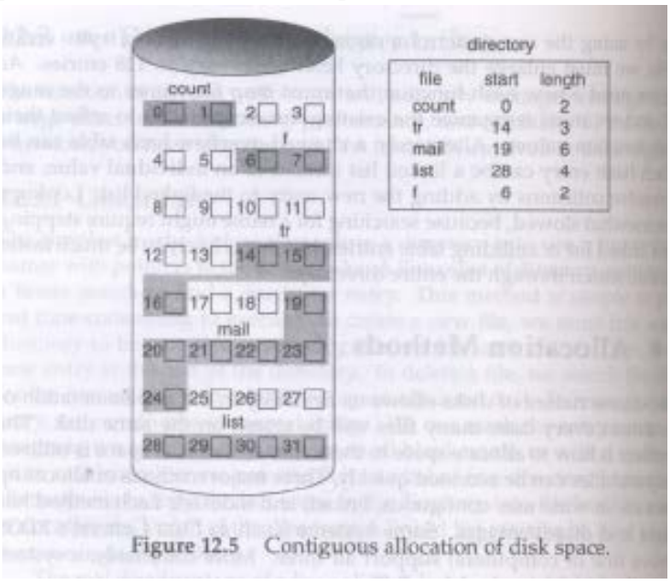


Figure 12.5 Contiguous allocation of disk space.

b. Linked allocation:

- This method solves the 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 a file.
- To create a new file, we create a new entry in the directory. Each directory entry has a pointer to the first disk bloc of a file. This pointer is initialized to nil to signify an empty file. The size field is also set to 0.
- A write operation on the file causes a free block to be found and a new block is written to and linked to the end of the file
- To read a file, we simply read the blocks by following the pointers
- There is no external fragmentation and any free block on the free space can be used to satisfy a request, the size of a file need not be declared when the file is created.

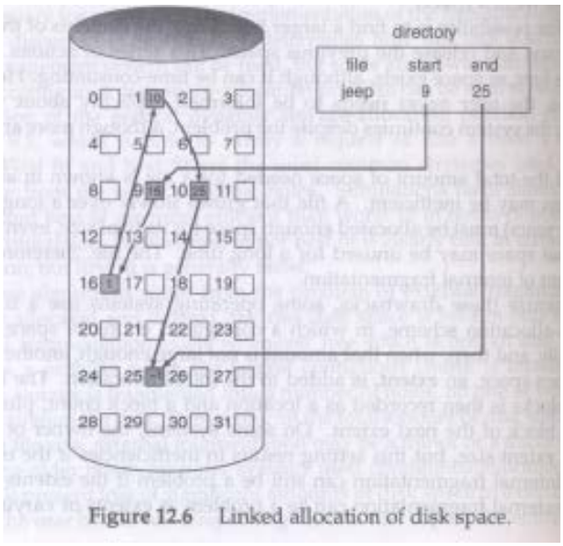


Figure 12.6 Linked allocation of disk space.

Limitations:

- It can be used only for sequential access files
- Space is required for pointers also
- Since files are linked together by pointers, there could be a chance for lost/damaged pointers.

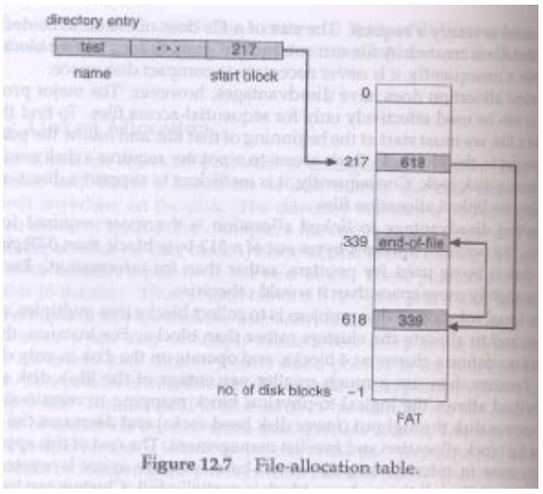


Figure 12.7 File-allocation table.

- A variation of this approach is a File Allocation Table(FAT)
- This is an efficient method used in MS-DOS
- A section of the disk at the beginning of each partition is set aside to contain FAT
- The directory entry contains the block number of the first block of the file

- The table entry indexed by that block number then contains the block number of the next block in the file. The chain continues until the last block which has a special EOF value as table entry is found
- Unused blocks are indicated by 0 table value
- Allocating a new block to s file is simple. First find the 0 valued table entry and replace the previous end of file value with the address of the new block
- The 0 is then replaced with the end of file value

c. Indexed Allocation:

- Linked allocation cannot support direct access
- Indexed allocation solves this problem by bringing all pointers together into one location – index block
- Each file has its own index block, which is an array of disk block addresses. The i^{th} entry in the index points to the i^{th} block
- The directory contains address of index block
- To read the i^{th} block, we use the pointer in the i^{th} index block entry to find and read the 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.

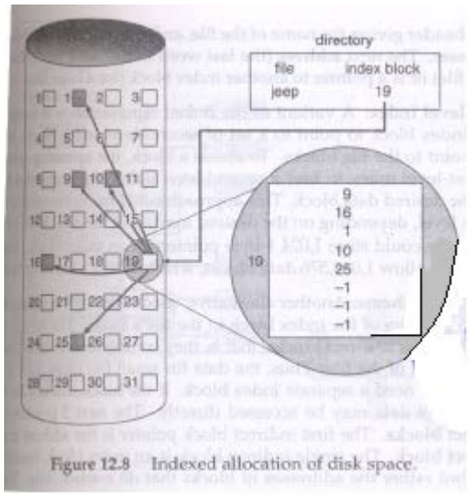


Figure 12.8 Indexed allocation of disk space.

- Every file needs an index block. Thus, an index block should be as small as possible. But, if the index block is small, it cannot hold pointers for a large file. The above problem can be solved using the following approaches

a) Linked scheme: An index block is normally on disk block. Thus, it can be read/written directly by itself. To allow large files, we may link several index

blocks. An index block may contain a small header giving name of file and a set of first n disk-blocks. The next address is nil for a small file or pointer to another index block in case of large file.

b) Multi-level index: This is a variant to linked scheme. A first level index block contains pointer set to the second block which in turn points to the file block. To access a block of information, the OS traverses from one index block to another until the actual file is reached.

c) Combined scheme: This scheme is used in Unix file system. The first say 15 pointers of index block are in file's inode. The first 12 pointers point to the direct blocks. The next three pointers are indirect blocks. The first indirect block is a single indirect block that contains address of blocks that contain data. The second indirect block is a double indirect block and the third indirect block is a triple indirect block.

8.12 Free Space Management:

- Since disk space is limited, we need to reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a free space list. It records all the free disk blocks- those not allocated to any file or directory. Free space is implemented as follows:

  1) Bit vector: Free space is usually implemented as a bit map or bit vector. Each block is represented by one bit. If the block is free the bit is 1 and if the bloc is allocated the bit is 0. For example consider a disk where blocks 2,3,4,6,7,9,11,12 and 15 are free and others are allocated. The free space bit map would be

0	0	1	1	1	0	1	1	0	1	0	1	1	0	0	1	……

  The main advantage here is the simplicity and efficiency in finding the first free block or n consecutive free blocks on the disk.

  2) Linked list: In this scheme, we link together all free disk blocks keeping a pointer to the first free block in a special location on the disk and caching it in memory. The first free block contains a pointer to the next free block and so on. The linked free space list is as follows

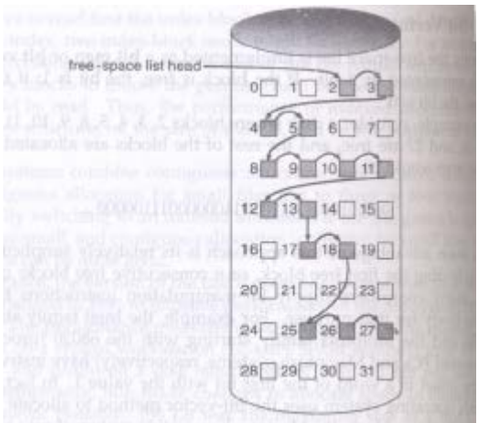


Figure 12.10 Linked free space list on disk.

3) Grouping: In this approach, the addresses of the first n free blocks are stored in the first free block. The first of these (n-1) blocks are free. The lat block contains the address of another n blocks and so on.
4) Counting: Here, we keep the address of the first free block and the number n of the free contiguous blocks hat follow the first block. Each entry in the free space list consists of disk address and a count.

8.13 Protection of File System:

- When information is stored in a computer system, we want to keep it safe from physical damage and improper access. Reliability can be provided by duplicate copies of files. Protection can be provided in many ways. Some aspects are as follows:

Types of Accesses:

- The need to protect files is a direct result of ability to access files.
- To provide protection we need controlled access.
- Protection mechanisms provide controlled access by limiting the types of file accesses that can be made. The different types of operations that may be controlled are read, write, execute, append, delete and list

8.14 Access Control:

- The most common approach to protection problem is to make access dependant on the identity of the user.
- Various users may have different types of access to files or directories
- Identity dependent access is implemented by associating each file/directory with Access Control List (ACL) specifying the user name and the types of access allowed for each user.

- When a user requests access to a particular file, the OS checks the access list associated with that file.
- If that user is listed for that requested access then, the access is allowed. Otherwise, protection violation occurs and the user job is denied access to the file.
- The main problem of the above approach is length of access lists.
- To condense the length of ACL, systems have recognized 3 classifications of users in connection with each file.
  - Owner – the user who created the file
  - Group – a set of users who share the file and need similar access
  - Universe – all other users of the system
- One mechanism is to associate a file with a password.
- Passwords must be chosen randomly and changed often. This scheme is effective in limiting the access to a file only to those users who know the password.
- The limitations of this approach are
  a) The number of passwords a user needs to remember may be large
  b) If only one password is used for all the files, then once it is discovered, all files are accessible
  c) Commonly, one password is associated with all the user's files.

Assignment Questions:

1. Explain the various attributes of files
2. Explain the various operations performed on files.
3. Explain the different types of files
4. Explain the various file access methods
5. Explain about the different directory structures
6. Explain the general structure of file system with a neat diagram
7. Explain the various allocation methods for files
8. Explain the different free space management techniques
9. How does an operating system protect its file? Explain

CHAPTER 9

UNIX

INTRODUCTION

In the mid of 1965, a joint venture was undertaken by Bell Laboratories, General Electric Company and Massachusetts Institute of Technology (MIT) to develop an Operating System (OS) that could serve a large community of users and allowed them to share data. That Operating System was called as MULTICS(Multiplexed Information Of Computing Service).

In 1969, Bell labs withdrew from the MULTICS effort. Some of the Bell Labs programmers who had worked on this project, Dennis Ritchie, Ken Thompson, Rudd Canaday, Joe Ossanna and Doug McIlroy designed and implemented the first multi-user operating system on PDP-7 computers. Brain W. Kernighan, who was connected with MULTICS, named this OS as UNIX (Uniplexed information of computing services). In 1971, this UNIX was ported to the PDP-11 computers. But this version was not easy to port on new machines, because all it's coding were machine dependent .

To resolve this problem, Ken Thompson created a new high-level language called “B” and rewritten the whole UNIX code in this high-level language. This language lacked in several features, so, Dennis Ritchie shifted the inadequateness orb and modified it to a new high-level language called “C”. Finally, the UNIX is written in this “C” language to stand tall on any machine.

9.1FEATURES OF UNIX

UNIX has several features including the following:

(1) Multi-user OS. 'Multi-user' means more than one user can access the same computer and its resources such as memory, printer, etc., at the same time. Here, the terminals having only monitors and keyboards that are connected with the main computer whose **resources** are availed by the user. Therefore, a user is allowed to sit on different terminals and access the main computer and its resources by giving his login name and password.

(2) Multitasking. 'Multitasking’ means the capacity of the Operating System to perform several tasks simultaneously. For example, it allows us to type a program in the editor, to print a document on the printer while simultaneously executing some other commands (programs).

UNIX does the multitasking by using Time-sharing technique. The OS allocates the CPU to a process for a period of time (time-slice). When this time-slice expires, the CPU is allotted to the next process that requires the CPU operation. After the completion of the waiting processes, the CPU is allotted to the first uncompleted process. There are also many scheduling algorithms

including Round-Robin, First-Come first-Serve, Shortest-Job-First, available for this multitasking.

(4) Security. UNIX provides the following types of security to prevent the unauthorized access:

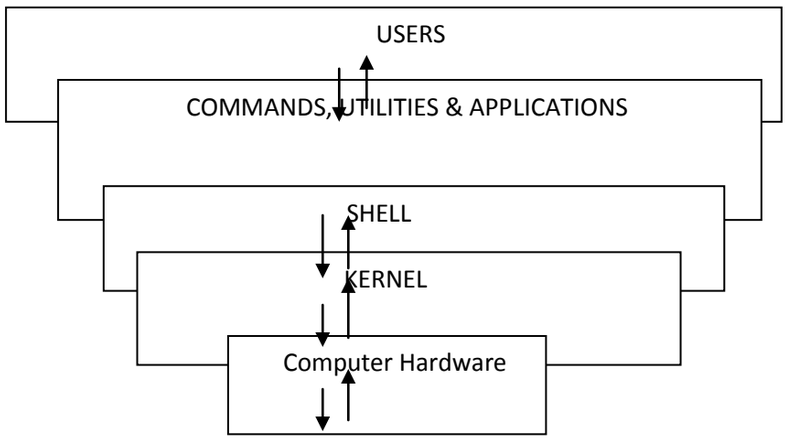
(i) Password to individual user. The system will log on for a particular user after verifying his username and password.

(ii) File accessing security. At the file level, there are Read, Write and execute permissions for each fide which deride who can access a particular file.

(iii) File encryption and decryption utilities. File-encryption makes our file in to unreadable format, so that even if someone succeeds in opening the file, he does not know the contents. File-decryption changes tiffs unreadable format into original format. communication. UNIX provides the communication facility to easily exchange mails, data and programs through network among its users. The communication may be online or *offline.* In online communication, the recipient user must be logged with the system. It is not necessary in the case of offline communication.

9.2 UNIX SYSTEM ORGANISATION

The following figure shows the organization of UNIX system:



Utilities and Applications. It represents user programs and commands for the execution.

Shell. It is the command interpreter which interprets the user commands and conveys them to the Kernel that executes them. Thus, it acts as a mediator between user and kernel.

Types of Shells. There are various types of Shells available in UNIX including *Bourne Shell, C Shell. Korn Shell, BASH Shell (Bourne-Again S-Hell), Z-Shell and TC Shell.* Different shells provide different user interfaces for Unix. The most frequently used shells are,

(i) *Bourne shell* It is the primary UNIX command interpreter. It comes along with every UNIX system and it is the default shell also . The prompt of this shell is dollar symbol ($) was created by Steve Bourne.

(ii) C Shell. The C Shell, created by Bin Joy, has some special facilities that are not available in Bourne Shell. The prompt of this shell is percentage symbol(%). We can shift in to this C Shell by using the "cash" command.

(iii)Korn shell It includes all the enhancements in the C shell like Command history and offers a few more features itself. David Korn of AT&T created it. The prompt of tiffs Shell is $. We can shift into Kern Shell by using the "ksh" in the command prompt.

Kernel. It is the heart of the UNIX system. It is a loaded into the memory whenever the system is booted. It interacts with the actual hardware in machine language. It manages resources such as files, memory, processor, I/o, ect., and it also does the functions such as *Keep tracking of a program in execution, Allotting the processor time to each process, etc.* Since the Kernel does not interact with the user directly, it uses Shell that will act as a mediator between the user and Kernel.

9.3 UNIX FILE SYSTEM

The file system refers a collection of files in UNIX. The UNIX file system looks like an inverter tree structure. The UNIX has a feature that it treats all the hardware devices as special files. The file system begins with a directory called root, denoted by the symbol /. From this root, UNIX has several special sub-directories each having different purposes.

bin: It contains executable files for the UNIX commands .

lib: It contains all the library files and functions provided by UNIX for programmers. Programs can be written in UNIX using these library functions by malting system calls.

dev: It contains device drivers (fries) that control various peripherals that are connected with the system like printers, disk-drivers, etc.

etc: It contains other additional commands that are related to system maintenance and administration.

tmp: It contains temporary files created by UNIX or by the users. These files are automatically deleted when the system is shutdown or restarted.

Usr: It contains several directions, each associated with a particular user.

Assignment Question:
1.what is time-sharing in Unix ? Explain.
2.Explain the system organization of UNIX

3.Explain the Unix file system ?

CHAPTER 10

LINUX:

INTRODUCTION

The LINUX(pronounced with a short I, as in LIH-nucks) is a free version of UNIX developed primarily by **Linus Torvals,** while a student at the University of Helsinki in Finland, with the help of hundreds of programmers scattered around the world across the internet for Intel based PCs. Linus takes MINUX, an UNIX-like operating system developed by Andrew Tanenbaun for Intel-based PCs, as inspiration. The source code of Linux itself is available on the internet. So, one can access the source to modify and expand the OS to satisfy yours needs. since Linux is an Unix-like kernel, all programs written for Unix can compiled and run under it but not vice versa.

10.1 REASONS FOR ITS POPULARITY

- It is a freely distributed implementation of an UNIX-like kernel.
- Since it is mostly written in 'c' language, it is flexible and portable.
- It supports multitasking: several programs running on the same machine at the same time, multiuser: several users working on the same machine at the same time and multiplatform: runs on many CPUs not just Intel.
- It is simply and consistent interface to peripheral devices.
- It is simple user interface, provides X-window system(or simply called as X).The X-Window system provides the foundations for the graphical user interface available with Linux.
- It provides data communication among users.
- It allows TCP/IP networking, including ftp, telnet, NFS, etc.
- It provides Multilevel File system. The Linux file structure is set up according to File Hierarchy Standard
- It also provides special file system called UMSDOS, which allows Linux to be in-stalled on a DOS file system.
- The machine architecture is hidden from the user.
- Most of the basic Linux utilities are GNU software. GNU utilities support advanced features that are not found in the standard versions of BSD and UNIX System V programs. (When UNIX had turned to commercial, Richard Stallman and Linus Torvalds strongly felt that it must be given away freely. Then, Stallman founded a Free Software Foundation formerly known as GNU -GNU's Not Unix.)
- It supports many national or customized keyboards, and it is fairly easy to add new ones dynamically.
- It allows Dynamic Link Libraries (DLL) and static libraries.
- It uses virtual memory using paging to disk (not swapping whole processes).

10.2 LINUX FILE SYSTEM

Linux treats all information as files. Hence, files form an important part of the LINUX system. The LINUX file system is a data structure - tree - that resides on the part of a disk. Almost the structure of the Linux file system is same as UNIX file system.

Bin	It contains all essential executable Linux program files
sbin	It contains executable files, which are used by super user
horne	It contains the subdirectories of the Linux users
etc	It contains the configuration files (administrative files) of all important programs
Var	It contains all your system logs
Usr	It is the central location for all your application program, x-windows, man pages, documents etc
Lib	It contains all the shared library files
tmp	It is the temporary file location
dev	It contains the special device files for keyboard, mouse, console, etc.

10.3 LOGIN AND LOGOUT

The process of entering into the LINUX system is called "login", and the complementary process of this is called **logout**. hen you start your terminal, it will display as follows: **login:** Type your login name here and press [Enter] key. If a password has been used by you, the system will wait for password by flashing the following message. **password:**

Type your password and press [Enter] key. The password would not be echoed on the terminal. Since Linux is case-sensitive, the passwords must be entered in correct cases. One can install or change password using the **"passwd"** command (Type 'passwd' without argument in the Linux prompt). By default, the Linux will display $ prompt for ordinary user and # prompt for super user.

To logout, press [Ctrl+d] keys (Ad)or type logout or exit in the command prompt. Behind the scenes, when you login, the following shell scripts will be executed automatically. They are used to set environment variables and other system settings.

/ ete / .profile: This is the first script file that is executed. This script is used to set Global parameters that are common to all users.

/ **home/<username>/.profile:** This is the next script file that is normally executed. It can be changed to set unique parameters for each user.

/home/<username>/.bashrc: This is the next script file that runs each time when you start a new shell.

Assignment Questions:

1. What is time-sharing in UNIX? Explain.
2. How does the Linux provide security to its user?
3. Explain the system organization of UNIX.
4. List any five Shells supported by Linux.
5. Discuss the uniqueness of Linux.
6. Explain: Linux File System.

CHAPTER 11

LINUX COMMANDS

11.1 COMMAND FORMAT

A command is an instruction given to the Shell; the Kernel will obey that instruction. Linux provides several commands for its users to easily work with it

The general format of a command is, *command options command arguments*

A command is normally entered in a line by typing from the keyboard. Even though the terminal's line width is 80 characters, the command length may exceed to 80 characters. The command simply overflows to the next line, though it is still in a single logical line. Commands, options and command arguments must be separated by white space(s) or tabes) to enable the system to interpret them as words. Options must be preceded by a minus sign (-) to distinguish them from command arguments. Moreover, options can be combined with only one minus sign.

For **Example**, you can use the command, *wc -1 -w -c a. c as "wc -lwc a. c .*

The command along with its options, command arguments is entered in one line. This line is referred as command line. A command line usually ends with a new-line character. The **command line** completes only after the user has hit the [Enter] key. The \ symbol placed at the end of a line continues the command to the next line, ignoring the hit of [Enter] key. Several commands may be written in a single command line. They must be separated by semicolon (;).

For **Example**, *$ date ; who*

The important Linux commands are grouped according to their functions and explained as follows.

- **Directory Oriented Commands**
- **File Oriented Commands**
- **Process Oriented Commands**
- **Communication Oriented Commands**
- **General Purpose Commands**
- **Pipes and Filters**

11.2 DIRECTORY ORIENTED COMMANDS

This command is used to list the content of the specified directory.
General format is,

ls [-options] <directory_name>

where options can be,
a Lists all directory entries including the hidden files

1 Lists the files in long format (filenames along with file type, file permissions, number of links, owner of the file, file size, file creation/modification time, number of links for a file). The number of links for a file refers more than one name for a file, does not mean that there are more copies of that file. This ls -l option displays also the year only when the file was last modified more than a year back. Otherwise, it only displays the date without year.

r Lists the files in the reverse order

t Lists the files sorted by the last modification time

R Recursively lists all the files and sub-directories as well as the files in the sub-directories.

p Puts a slash after each director

s Displays the number of storage blocks used by a file.

x Lists contents by lines instead of by columns in sorted order

F Marks executable files with (i.e the file having executable permission to the user) and directories with /

<directory _name> specifies a name of the directory whose contents are to be displayed.
If the <directory _name> is not specified, then the contents of the current directory are displayed.

Examples

```
[bmi@kousar bmi] $ Is
bmi desktop maxsizefile .sh testl .txt test2 .txt
[bmi@kousar bmi] $ Is - r
text testl .txt maxisizefile .sh Desktop bmi
test2 .txt polindrome .sh java e

[bmi$kousar bmi] $ Is - 1
total 36
drwxrwxr-x 4 bmi bmi 4096 Jan 10 15:36 bmi
drwxrwxr-x 2 bmi bmi 4096 Jan 10 12:11 e
drwxr-xr-x 2 bmi bmi 4096 Dee 11 2002 Desktop
drwxrwr-x 3 bmi bmi 4096 Jan 13 10 :03 java
```

WILD CARD CHARACTERS

,*, represents any number of characters.
'?' represents -a single character.
For **Example**,
*$ 18 pgm**
This command will list out all the file-names of the current directory, which are starting with "pgm". Note that the suffix to pgm may be any number of characters.

*$ 1s *8*
This command will display all the filenames of the current directory, which are ending with "s . Note that the prefix to s may be any number of characters.

$ is ?gms
This command will display four character filenames, which are ending with "gms" starting with any of the allowed character. Note that the prefix to gms is a single character.

"[ ] " represents a subset of related filenames. This can be used with range operator "-" to access a set of files. Multiple ranges must be separated by commas.

$ is pgm[1-5]
This command will list only the files named, pgml, pgm2, pgm3, pgm4, pgm5 if they exist in the current directory. Note that the [1·5] represents the range from 1through 5.

Examples
[bmi@kousar bmi] $ Is test?txt
textl .txt text2.txt test3.txt
[bmi@kousar bmi] $ Is test[l-2].txt
textl.txt text2 .txt

2. mkdir

This mkdir (make directory) command is used to make (create) new directories. General format is,

mkdir [-p] <directory_namel> <directory_name2>
The option -p is used to create consequences of directories using a single mkdir command.

Examples
$ mkdir ibr
This command will create 'ibr' a subdirectory of the current directory.
$ mkdir x x/y
This command will create x as a subdirectory of current working directory, y as subdirectory of x.

$ mkdir ibr/ib/i
This command will make a directory i as a subdirectory of ibr/ ib, but the directory structure -ibr / i must exist.

$ mkdir -p ibr / ib / i

Then, for the current directory, a subdirectory named *ibr* is created. Then, for the directory *ibr,* a subdirectory named *ib* is created. after that, the subdirectory i is created as a subdirectory of the directory ib.

3. rmdir

This rmdir (remove directory) command is used to remove (delete) the specified directories. A directory should be empty before removing it.
General format is,
rmdir [-p] <directory_namel> <directory_name2>

The option -p is used to remove consequences of directories using a single r*mdircommand.*
Example
$ rmdir ibr
This command will remove the directory ibr, which is the subdirectory of the current directory.

$ rmdir ibr/ ib / i
This command will remove the directory i only .

$ rmdir -p ibr / ib / i
This command will remove the directories i, ib and ibr consequently.

4. cd

This cd (change directory) command is used to change the current working directory to a specified directory.
General format is,
cd <directory_name>

Examples
$ cd / home /ibr
Then, the directory / home / ibr becomes as the current working directory.

$ cd ..
This command lets you bring the parent directory as current directory. Here, … represents the parent directory.

5. pwd

This *pwd* (print working directory) command displays the full pathname for the current working directory.
General format is,
pwd
Example
$ pwd
/ home / bmi

Your present working directory is */ home / bmi.*

6.find

This command recursively examines the specified directory tree to look for files matching some file attributes, and then takes some specified action on those files. General format is,

find <path_ list> <selection_criteria> <action>

It recursively examines all files in the directories specified in <path_list> and then matches each file for <*selection_criteria*>(file attributes). Finally, it takes the pacified <action> on those selected files.

The selection_criteria may be as follows,

-name <filename> Selects the file specified in <file name>. If wild-cards are used, then double quote <filename> -user <username> Selects files owned by <username> -typed Selects directories.

-size {+n –n} select files that are greater than/less than "n" blocks.(generally one block is 512 bytes).

-mtime {n +n -n} selects files that have been modified on exactly n days / more than n days / less than n days.

-min {n +n –n} selects files that have been modified on exactly on n minutes / more than n minutes / less than n minutes.

-atime {n +n –n} selects files that have been accessed on exactly n days / more than n days / less than n days.

-amin {n +n –n} selects files that have been accessed exactly n minutes / more than n minutes / less than n minutes

The **action** may be as follows,

-print : Displays the selected files on the screen

-exec <corrunand> : Executes the specified Linux command ends with {}\;

If <path_list> and <action> are not specified, then *current directory* and -print are taken respectively as default arguments.

Examples

*$ find / home / ibrahim -name " *. Java" -print*

This command will recursively displays all *.Java* files that are stored in the directory */ home / ibrahim* including all its sub-directories.

$ find / home / ibrahim -mtime 5 -print

If the current date is 20-02-2003, then this command will display the files that have been modified on 15-02-2003.

$ find / home / ibrahim -mtime +5 -print

If the current date is 20-02-2003, then this command will display the files that have been modified before 15-02-2003.
$ find / home / ibrahim -mtime -5 -print
If the current date is 20-02-2003, then this command will display the files that have been modified after 15-02-2003.

Making changes on a file means "modifying . Opening/ Modifying a file means "accessing

FILE ORIENTED COMMANDS
1. cat
This cat (catenated -concatenate) command is used to display the contents of the specified file(s).
General format is,
cat [-options] <filenamel> [<filename2> …]
where options can be,
s Suppresses warning about non-existent files
d Lists the sub-directory entries only
b Numbers non-blank output lines
n Numbers all output lines

Examples
$ cat a.c

This will display the contents of the file -*a.c.*
$ cat. a .c b.c
This will display the contents of the files, *a.c* and *b.c,* one by one.
This command can be used with redirection operator (>) to create new files.
General format is,
cat > filename
<Type the text>
^d (press [ctrl + d] at the end)

Example
$ cat > x .txt
Hi ! This
is a file. Press [^ d]

Then, a file named x. txt is created in the current working directory with 3 lines content.

2.cp
This cp (copy) command is used to copy the content of one file into another If the destination is an existing file, the file is overwritten; if the destination is an existing directory, the file is copied into that directory.

General format is,

cp [-options] <source-file> <destination-file>
where options can be,
i Prompt before overwriting destination files
p Preserve all information, including owner, group, permissions, and timestamps
R Recursively copies files in all subdirectories

Example
$ cp a.c b.c
The content of a.c is copied in to b.c ..

3.rm
This rm (remove) command is used to remove (delete) a file from the specified directory. To remove a file, you must have *write* permission for the directory that contains the file, but you need not have permission on the file itself. If you do not have write permission on the file, the system will prompt before removing.
General format is,
rm [-options] <filename>
where options can be,
r Deletes all directories including the lower order directories. Recursively deletes entire contents of the specified directory and the directory itself
i Prompts before deleting
f Removes write-protected files also, without prompting

Examples
$ rm a.c

This command deletes the file - *a.c* from the current directory. (But current directory will still exist with other files.)

$ rm -f / usr / ibrahim
This command deletes all the files and subdirectories of the specified directory */usr / ibrahim*. Note that the directory *'ibrahim* ' also will be deleted.

4. mv
This *mv* (move) command is used to rename the specified files / directories.
General format is,
mv <source> <destination>

Note that to make move, the user must have both write and execute permissions on the <source>.

Example

$ mv a.c b.c
Then, the file a.c is renamed to b.c.

5.wc
This command is used to display the number of lines, words and characters of information stored on the specified file.
General format is,
wc [-options] <filename>
where options can be,
1 Displays the number of lines in the file
w Displays the number of words in the file
c Displays the number of characters in the file

Examples
$ wc a.c
It displays the number of lines, words and characters in the file - *a.c.*
$ wc -1 a .c
It displays the number of lines in the file - *a.c.*

7.file
This command lists the general classification of a specified file. It lets you to know if the content of the specified file is ASCII text, C program text, data, separate executable, empty or others.
General format is,
file <filename>
[bmi@kousar bmi] $ file testl .txt
testl .txt: ASC II text
[bmi@kousar bmi] $ file test2.txt
test2.txt: ASCII text, with escape sequences
*[bmi@kousar bmi] $ file **
Desktop: directory
bmi: directory
c: directory
java: directory
shell: ASC II text
testl.txt : ASCII text
test2.txt: ASCII text , with escape
test3.txt : ASCII English text
text: directory

8. cmp
This *cmp* (compare) command is used to compare two files.
General format is,
cmp <filename1> <filename2>

This command reports the first instants of differences between the specified files. That is, the two files are compared byte by byte, and the location of the first mismatch is echoed to the screen.

Examples

[bmi@kousar bmi] $ cat filel.txt
I am ibrahim,
What is your name?
[bmi@kousar bmi] $ cat file2 .txt
I am ibrahim,
What are you doing?
[bmi@kousar bmi] $ cmp file l .txt file 2.txt
filel.txt file2 .txt differ : char 20, line 2

The file1.txt differs from file2.txt at 20th character, which occurs at the 2nd line.

FILE ACCESS PERMISSIONS

Linux treats everything as files. There are three types of files in Linux as follows,

- Ordinary file
- Directory file
- Special file (Device file)

The ordinary files consist of a stream of data that are stored on some magnetic media. A directory does not contain any data, but keeps track of an account of all the files and sub- directories that it contains. Linux treats even physical devices as files. Such files are called special files.

There are three types of modes for accessing these files as follows,

- Read mode (r)
- Write mode (w)
- Execute mode (x)

-rw-rw-r--1 bmi bmi 31 Dee 14 09 :39 test2 .txt

-rw-rw-r--1 bmi bmi 1484 Dee 16 15:42 typescript

-rw-rw-r--1 bmi bmi 93 Dee 16 16:14 userlist.txt

-rw-rw-r--1 bmi bmi 46 Dee 19 12:18 www

This command displays the last 10 directory listing of the current directory.

19. head

This command displays the top of the specified file.

General format is,
head [-n] <filename>
If -n option is specified, then the first n lines of the file are displayed. Default value for this option is 10.

Examples

```
[bmi@kousar bmi] $ head -3 employee.dat
1005 yasin computer es 15-08-1978
1002 abdulla zoology zoo 22-07-1956
1003 abdulla commerce com 25-11-1985

[bmi@kousar bmi] $ ls -1 I head

TOTAL 428
-rw-rw-r-- 1 bmi bmi 34520 Dee 24 10 :28 a
-rw-rw-r-- 1 bmi bmi 16481 Dee 24 10 :34 a1
-rw-rw-r-- 1 bmi bmi 19941 Dee 24 10 :34 a2
drwxrwxr-x 2 bmi bmi 4096 Dee 16 09:23 epp
-rwxr---- 1 bmi bmi 21 Dee 16 09 :54 d
drwxr-xr-x 2 bmi bmi 4096 Dee 11 11: 08 Desktop
-r--r--r-- 1 bmi bmi 21 Dee 11 09 :54 dq
-rw-rw-r-- 1 bmi bmi 213 Dee 18 12:29 e
-rw-rw-r-- 1 bmi bmi 25 Dee 19 12 :07 e1 .dat
```

PROCESS ORIENTED COMMANDS

A process is a job in execution. Since Linux is a multi-user operating system, there might be several programs of several users running in memory. There is a program called Scheduler always running in memory which decides which process should get the CPU time and when. Note that only one process will be executed at a time, because the system has only one processor (CPU).

Some of the process-oriented commands are given below.

1.ps

This command is used to know which processes are running at our terminal.
General format is,
ps -a: This command lists the processes of all the users who are logged on the system.
ps -t <terminal_name>: This command lists the processes, which are running on the specified terminal-<terminal_name>.
ps -u <user_name>: This command lists the processes, which are running for the specified user -<username>.
ps -x: This command lists the system processes. Apart from the processes that are generated by user, there are some processes that keep on running all the time. These processes are called system processes.

Examples
$ ps
$ ps -a
$ ps -t tty3d

This displays the processes, which are running on the *terminal- tty3d.*

$ ps -u ibrahirn

This displays the processes, which are running for the user - ibrahim.

BACKGROUND PROCESSING

Linux provides the facility for background processing. That is, when one process is running in the foreground, another process can be executed in the background. The ampersand (&) symbol placed at the end of a command sends the command for background processing.

Example
$ sort ernp .doc&
By the execution of this command, a number is displayed. This is called PID (Process IDentification) number. In Linux, each and every process has a unique PID to identify the process. The PIDs can range from 0 to 32767.

Then, the command 'sort emp.doc' will run on background. We can execute another command in foreground (as normal).

2.kill
If you want a command to terminate prematurely, press [ctrl + c].This type of interrupt characters does not affect the background processes, because the background processes are protected by the Shell from these interrupt signals. This kill command is used to terminate a background process.
General format is,

kill [-SignalNumber] <PID>

The PID is the process identification number of the process that we want to terminate. Useps command to know the PIDs of the current processes. By default, this *kill* command uses the signal number 15 to terminate a process. But, some programs like login shell simply ignore this signal of interruption, and continue execution normally. In this case, you can use the signal number 9 (often referred as sure kill).
Examples
$kill 120
This command terminate the process who has the PID 120.

$kill -9 130
$kill -9 0

This command kills all the processes including the login Shell. (The Kernel itself being the first process gets the PID O. The above command kills the Kernel, so all the processes are killed.)

4.at

This command is used to execute the specified Linux commands at future time.
General format is,

at <time>
<commands>
^d
(Press [ctrl + d] at the end)
Here, <time> specifies the time at which the specified <commands> are to be executed.

Example
$ at 12:00
echo "LUNCH BREAK"
^d
***at** offers the keywords - **now, noon, midnight, today** and **tomorrow** and they convey special meanings.*
$ at noon ← 12:00
echo "LUNCH BREAK"
at also offers the keywords hours, days, weeks, months and years, can be used with + operator as shown in the following **Example**s.
$ at 12 :00 + 1 day ← at 12:00 tomorrow
$ at 13 :00 Jan 20, 2003 + 2 days ← at 1pm January 20, 2004

5.batch

This command is used to execute the specified commands when the system load permits (when CPU becomes nearly free).
General format is,
batch
<commands>
^d
Any job scheduled with batch also goes to the at queue, you can list or delete them through *atq* and *atrm* respectively.

Example
$ batch
sort a.c

sort b.c
^d

COMMUNICATION ORIENTED COMMANDS

Linux provides the communication facility, from which a user can communicate with the other users. The communication can be online or offline. In online communication, the user, to whom the message is to be sent (recipient), must be logged on the system. In offline communication, the recipient need not be logged on the system.

1.write
This online communication command lets you to write messages on another user's terminal.
General format is,
write <RecipientLoginName>
<message>
^d
Example
$ wirte ibrahim
Hello ibrahim
How are you ?
^d
On the execution of this command, the specified message is displayed on the terminal of the user - ibrahim. If the recipient does not want to allow these messages (sent by other users) on his terminal, then he can use **" mesg n "** command. If he wants to revoke this option and want to allow anyone to communicate with him, then he can use **" mesg y "** command.
General format is,
mesg [y | n]
y Allows write access to your terminal.
n Disallows write access to your terminal.
The **mesg** without argument give the status of the **mesg** setting.
The "**finger"** command can be used to know the users who are currently logged on the system and to know which terminals of the users are set to **mesg y** and which are set to mesg n. A '*' symbol is placed on those terminals where the **mesg** set to n.

2. mail

This command offers off-line communication.
General format is,
To send mail,
mail <username>
<message>
^d

The mail program mails the message to the specified user. If the user (recipient) is logged on the system, the message you have new mail is displayed on the recipient's terminal. However, the user is logged on the system or not, the mail will be kept in the mailbox until the user issues the necessary command to read the mails.

To check mails, give the mail command without arguments. This command will list all the incoming mails received since the latest usage of the mail command. A & symbol is displayed at the bottom. This is called mail prompt. Here we can issue several mail prompt commands (referred as internal commands).

Some commands which can be given in mail prompt are given below,

Mail Prompt Commands	**Functions**
+	Displays the next mail message if exists
-	Displays the previous mail message if exists
<number>	Displays the <number>th mail message if exists
D	Deletes currently viewed mail and displays next mail message if exists
d <number>	Deletes the <number>th mail
s <filename>	Stores the current mail message to the file specified in <filename>
s<number> <filename>	Stores the <number>th mail message to the file specified in <filename>
R	Replies to the sender of the currently viewing mail
r <number>	Replies the <number>th mail to its sender
Q	Quits the mail program

3.Wall

Usually, this wall (write all) command is used by the super-user to send a message to all the users who were currently logged on the system.

General format is,

wall

<message>

<Press [ctrl + d) at the end>

Example

$ wall

Meeting at 16:00 hrs.

^d

The specified message "Meeting at 16:00 hrs." will be displayed on everyone's terminal with a beep sound like **write's** message, but ignoring mesg settings.

GENERAL PURPOSE CPMMANDS

1. date

This command displays the system's date and time.

General format is,
date +<format>
where <format> can be,
%H Hour - 00 to 23
%1 Hour - 00 to 12
%M Minute - 00 to 59
%s Second - 00 to 59
%D Date - MM / DD / YY
%T Time - HH:MM:SS
%w Day of the week
%r Time in AM / PM
%y Last two digits of the year

Examples

$ date
Mon Dee 16 15:13 :10 IST 2002
$ date +%H
15
$ date +%I
03
$ date +%M
13
$ date +%S
10
$ date +%D
12/16/02
$ date +%T
15 :13:10
$ date +%w
1 ← O=Sunday, l=Monday, 2=Tuesday, ...
$ date +%r
03:13:10 PM
$ date +%y
02

2. who
Since Linux is a multi-user operating system, several users may work on this system. This command is used to display the users who are logged on the system currently.
General format is,
who
Example
$ who
ibrahim tty1 Feb 19 10 :17

```
sheik       tty3      Feb      19    10:20
mukash      tty8      Feb      19    11:02
```

The first column of the output represents the user names. The second column represents the corresponding terminal names and the remaining columns represents the time at which the users are logged on.

3. who am i

This command tells you who you are. (Working on the current terminal)
General format is,
who am i

Example

```
$ who am i
ibrahim     tty1
```

4. man

This man (manual) command displays the syntax and detailed usage of the Linux command, which is supplied as argument.
General format is,

man <LinuxCommand>

Example

$ man we
This will display the help details for "we command.
Almost all of the commands offer --help option that displays a short listing of all the options.
$ we --help

5.cal

This command will display calendar for the specified month and year.
General format is,
cal [<month>] <year>

Where, month can be ranged from 1 to 12.

Example

$ cal 2002
This command will display calendar for the year 2002 (for 12 months).
$ cal 1 2003
This will display the calendar for the month - January of the year 2003.
$ cal 2 2003

```
February 2003

Su Mo Tu We Th Fr Sa
```

```
                        1

 2   3    4    5    6    7    8

 9  10   11   12   13   14   15

16  17   18   19   20   21   22

23  24  25   26   27   28
```

6.lpr

This command is used to print one or more files on printer.

General format is,

lpr [-options] <filename> <filename2> ... <filename N>

where options can be,

r Removes file(s) from directory after printing

m Mail inform you when printing is over

Example

$ lpr a.c

7.tee

This command does the operations of pipe and redirection. It will send the output of a command into standard output as well to a specified file.

General format is,

command | tee <filename>

Examples

$ ls

$ ls > list .doc

We can combine these two commands in to a single command (using tee) as follows,

$ ls I tee list .doc

The output of the command 'ls' is sent to the standard output device (screen) and also to the file -list.doc.

$ who | tee user list .txt | wc -1

This command stores the current user information in to the file named userlist.doc and displays the number of current users.

The **-a** option can be used with the tee command, which appends the output of the specified command into the specified file. Otherwise, the file content will be overwritten.

$ cat a.txt

This is a file named a.txt

$ cat b .txt

This is a file named b.txt
$ cat a .txt | tee result.txt
This is a file named a .txt
$ cat result .txt
This is a file named a .txt
$ cat b.txt | tee result.txt
This is a file named b.txt
$ cat result.txt
This is a file named b .txt
$ cat a .txt | tee -a result.txt
This is a file named a.txt
$ cat result.txt
This is a file named b.txt
This is a file named a.txt

8. script

This command stores your login session in to a specified file. All the operations that you have done (all the given commands, their outputs, error messages, ...) are stored on that file. Start the login session by issuing the following command.

$ script

Then do your operations, and finally use the exit command to end script. Then all the operations in between script and exit will be stored on a file named typescript.

The next script command will overwrite the typescript file. You can append instead of overwriting by using the script command with **-a** option.

$ script -a

Otherwise you can use other file-name instead of the default -typescript by specifying the filename as argument.

$ script actions.txt

Example

[bmi@kousar bmi] $ script actions.txt
Script started, file is actions .txt
[bmi@kousar bmi] $ cat test.txt
This is a test file.
[bmi@kousar bmi] $ we emp.doc
20 18 51 emp.doc
[bmi@kousar bmi] $ exit ← *Press[Enter]*
Then, the content of actions.txt is,
Script started on Fri Jun 6 16:21:20 2003
[bmi@kousar bmi]$ cat test .txt

This is a test file .
bmi@kousar bmi] $ wc emp.doc
20 18 51 emp.doc
[bmi@kousar bmi] $ exit
Script done on Fri Jun 6 16:21:44 2003

9. tput
This command with **clear** option can be used to clear the screen content.
General format is,
tput clear
You can also use clear command for this purpose.

Example
$ tput clear

10 split
If a file is very large, then it cannot be edited on an editor. In such situations, we need to split the file in to several small files. For this purpose, this split command is used.
General format is,
split -<number> <filename>

The -<number> option splits the file specified in <filename> into <number> lined files. Default for this option is 1000 lines.
The splitted contents are stored in the file names xaa, xab, xac, ... ,xaz, xba, xbb, xbc, ... ,xzz (totally 676 filenames).

Example
$ split test .txt
If the specified file - test.txt contains 5550 lines, then this command creates the files namely xaa, xab, xac, xad, xae containing 1000 lines and xaf containg remaining 550 lines.

$ split -500 test .txt
This command splits the test.txt file into 500 lined files.

11.expr
This command is used to perform arithmetic operations on integers.
The arithmetic operators and their corresponding functions are given below.
+ Addition
- Subtraction
* Multiplication
/ Division (Decimal portion will be truncated. Because it perform division operation on integers only. It gives only quotient of the division.)
% Remainder of division (modulus operator)

A white space must be used on either side of an operator. Note that the multiplication operator (*) has to be escaped to prevent the Shell from interpreting it as the filename meta character. This command only works with integers, so, the division yields only integer part.

Examples

$ x=5
$ y=2
Then,
$ expr $x + $y
7
$ expr $x - $y
3
*$ expr $x * $y*
10
$ expr $x / $y

2 ← Note that decimal portion is truncated.

$ expr $x % $y

1 ← *Remainder part of division.*

$ expr $x + 10

15

12.bc

This command is used to perform arithmetic operations on integers as well as on floats (decimal numbers).
Type the arithmetic expression in a line and press [Enter] key. Then the answer will be displayed on the next line. Ater you have finished your work, press [Ctrl + d] keys to end up.

Examples

Add 10 and 20.
$ bc
10 + 2 0 ← arithmetic expression
30 ← result is-displayed on the next line
Press ^d to end the work.

Divide 8 by 3.
$ bc
8 / 3

2 ← Decimal portion is truncated.

$ bc
a=8
b=3
c=a/b
c
2

By default, bc performs truncated division (integer division). If you do not want to truncate any value (requiring float division), then you have to set scale to the number of digits of precision before the operation.

$ bc
scale=l
813
2.6
scale=2
8 / 3
2.66
^d

This bc command can be used with ibase (input base) and obase (output base) to convert numbers in one bash into another.

$ bc
ibase=2 ← Set ibase to binary (2)
101 ←Type the input binary number
5 ←Result in decimal
$ bc
obase=2 ← Set obase to binary
5 ←Type the input decimal number
101 ← Result in binary
$ bc
ibase = 16 ← Set ibase to hexadecimal
B ←Type the input hexadecimal number
11 ←Result in decimal
$be
Obase = 16 ← Set obase to hexadecimal
11 ←Type the input decimal number
B ← Result in hexa-decimal

PIPES AND FILTERS

"pipe" is a mechanism in which the output of one command can be redirected as input to another command.
General format is,
Command 1 | command 2
The output of command 1 is sent to command2 as input.

Example

$ ls | more

The output of the command **'ls'** is sent to the **'more'** command as input. So, the directory listing of the current directory is displayed is displayed page (with pause). The following two Command group display the total number of user who are currently loged on the system.

$ who > userlist.txt
$ wc -l userlist.txt

REDIRECTION

Linux treats the keyboard as the standard input (value 0) and terminal screen as standard output (value 1) as well as standard error (value 2). However, input can be taken from sources other than the keyboard and output can be passed to any sources other than the terminal screen. Such a process is called "**redirection"**

Redirecting inputs:
The '<' symbol is used to redirect inputs.

Example

$ cat < filel.txt

Then the file - filel.txt is taken as input for the command --cat.

Redirecting outputs:
The > symbol is used to redirect outputs.

Example

$ ls > list.doc

Then, the output of the command 'ls' is stored on the file 'list.doc'. We can also use '1>' instead of '>'.

Redirecting Error messages:
The '2>' symbol is used to redirect error messages.

Example

$ cat lis11 . doc

If there is no file named 'listl.doc ' in the current directory, then the error message is sent to the standard error device (usually screen). We can redirect this error message using '2>' symbol.

$ cat listl.doc 2> errormes .txt

Then, the error message, if generated, will be stored on the file - errormes.txt.

FILTERS

There are some Linux commands that accept input from standard input or files, perform some manipulation on it, and produces some output to the standard output. Since these commands perform some filtering operations on data, they are appropriately called as "filters ". These filters are used to display the contents of a file in sorted order, extract the lines of a specified file that contains a specific pattern, etc.

1. sort

This command sorts the contents of a given file based on ASCII values of characters.
General format is,
sort [options] <filename>
where options can be,

-m <file list>	Merges sorted files specified in <filelist>
-0 <filename>	Stores output in the specified <filename>
-r	Sorts the content in reverse order (reverse alphabetical order)
-u	Removes duplicate lines and display sorted content
-n	Numeric sort
-t "char "	Uses the specified "char " as delimiter to identify fields
-c	Checks if the file is sorted or not
+pos	Starts sort after skipping the pos^{th} field
-pos	Stops sorting after the pos^{th} field
+pos .n	Starts sort after the nth column of the $(pos+l)^{th}$ field
-pos .n	Stops sort on the nth column of the $(pos+l)^{th}$ field

Example

```
$ cat empname. txt
 yasin
 abdulla
 ibrahim
 abdulla
 ismail
 $ sort empname. txt
 abdulla
 abdulla
 ibrahim
 ismail
 kadar
 yasin
```

This command displays sorted contents of the file -empname.txt on the screen.

```
$ sort -r empname .txt
yasin
kadar
```

ismail
ibrahim
abdulla
abdulla

This command displays the sorted content, sorted in reverse alphabetical order, of the tile - empname.txt.

$ sort empname .txt -0 result .txt
$ cat result .txt
abdulla
abdulla
ibrahim
ismail
kadar
yasin

This command stores the sorted contents of "empname.txt" in to "result.txt".
$ cat empnamel.txt
yasin
abdulla
Ibrahim

uniq
This uniq (unique) command is used to handle duplicate lines in a file. If this command used without any option, it displays the lines by eliminating duplicate lines.

General format is,
uniq [-option] <filename>
where options can be,
u Displays only the non-repeated lines
d Displays only the duplicated lines
c Displays each line by eliminating duplicate lines, and prefixing the number of times it occurs.

Examples
[bmi@kousar bmi] $ cat employee .dat Ii

1005	yasin	computer	cs	15-08-1978
1002	abdulla	zoology	zoo	22-07-1956
1002	abdulla	zoology	zoo	22-07-1956
1006	ibrahim	computer	cs	18-09-1987
1006	ibrahim	computer	cs	18-09-1987
1006	ibrahim	computer	cs	18-09-1987
1003	abdulla	commerce	com	25-11-1985
1001	ismail	botany	bot	28-03-1965

```
1004        kadar        computer        cs        23-05-1988

[bmi@kousar   bmi]  $ uniq   employee.dat

1005        yasin        computer        cs        15-08-1978
1002        abdulla      zoology         zoo       22-07-1956
1006        ibrahim      computer        cs        18-09-1987
1003        abdulla      commerce        com       25-11-1985
1001        ismail       botany          bot       28-03-1965
1004        kadar        computer        cs        23-05-1988

[bmi@kousar   bmi]  $ uniq   -u  employee.dat
1005        yasin        computer        cs        15-08-1978
1003        abdulla      commerce        com       25-11-1985
1001        ismail       botany          bot       28-03-1965
1004        kadar        computer        cs        23-05-1988
[bmi@kousar   bmi]  $ uniq   -d  employee.dat
1002        abdulla      zoology         zoo       22-07-1956
1006        ibrahim      computer        cs        18-09-1987
[bmi@kousar   bmi]  $ uniq   -c  employee.dat
1   1005    yasin        computer        cs        15-08-1978
2   1002    abdulla      zoology         zoo       22-07-1956
3   1006    ibrahim      computer        cs        18-09-1987
1   1003    abdulla      commerce        com       25-11-1985
1   1001    ismail       botany          bot       28-03-1965
1   1004    kadar        computer        cs        23-05-1988
```

4. more

If the information to be displayed on the screen is very long, it scrolls up on the screen fastly. So, the user cannot be able to read it. This more command is used to display the output page by page (without scrolling up on the screen fastly). Use [spacebar] or [f] key to scroll forward one screen, use [b] key to scroll backward one screen, use [q]key to quit displaying.

General format is,

more <filename>

Example

$ more a.c

This command will display the content of the file -a. c page by page.

$ ls | more

This command will display the directory listing page by page.

You can also use more specialized command -less, instead of more. This less command has the same syntax and functions of more.

$ less a .c

$ ls | less

Note: For this paging purpose, you can also use key combinations. The [ctrl + s] is used to stop scrolling of the screen output. The [ctrl + q] is used to resume scrolling of the screen output. But it is not an ideal method.

5.pr

This command displays the contents of the specified file adding with suitable headers and footers. This command can be used with *lpr* command for neat hard copies. The Header part consists of the last modification date and time along with file-name and page number.
General format is,

pr [-options] <filename>
where options can be,
-1 <number> It changes the page size to specified <number> of lines (By default, the page size is 66 lines)
-<number> Prepares the output in <number> columns
-n Numbers lines
-t Turns off the heading at the top of the page

Examples

```
[bmi@kousar bmi] cat employee.dat
1005 yasin computer cs 15-08-1978
1002 abdulla zoology zoo 22-07-1956
1003 abdulla commerce com 25-11-1985
1001 ismai1 botany bot 28-03-1965
1004 kadar computer cs 23-05-1988
[bmi@kousar bmi] premployee .dat
2002-12-19 10: 13 employee.dat Page 1
1005 yasin computer cs 15-08-1978
1002 abdulla zoology zoo 22-07-1956
1003 abdulla commerce com 25-11-1985
1001 ismail botany bot 28-03-1965
1004 kadar computer cs 23-05-1988
```

[bmi@kousar bmi] $ pr -n employee.dat

```
2002-12-19 10: 13 employee .dat Page 1
1 1005 yasin computer cs 15-08-1978
2 1002 abdulla zoology zoo 22-07-1956
3 1003 abdulla commerce com 25-11-1985
4 1001 ismail botany bot 28-03-1965
5 1004 kadar computer cs 23-05-1988
```

-Note that remaining lines are empty-

```
[bmi@kousar bmi] $ pr -2 employee.dat
2002-12-19 10 :13 employee.dat Page 1
1005 yasin computer cs 15-08-1978
1002 abdulla zoology zoo 22-07-1956
1003 abdulla commerce com 25-11-1985
-Note that remaining lines are empty-
```

6. cut

This command is used to cut the columns I fields of a specified file (Like the head and tail commands cut the lines - rows).

General format is,

cut [-options] <filename>

where options can be,

*c<columns>*Cuts the columns specified is<columns>.You must separate the column numbers by using commas

*f <fields>*Cuts the fields specified in <fields>. You must separate field numbers by using commas

Examples

```
[bmi@kousar bmi] $ cat employee .dat
1005 yasin computer cs 15-08-1978
1002 abdulla zoology zoo 22-07-1956
1003 abdulla commerce com 25-11-1985
1001 ismail botany bot 28-03-1965
1004 kadar computer 23-05-1988
[bmi@kousar bmi] $ cut -f 3 employee.dat
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] cut -f 3-5 employee.dat
computer cs 15-08-1978
zoology zoo 22-07-1956
commerce com 25-11-1985
botany bot 28-03-1965
computer cs 23-05-1988

[bmi@kousar bmi] $ cut -f 1-3,5 employee .dat

1005 yasin computer 15-08-1978
1002 abdulla zoology 22-07-1956
1003 abdulla commerce 25-11-1985
```

1001 ismail botany 28-03-1965
1004 kadar computer 23-05-1988
[bmi@kousar bmi] $ cat e.dat
1005| yasin |computerlcs|15-08-1978
1002|Abdulla | zoology | zoo|22-07-1956
1003Iabdullalcommerce|com|25-11-1 985
1001 Iismail!botanylbot|28-03-1965
1004|kadarlcomputer|cs|23-05-1988

[bmi@kousar bmi] $ cut -d"|" -f 3-4 e .dat
Computer|cs
Zoology|zoo
commerce |com
botany|bot
computer|cs
[bmi @kousar bmi] $ cut -c 1-4 e .dat
1005
1002
1003
1001
1004
[bmi@kousar bmi] $ cut -c 6-10,15,17 e .dat
yasinpt
abdulol
abdulom
ismaitn
kadarpt

7. paste

This command concatenates the contents of the specified files into a single file vertically. (Like cut command separates the columns, this paste command merges the columns). General format is,

paste <filenamel> <filename2> ...

Examples

[bmi@kousar bmi] $ cat e1 .dat

1005
1002
1003
1001
1004
[bmi@kousar bmi] $ cat e2.dat
computer
zoology

commerce
botany
computer
[bmi@kousar bmi] $ paste
1005 computer
1002 zoology
1003 commerce
1001 botany
1004 computer

8.tr

This tr (translate) command is used to change the case of alphabets.
General format is,
tr <CharacterSetl> <CharacterSet2> <StandardInput>
This command translates the first character in the <CharacterSetl> into the first charter in the <CharacterSet2> and this same procedure is continued for remaining characters that this command gets input from standard input not from a file. But you can use pipe or redirection to use a file as input.

Examples

[bmi@kousar bmi] $ cat e2 .dat
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] $ cat e2.dat I tr "[a-z]" "[A-Z]"
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] $ tr "e,o" "C,O" < e2 .dat
computer
zoology
Commerce
botany
computer
[bmi@kousar bmi] $ tr -d "c,o" < e2 .dat
computer
zlgy
mmerce

The tr command used with **-d** option deletes the characters specified in <CharacterSet> from input does not translate.

9.sed

This command acts as a line editor.

General format is

sed 'EditComrnands' <filename>

where EditCommands can be

i Inserts before line
a Appends after line
c Changes lines
d Deletes lines
p Prints lines
q Quits

s / string1 / string2 Substitutes string l by string2

Examples

```
[bmi@kousar bmi]    $  cat  employee.dat
1005        yasin        computer        cs          15-08-1978
1002        abdulla      zoology         zoo         22-07-1956
1003        abdulla      commerce        corn        25-11-1985
1001        ismail       botany          bot         28-03-1965
1004        kadar        computer        cs          23.-05-1988
[bmi@kousar bmi]      sed  '2q' employee.dat
1005        yasin        computer        cs          15-08-1978
1002        abdulla      zoology         zoo         22-07-1956
```

This command displays the first two lines of the employee.dat.

```
[bmi@kousar bmi]    $ sed  '2q' employee.dat
1005        yasin        computer        cs          15-08-1978
1003        abdulla      commerce        com         25-11-1985
1001        ismail       botany          bot         28-03-1965
1004        kadar        computer        cs          23-05-1988
```

This command displays the content of employee.dat by deleting the second line.

```
[bmi@kousar bmi]    $  sed  -n  '2q' employee.dat
1002        abdulla      zoology         zoo
```

This command displays the second line of the employee.dat.

```
[bmi@kousar bmi]    $  sed  -n  '2,5p' employee.dat
1002        abdulla      zoology         zoo         22-07-1956
1003        abdulla      commerce        corn        25-11-1985
1001        ismail       botany          bot         28-03-1965
1004        kadar        computer        cs          23-05-1988
```

This command displays the lines 2 through 5 of employee.dat. Use -n option if you use the p edit command.

VI EDITOR

Linux offers various types of editors like **ex**, **sed**, **ed, vi, vim, xvi, nvi, elvis etc**., to create and edit your files (data files, program files, text files etc.). The famous one is vi editor (visual full screen editor) created by Bill Joy at the University of California at Berkeley.

Starting VI

This editor can be invoked by typing vi at the $ prompt. If you specify a filename as an argument to vi, then the vi will edit the specified file, if it exists.

vi [<filename>]

A status line at the bottom of the screen (25th line) shows the filename, current line and character position in the edited file.

vi +< linenumber > <filename>

- Edits the file specified in <*filename*> and places the cursor on the <linenumber>th line.

VI MODES

The vi editor on three modes as follows:

INSERT MODE:

- The text should be entered in this mode. And any key press in this mode is treated as text.
- We can enter into this mode from command mode by pressing any of the keys:

i, I, a, A, o, O, r, R, s, S.

COMMAND MODE:

- It is the default mode when we start up vi editor.
- All the commands on vi editor (cursor movement, text manipulation, etc.) should be used in this mode.
- We can enter into this mode from Insert mode by pressing the [Ese] key, and from Ex mode by pressing [Enter] key.

EX MODE:

- The ex mode commands (saving files, find, replace, etc.,) can be entered at the last line of the screen in this mode.
- We can enter into this mode from command mode by pressing [:] key.

The following are some of the commands that should be used in command mode.

INSERT COMMANDS:

i	Inserts before cursor
I	Inserts at the beginning of the current line (the line at which the cursor is placed)
a	Appends after cursor
A	Appends at the end of the current line
o	Inserts a blank line below the current line
O	Insert a blank line above the current line

DELETE COMMANDS:

x	Deletes a character at the cursor position
<n> x	Deletes specified number (n)of characters from the cursor position
X	Deletes a character before the cursor position
<n> X	Deletes specified number (n)of characters before the cursor position
dw	Deletes from cursor position to end of the current word. It stops at any punctuation (eg. " , .) that appears with the word
dW	Same as "dW'; but ignores any punctuation that appears with the word
db	Deletes from cursor position to beginning of the current word. It stops at any punctuation that appears with the word
dB	Same as "db"; but ignores any punctuation that appears with the word
dd	Deletes current line
<n>dd	Deletes specified number of lines (n)from the currentline
d[Enter]	Deletes current line and the following line
d0	Deletes all the characters from the beginning of the current line to previous character of cursor position
D	Deletes all the characters from the current character to the end of the current line I
d/<pattern>/	[Enter] I Deletes all characters but before to specified **pattern** occurs
d)	Deletes all the characters from current cursor position to end of the current sentence .
d	Deletes all the characters from current cursor position to the beginning of the current sentence
d}	Deletes all the characters from the current cursor position to the end of the current paragraph
d{	Deletes all the characters before the cursor position to beginning of the current paragraph

REPLACE COMMAND

r	Replaces single character at the cursor position
R	Replaces characters until [Ese] key is pressed from current cursor position
s	Replaces single character at the cursor position with any number of characters
S	Replaces entire line

CURSOR MOVEMENT COMMANDS:

H or [back space]	Moves cursor to the left (left arrow)
L or [space bar]	Moves cursor to the right (right arrow)

K	Moves cursor to up (up arrow)
J	Moves cursor down (down arrow)
W	Forwards to first letter of next word; but stops at any punctuation that appears with the word
B	Backwards to first letter of previous word; but stops at any punctuation the appears with the word
E	Moves forward to the end of the current word; but stops at any punctuation that appears with the word
W	Same as w; but ignores punctuation that appears with the word
B	Same as b; but ignores punctuation that appears with the word
E	Same as e; but ignores punctuation that appears with the word
[Enter]	Forwards to beginning of next line
0	Moves to the first location of the current line
^	Moves to the first character of the current line
$	Moves to the last character of the current line
H	Moves to the first character (left end) of top line on the current screen
M	Moves to the first character (left end) of middle line on the current screen
L	Moves to the first character (left end) of lowest line on the current screen
G	Moves to the first character of the last line in the current file
<n>G	Moves to the first character of the specified line (n) in the current file
(	Moves to the first character of the current sentence
)	Moves to the first character of next sentence
{	Moves to the first character of current paragraph
}	Moves to the first character of next paragraph

SEARCH COMMANDS:

/string[Enter]	Searches the specified string forward in the file
?string[Enter]	Searches the specified string backward in the file
N	Finds the next string in the same direction (specified by the above commands)
N	Finds the next string in the opposite direction (specified by the above commands)

YAKING COMMANDS:(COPY & PASTE)

yy (or) Y	Yanks the current line in to the buffer (copy)
nyy (or) nY	Copies the 'n' lines from the current line to the buffer
p	Pastes the yanked text below the current line (below the cursor)
p	Pastes the yanked text above the current line (above the cursor)

REDO COMMAND:

. (period)	Reapeats the most recent editing operation performed. Note that it is not Applicable to cursor movement commands.

UNDO COMMAND:

U	Undoes the most recent editing operation performed.

For **Example**, assume that you have deleted a line currently, if you use this undo command, then the deleted line will be displayed (undone the delete operation).

Forward
SCREEN COMMANDS:

Ctrl F	Scrolls full screen (screen of text - page) forward
Ctrl B	Scrolls full screen backward
Ctrl D	Scrolls half screen forward
Ctrl U	Scrolls half screen backward
Ctrl G	Display the status (filename , total number of lines in the file , current line number, percentage of file that precedes the cursor) on the status line (at bottom of the screen)

SHELL PROGRAMMING

The shell is a Linux system mechanism for the communication between the users and the system. And it is a command interpreter that interprets the user commands and conveys them to the kernel, which executes them. There are many Shells available in Linux including Bash (Bourne again shell) Shell, Bourne Shell, C Shell (tcsh), Korn Shell. (public domain korn shell- pdksh). Among them, the most popular one is bash, the super set of the Bourne Shell. This bash Shell is compatible with Bourne. The prompt of this bash Shell is $ symbol.

And we have already seen several commands, which can be used in this Shell. Let us discuss the features of this Shell in detail. If a sequence of commands is to be used repeatedly, we can assign them permanently in a file. This file is called Shell script. These scripts (files) act like batch files in Disk Operating System(DOS). Note that the name of the Shell script must not be a Linux command name.

Shell script

The Shell script can be executed in two methods. One method is using the sh command. Another one is to grant executable permission to the file using chmod command and then type the filename at the $ prompt.
General format of **sh** command is,
sh [-v] <script_filename>

The **-v** option causes the Shell to echo (display) each command before it is executed.

Example
Store the following commands in the file named **disp.sh.**
pwd
date
Linux does not require file extension. A file extension **.sh** may be used for user's convenience.
To execute this script (file) using sh command,
[bmi@kousar bmi] $ sh disp.sh
/home/bmi
Fri Dee 27 10:10:43 IST 2002
(Results of the commands -pwd and date - on the disp.sh are displayed. The **sh** command creates a child shell process. This new Shell reads the commands from the specified file - disp.sh, executes them and returns the control to the parent Shell.

[bmi@kousar bmi] $ sh -v disp.sh
pwd
/home/bmi
date
Fri Dee 27 10:09:22 IST 2002

To execute the script by giving execute permission, use the following commands.
[bmi@kousar bmi] $ chmod u+x disp.sh
[bmi@kousar bmi] $.\ disp.sh
/home/bmi

Fri Dee 27 10:11:05 IST 2002
This chmod command gives executable permission to user. Then disp.sh becomes executable and acts as a command. To execute a command that is stored in the current directory, we must include a • (**.** represents current directory) in the PATH Shell variable. Because the Shell first searches the working directory for executing a command, only when a dot (.) is specified at the start of the PATH variable. But it is not advisable to include a dot in the beginning of the PATH variable, because it may execute a **Trojon Horse** program (a program that has the same name as a standard Linux command that causes severe consequences). So, if a program has to be run in the working directory, it can be performed by preceding the command name with **. / ,** for instance **./** disp.sh

COMMAND GROUPING
We can combine several commands using semi-colon instead of executing one by one. General format is,
Command l ; command2 ; …
where; is the command separator.

Examples

[bmi@kousar bmi] $ pwd ; date
/ home / bmi
Fri Dee 27 14:52:52 IST 2002

First the pwd command is executed, and then the command date is executed.

[bmi@kousar bmi] $ (pwd ; date) > resu1t.txt
[bmi@kousar bmi] $ cat resu1t.txt
/home/bmi
Fri Dee 27 14:53:19 1ST 2002

The above command redirects the outputs of the pwd & date command into the file named result.txt.

The Shell variables are classified into two types as follows:
(i) Build-in Shell variables
(ii) User-defined Shell variables

Build-in Shell Variables:

These variables are created and maintained by the Linux system. These variables are used to define an environment. So, it is also called as "environmental variables". An environment is an area in memory where we can place definitions of Shell variables so that they are accessible from the programs. We can list these system variables (built-in variables) corresponding to a user, using the "set" command.

Example

[bmi@kousar bmi] $ set
BASH=/bin/bash
BASH_ENV=/home/bmi/.bashre
BASH_VERS1NFO=([O]="2 1]="04* [2]="21* [3]="1**
BASH_VERS10N='2.04 .~1(1,-re1ease '
COLORS=/ete/D1R_COLOUR
COLUMNS = 97
D1RSTACK= ()
EU1D=502
GROUPS= ()
H1STF1LE=/home/bmi /.bash_history
H1STF1LES1ZE=1000
H1STS1ZE=1000
HOME=/home /bmi
HOSTNAME=kousar
H1STTYPE=i386
1FS='
1NPUTRC= / ete / inputre

KDED1R= / usr
LANG = en_US
LESSOPEN=' l/ usr / bin / 1esspipe.sh %s'
L1NES=24
LOGNAME=bmi

Left side of the "=" sign is the system variable, right sides of the "=" sign is its corresponding value. In Linux, the system variables are specified by uppercases for convenience. Some of the build-in (system) variables are given below.

System Variables	Meanings
HOME	Contains the path name of home directory (where your login Shell is initially located after logged in)
LOGNAME	Contains user's login name
PATH	Contains the directories in which the Shell will search for files to execute the commands that are given by the users
MAIL	Contains the name of the directory in which electronic mails addressed to the user are placed
IFS	A list of characters that are used to separate words in a command line including[space], [tab],[newline]character(IFS - Internal Field Separators)
EXINIT	Initialization instructions for the vi and ex editors
PSl	Prompt String 1,normally PSI is "$"
PS2	Prompt String 2, it is used when Linux thinks that you have started a new line without hitting [Enter] key, normally PS2 is ">".
TERM	Determines the terminal type being used

User-defined Shell Variables:
These variables are created by the users with the following syntax,
<variable_name>=<value>
Note that there must not be a space on either of the equal sign.

Rules for Naming Shell Variables:

(i)The variables must begin with a letter or underscore character. The remaining characters may be letters, numbers or underscores.
(ii) No spaces are allowed on either side of the equal sign.
(iii) If the <value> contains blank-spaces, then it should be enclosed within double quotes.

Examples
$ netpay=14000
Then, the value 14000 is assigned to the variable *netpay.*
$ name="MOHAMEDIBRAHIM"

echo

This command is used to display values on the screen.
General format is,
echo [<string> $variable_name]
The symbol $ prefixed to a variable gives the value of that variable. The echo without any argument produces an empty line.
Example
[bmi@kousar bmi] $ echo Hi
Hi !

[bmi@kousar bmi] $ echo MOHAMED IBRAHIM
MOHAMED IBRAHIM
[bmi@kousar bmi] $ netpay=14000
[bmi@kousar bmi] $ echo $netpay 14000
[bmi@kousar bmi] $ echo name name
[bmi@kousar bmi] $ echo $name
[bmi@kousar bmi] $ name="MOHAMED IBRAHIM"
[bmi@kousar bmi] $ echo $name
MOHAMED IBRAH IM
[bmi@kousar bmi] $ echo Hi $name , your salary is $netpay
Hi MOHAMED IBRAHIM, your salary is 14000

Whenever the Shell finds continuous spaces and tabs in the command line, it compresses them to a single space. That's why, when you issue the following echo command, you find the output to be compressed.
[bmi@kousar bmi] $ echo MOHAMED
MOHAMED IBRAHIM

To preserve the spaces, you have to place the string within quotes.
[bmi@kousar bmi] $ echo "MOHAMED
MOHAMED IBRAHIM

The echo command will generate a carriage return by default. You can use the echo command with -n option to avoid such automatic generation of the carriage return.
[bmi@kousar bmi] $ echo "Dear" ; echo "ibrahim"
Dear
ibrahim
[bmi@kousar bmi] $ echo -n "Dear" ; echo "ibrahim "
Dear ibrahim
The -e option of the echo command enables the interpretation of the following charac. ter sequences (escape sequences) in the argument string.
Escape Meanings

sequences

\ b	back space
\ n	new line
\ r	carriage return
\t	tab
\ a	alert(beep sound)
\\	back slash
\ ‘	single quote
\ “	double quote

SHELL META CHARACTERS

The Linux separates certain characters for doing special functions. These characters are called "Shell meta characters” . Some of them are listed below.

< or 0<	Redirect for standard input from a specified file command < file name
> or 1>	Redirect standard output into the specified the command > filename
2>	Redirect standard error into the specified the command 2 > filename
>>	Appends standard output into the specified file command < filename
<	Takes standard output into the specified file command > filename
|	Connects standard output of one command (c1) into standard input of another command (c2) C1 | c2

Pattern Matching Characters:

We can use the wild card characters (*, ?) in filenames to represent a group of files.

?	Matches any single character in filename. For **Example**, a?c represents three character filename(s) starting with 'a', ending with 'e' and the middle may be any character
*	Matches any string of zero or more characters in filename(s) For **Example**, a *e represents the filename(s) started with 'a', ending with 'e' and the middle may be any combination of characters of any length.
[character_list]	Matches any single specified character (specified in character_list) in filenames. For **Example**, a [be] d represents the filename abe or aed.
[c1-c2]	Matches a single character that is within the ASCII range of char acters c1 and c2. For **Example**, a [b-e] k represents the filenames starting with the letter 'a', ending with the letter 'k', and the middle character is b, e, d or e.
[character_list]	Matches any single character that is not specified in character_list

Command Terminating Characters:

	Separates the commands when more than one command is given in a line. commandl ; command2 Executes the command and then"executes command2.
&	Like; character, but does not wait the previous command to complete. *commandl & command2* Unlike like the; character, the command2 will not wait for the completion of commandl.

Comment Charactes:

#	If an # symbol appears in a line, then the rest of the line will be treated as comment For **Example**, # This is a comment

&&	commandl && command2 Executes commnadl, if successful executes command2.
| |	commandl I I command2 Executes commandl, if failure executes command2.

Control Statements

Shell provides some control statements to execute the commands based on certain conditions.

if *if (conditional_command)*
then
(format 1) *<commands>*

fi

if *if (conditional_command)*
then
(format 2) *<commands>*
else
<commands>

if *if (conditional_command)*
then
(format 3) *<commands>*
elif (conditional_command)
then
<commands>
.
.
.

else
<commands>
fi

If the specified *<conditiona_command>* is executed successfully (exit status is 0), then the commands in between then and else (if block) will be executed: Otherwise, if the specified *<conditionaCcommand>* is not executed successfully (exit status is nonzero), then the commands in between else and fi (else block) will be executed.

Consequences of if. else statements may be written using a if. elif statement.

Example

```
if grep printf a.c > /dev/null
then
echo "The pattern -printf is found in the file -a.c ."
else
echo "The pattern -printf is not found in the file -a.c. "
fi
```

Here, if the command -grep printf a..c is successfully executed (the pattern printf is found in a.c - exit status is 0) then the message "The pattern –printf is found in the fil e -a.c is displayed. Otherwise if the command is failure (the pattern printf is not found in a.c - exit status is nonzero) then the message 'The pattern -printf is not found in the file -a.c is displayed.

Test command

This command is used to test the validity of its arguments. And mostly this command is used in *conditional_commands.* This test command returns an exit status 0 (true) if the test succeeds and 1 (false) if the test fails.

General format is

test expression

String comparison:

General format is,

test operation

where the operation can be,

stringl=string2	Compares two strings; returns true if both are equal, else returns false
stringl!=string2	Compares two strings; returns true if both are not equal, else returns false
-z string	Checks whether the specified string is of zero length or not; returns true if the string is null, else returns false
- n string	- Checks whether the specified string is of non-zero length or not; returns true if the string is not null, else returns false

Numerical Comparison:
General format is,
test numberl operator number2
where the operator can be,

eq	Returns true if number 1is equal to number2; else returns false
-ne	Returns true if number l is not equal to number2; else returns false
-gt	Returns true if number 1is greater than number2; else returns false
-It	Returns true if the number l is less than number2; else returns false
-ge	Returns true if the number l is greater than or equal to number2; else returns false
-Ie	Returns true if number l is less than or equal to number2; else returns false

Example

```
if test $# -ne 2
echo Usage : $0 <filenamel> <filename2>
echo This script requires two arguments
fi
```

Note that $# refers the number command line arguments.

File Checking:
General format is,
test operator <filename>

where the operator can be,

-e	Returns true if the file specified in <filename> exists; else returns false
-f	Returns true if the file specified in <filename> exists and is a regular file (not a directory); else returns false
-d	Returns true if <filename> is a directory, not a file; else returns false
-r	Returns true if the file specified in <filename> is readable; else returns false
-w	Returns true if the file specified in <filename> is writeable; else returns false
-x	Returns true if the file specified in <filename> is executable; else returns false
-s	Returns true if the file specified in <filename> exists and its size is greater than zero; else returns false

Example

```
If test –r a.c
Then
```

Echo the file –a.c exits and it is readable
fi

The expressions (combination of operands and operators) can be combined using the following logical operators.

!	NOT operation; It is a unary operator that negates (true to false, false to true) the result of the specified expression
-a	AND operation; Returns true if both the expressions are true, else returns false.
-o	OR operation; Returns false if both the expressions are false, else returns true.

Examples

(i) if test -r a .c -a -w a.c
then
echo "a.c is both readable and writeable"
fi
(ii) if test ! -r a.c
then
echo "a.c is not readable"
fi

case statement:

This is a multi-branch control statement.
General format is,

case value in
patternl) <commands> ;;
pattern2)<commands> ;;
.
.
.
Pattern) <commands>;;
esac

This statement compares the value to the patterns from the top to bottom, and performs the commands associated with the matching pattern. The commands for each pattern must be terminated by double semicolon.

Example

echo "1. Directory listing ; d. date "
echo "po print working directory ; q. quitW
echo "Enter your choicew"

```
read   choice
case  $choice in
1) ls   ;;
d)  date    ;;
p)  pwd   ;;
q)  exit    "
*)  echo  "Invalid    choicen    "

esac:
```

ITERATIVE STATEMENTS

These statements are used to execute a sequence of commands repeatedly based on some conditions.

while loop:

General format is,

```
while   <conitional_command>
do
<commands>
done
```

The <*conditional_command*> is executed for each cycle of the loop, if it returns a zero exit status (success), then the commands between do and done are executed. This process continues until the <*conditiona_command*> yields a non-zero exit status (failure) ..

Example

```
[bmi@kousar    bmi]  $  cat  e1.dat
1001  ibrahim
1002  yasin
[bmi@kousar    bmi]   $  cat  e2.dat
1003  kadar
1001  ibrahim
[bmi@kousar    bmi]   $  cat  e3.dat
1004  abdulla
1005  kumar
[bmi@kousar    bmi]   $  cat  e4.dat
 1001   ibrahim
 1006  ashok
 [bmi@kousar    bmi]   $  cat  whiledemo.sh
 pattern=$l
 shift
 while   grep   $pattern    $1  >  /dev/null
 do
 echo   "The  pattern    is  found   in  the   file  _$ln
 shift
```

```
done
echo "The pattern is not found in the file _$ln
echo "So, the loop ends without examining remaining arguments"

[bmi@kousar bmi] ,$ chmod u+x whiledemo .sh
[bmi@kousar bmi] $ ./whiledemo .sh ibrahim el .dat e2 .dat e3 .dat e4 .dat
The pattern is found in the file -el.dat
The pattern is found in the file -e2 .dat
The pattern is not found in the file -e3 .dat
So, the loop ends without examining remaining arguments
```

until loop

General format is,

until <conditional_command>
do
<commands>
done

This loop is similar to the while loop except that it continues as long as the *<contional_command>* fails (returns a non-zero exit status).

Example

```
until who | grep ibrahim > /dev/null
do
sleep 5
done
echo "The user -ibrahim is logged on .
```

This Shell script will delay the process until the user -ibrahim logs on Le. the loop will be executed until the user -ibrahim logs on. The **'sleep'** command is used to wait the process at the specified number of seconds mentioned as its argument. (sleep - do nothing)

for loop

General format is,

for <variable_name> in <list_of_values>
do
<commands>
done

The *<variable_name>* has a value from *<lisCof_values>* in each cycle of the loop. And the loop will be executed until the <lisCof_values> becomes empty.

Examples

(i)

```
for i in 1 2 3 4 5
do
echo $i
```

done
The output of this Shell script will be,
1
2
3
4
5
*(i i) for i in $**
do
cat $i
done
If we pass el.dat, e2 .dat, and e3 .dat as arguments to the above Shell script, then the contents of these files are displayed.
*(i i i) for i in **
do
echo FILE NAME: $i
cat $i
done
This Shell script will display the contents of all the files in the current directory.

break
This command is used to exit the enclosing loop (for, while, until) or case command. The optional parameter n is an integer value that represents the number of levels to break when the loop commands are nested. This method is very convenient to exit a deeply nested looping structure when some type of error has occurred.
General format is,
break [n]
where n represents the number of levels to break.

continue
This command is used to skip to the top of the next iteration of a looping statement. Any commands within the enclosing loop that follow this continue statement are skipped and the execution continues at the top of the loop. If the optional parameter "n" is used, then the specified number of enclosing loop levels are skipped.
General format is,
continue [n]
Example
[bmi@kousar bmi] $ cat continuedemo.sh
while true
do
echo "Enter a word"
read word
if test -z "$word"
then

```
echo "Null string"
continue
fi
echo "Given word is $word"
break
done
[bmi@kousar bmi] $ chmod u+x continuedemo.sh
[bmi@kousar bmi] $ ./continuedemo.sh

Enter a word
ibrahim
Given word is ibrahim
[bmi@kousar bmi] $ ./ccntinuedemo.sh
Enter a word
Null string
Enter a word
Null string
Enter a word
mohamed
Given word is mohamed
```

INFINITE LOOPS

The loops that we have discussed are executed based on condition. The following loops are executed infinitely. Only break or exit commands used within the body of the loop will exit the infinitely.

```
1) while true
   do
   statements
   done

2) until false
   do
      statements
   done
```

Shell functions

A Shell function is a group of commands that is referred to by a single name. Shell functions are similar to the Shell scripts except that the Shell functions can be executed directly by the login Shell, but Shell scripts are executed by a sub-shell.

General format is,

```
function_name ( )
 {
    commands
{
```

Example

```
[bmi@kousar bmi] $ disp()
> {
```

```
> echo "THE .TXT FILES IN THE CURRENT DIRECTORY ARE,"
> Is -1 *.txt
> echo " ...over."
>}
[bmi@kousar bmi] $ disp
THE .TXT FILES IN THE CURRENT DIRECTORY ARE,
-rw-rw-r-- 1 bmi bmi 31 Jan 2 09:39 capital.txt
-rw-rw-r-- 1 bmi bmi 0 Dee 31 13:00 e1.txt
-rw-rw-r-- 1 bmi ibrahmi 43 Mar 18 2002 file1.txt
-r---------- 1 bmi bmi 34 Dee 16 2002 file2.txt
-rw-rw-rw- 1 bmi bmi 67 Dee 16 2002 letter .txt
-rw-rw-r-- 1 bmi bmi 31 Jan 2 09 :39 lower .txt
-rw-rw-r-- 1 bmi bmi 14920 Dee 16 2002 lpr .txt
-rw-rw-r-- I bmi bmi 39 Dee 27 2002 result .txt
-rw-rw-r-- 1 bmi bmi 31 Jan 2 09 :38 small .txt
-rw-rw-r-- 1 bmi bmi 31 Dee 16 2002 test1 .txt
-rw-rw-r-- 1 bmi bmi 31 Dee 14 2002 test2 .txt
-rw-rw-r-- 1 bmi bmi 93 Dee 16 2002 userlist .txt
...over .
```

On typing the name of the function -disp at the $ prompt, the output of the commands specified in the Shell script are executed.

sleep

This command generally used in Shell scripts to delay the execution.

General format is,

sleep <NumberOfSeconds>

This command delays for the specified number of seconds.

Example

```
[bmi@kousar bmi] $ eeho "Hello
Hello
ibrahim
```

The string "ibrahim will be displayed after one minute of the display "Hello .

```
[bmi@kousar bmi] $ eat delayeg
eeho "Hello"
sleep 60
eeho "ibrahim"
[bmi@kousar bmi] $ sh delayeg
Hello
Ibrahim
```

SYSTEM ADMINISTRATION

The system administration, maintained by System Administrator and he is also called as Super user, includes the management of the entire system such as installing and removing terminals and workstations, installing printers and backup activities, adding and removing new users and user group, providing security, managing disks and file systems and monitoring the network.

The system administrator is the super user who maintains the entire system. The system administrator has a special user login name, named root and special prompt t. The System Administrator can create files in or delete any file from any directory. He can also read from, write to or execute any file. The super user mode can be acquired by,

- Running in single user mode
- Logging as r t with giving correct super-user password
- By giving **su** (substitute user) command from a user's Shell

When the su command was used to become the System Administrator, the user can return to the normal mode by terminating the Shell by pressing [Ctrl+d] or by giving the exit command.

Most of the commands that are used as a System Administrator are stored in the /shin and /usr shin directories. The commands can be executed by giving their full path names or by including these **/ sbin and /usr / sbin** directories in the PATH when logged as System Administrator.

BOOTING THE SYSTEM

When the system is switched on at the first time, the system executes the boot loader from the start-up disk. This boot loader loads the operating system kernel and then runs the kernel. A general-purpose utility named **lilo** (linux loader) writes this boot loader to the start of the active partition. When the system is switched o~ the next time, the kernel will be started by this **lilo** utility.

The kernel itself being the first process gets the PID 0. The kernel executes the **/ ete / 'init'** file to invoke the "init" process. This process has the PID 1 and this is responsible for the initiation of all other processes.

This init process invokes the **/etc/inittab** file. Thi sinittab file informs in it about the status of terminals such as which terminals can be used. And also decides what programs are to be run to perform various startup and maintenance functions.

Then, the init invokes the **/etc/getty** file. This file has the various parameters for communication between the terminal and the computer system. By using these parameters, the login prompt is displayed.

The Linux system can be set up in a number of modes, called as run levels that are controlled by **init**. These run levels and their actions are given below.

Run level	Action
S	Single user mode
5	Multi-user mode
o	Halts the system
4	Starts X windows immediately after booting

For **Example**, the command init 0 halts the system and the command init 5 brings the system in multi-user mode.

Single-user mode: When the system is operating in the single user mode, only the system console will be enabled.

Multi-user mode: It is the default mode and all individual file systems are mounted and also system domains are started.

The following syntax of **telinit** command can be used to invoke the multi-user mode when the system is working in the single user mode.

telinit 5

The **shutdown** utility performs all the necessary tasks to bring down the system safely.

General format of this shutdown utility is

shutdown [-options] time

where the options can be,

h Halts the system after shutdown

r Reboots the system after shutdown

k Don't really shutdown, but only warn

The time specifies the absolute or relative time that when the shutdown command starts its function.

Example

 # shutdown -h 16:30

This command halts the system at 4:30 PM. (Absolute time)

 # shutdown -r +40

This command reboots the system after 40 minutes of issuing this command. (Relative

The halt command can be used to halt the system. The reboot command reboots the system. Note that halt will do the function of -h option of shutdown command and the reboot does the function of - r option of the shutdown command.

Always do proper shutdown. Because, Linux stores data in memory buffers that are periodically written into the disk to speed up disk access. If there is no proper shutdown, the contents of the disk buffers are lost. The shutdown command forces the buffers to be written into the disk.

This sync (synchronize) command can be used to flush the file-system buffers i.e., it writes the information stored in the buffers (main memory) into the appropriate files on the disk. Generally, to improve the system performance, the kernel does not write the information to the disk immediately; but it keeps the data in main memory and writes them into the disk when it desires. This sync command forces the kernel to write any data buffered in memory out to disk. And this command can be used to save the data from system crashes.

ADDING A USER

The System Administrator can add / remove users to the system manually, or he can also use tools such as useradd, to do the same job which avoids risks.
A typical format of the 'useradd' command is,

useradd <username>
useradd -u <user-id> -g <group-id>
-c <comment> -d <user's_home_directory>
-s <default_lonin_Shell> <username>

The System Administrator can set an initial password for a user by using the following format of this passwd command.
passwd <username>

If the SA uses add user command for creating a user, then he must set an initial password for the user using the above format of the passwd command, or he must delete the password entry of the user in /etc/passwd file. However, the logged user can change this password using the passwd command without any arguments.

This useradd command inserts the necessary entries in the files -
/ etc / passwd, fete / shadow (if shadow password system is used) and fete/group .
All the new user's informations are stored in the / etc /passwd file as a single line in the following format:

Username:Password:User_ID:Group_ID:Comment:Home_directory:User's_Shell

Username :Maximum eight characters-length name of the user

Password : The system places 'x' in this field if the shadow password system is used, otherwise the user's encrypted password is stored in this field.
User ID : A number that identifies the user
Group ID :A number that identifies the group that the user belongs
Comment : Comments about user's original name, designation, address etc.
Home Directory : It is the directory chat the system will place the corresponding user after the successful login
User's Shell : This specifies the Shell type for the user. If /bin/tcsh is specified, then the System will provide C Shell for the user after the successful login. If no Shell type is specified, then /bin/bash (Bash Shell) is taken by default.

A typical entry in the /etc/passwd file is shown below.
bmi:x :l00:25:IBRAHIM:/home/kgr:/bin/bash

For every line in / etc /passwd file, there is a corresponding entry in */ etc /* shadow (if the System uses shadow password) and in */ etc / group.*
The entry in the / etc / shadow includes user_name, encrypted password, the number of days that the user-account is valid.

User's_name: Encrypted-password: **. . .**

Then, the user's name is inserted at the corresponding group that is stored in the / etc / group file. A typical line in the / etc / group file has the following format:

Group_name: Password: Group_ID: Users_belonging_to_this_group

Example

[root@kousar /root] adduser kgr
[root@kousar /root] passwd kgr
Changing password for user kgr
New UNIX password :
Retype new UNIX password:
password : all authentication tokens updated successfully
(Note that the password characters are not echoed.)
Then, a user account for the user - kgr is created and his home directory is / home / kgr, login Shell is bash. All these informations are default excluding username and password.
By changing the username in the / etc / passwd, the System Administrator can make a user account inaccessible temporarily. The System Administrator must change the changed username into the original name in the /etc/ passwd file in order to bring that user account accessible.

DELETING A USER
The userdel command is used to delete a user from the System, which removes all the entries pertaining to the specified user from the three files - / etc /passwd, / etc / shadow, and / etc / group.
General format is,
userdel <username>
This command only removes the user account, but the user's files remains as they are, not deleted. The System Administrator can delete these files separately if required.

The System Administrator can also create the necessary groups and add the users to these groups, instead of leaving to the add user command.

The group add command is used to add a group to the System, which has the following syntax.

groupadd [-g <group_id>] <group_name>

The *<.group_id>* is a non-negative numeric value, which refers the group by number.
The default for *<group_id>* is to use the smallest numeric value greater than 500 and greater than every other group. The values between 0 and 499 are typically reserved for System accounts.

The *groupdel command* is used to delete an existing group. But all the users who belong to that group must be removed before using of this command.
General format of this command is,
groupdel <group_name>

Example
 # mkdir /home/kumar
 # group add -g 550 mca
 # useradd -u 1050 -g 550 -e KUMAR -d /home/kumar -s /bin/bash kumar
#passwd kumar
(Set initial password for kumar)
Then, the user named kumar is created, who belongs to the group mca.

Remote system accessing
Linux provides several commands to access the remote systems, which are connected with the local system (your system) through network. Linux works on TCP/IP, and then the users can acquire all the features of the architecture.

telnet

This command is used to connect a remote system from the working system.
telnet <IP_address>

The above format connects the remote system whose IP address is specified in <IP_address>.The above format prompts for the login-name of a user and his password, who has a login account on the remote system. By specifying the correct login-name and the password, an user can logon to the remote system successfully; sitting on the local system.
Then, the user can access the allowed resources on the remote system from the local system. Any command given on the local system during this telnet session are sent to the remote system for execution and the local system will become a dump terminal until the user terminates this telnet session (remote login) by using the \ exi t I command. Thus, the remote loin provides the facility to manipulate the resources on a remote machine from a local machine.
The user can also make a telnet connection to a Linux system from a Microsoft Windows system (they must be connected) by using the above telnet command in the MS-DOS command prompt.
Note that the TCP/IP is independent of the network hardware**.**

Example

Connecting a remote Linux system from the local Linux system:

The user -bmi is logged with the local Linux system. And the user -john has a login account on the remote Linux system, who is going to do the remote login.

```
[bmi@kousar bmi]$ ls

address bmi1.txt bmi2.txt bmi3.txt bmi4.txt bmi5.txt Desktop emp.doc
[bmi@kousar bmi]$ telnet 172.16.2.82
Trying 172.16.2.82 ...
Connected to 172.16.2.82/
Escape character is 'A'l.
Red Hat Linux release 7.1 (Seawolf)
Kernel 2.4.2-2 on an i686
login: john
Password:
Last login: Mon May 19 09:48:00 from ibrahim
[John@ibrahim john ] $ Is
Desktop john1.txt john2.txt john3.txt test.doc
[John@ibrahim john ] $ cat> newfile
This is a new file created on the remote system.
[John@ibrahim john ] $ Is
Desktop john1.txt john2.txt john3.txt newfile test.doc
[John@ibrahim john ] $ exit
logout
Connection closed by foreign host.
[bmi@kousar bmi]$
[bmi@kousar bmi]$ Is

address bmi1.txt bmi2.txt bmi3.txt bmi4.txt bmi5.txt Desktop emp.doc
```

Note that all the commands are sent to the remote system for execution after john did the remote logon successfully.

Connecting a remote Linux system from the local Windows NT system:

Invoke a Dos window and give the telnet command.

Assignment question:

1.what is shell script?

2.Explain the two types of shell variables?

3.explain the users of echo statement.

4. write a note on conditional parameter substitution in bash shell?

5.what are positional parameters? Explain their roles in shell scripts?

6. explain the uses of shell meta-characters.

7.how the conditional are tested using test command on file and strings?

8.explain the conditional statements in bash shell with suitable example?

9.Illustarte iterative statements available in bash shell?

10.what are the different between while and until loops? Explain?

11.how do you define and call shell functions? How does it differs from shell scripts?

VALUE ADDED CHAPTER

PROTECTION

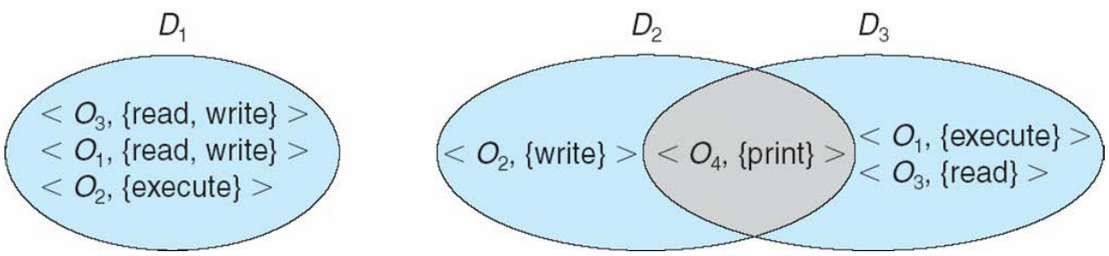
12.1 Goals of Protection

Discuss the goals and principles of protection in a modern computer system. Explain how protection domains combined with an access matrix are used to specify the resources a process may access Examine capability and language-based protection systems.

- Operating system consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so .

12.2 Domain Structure

- Access-right = <*object-name*, *rights-set*>
  where *rights-set* is a subset of all valid operations that can be performed on the object.
- Domain = set of access-rights .



Domain Implementation

1.System consists of 2 domains:

- User
- Supervisor

2.UNIX

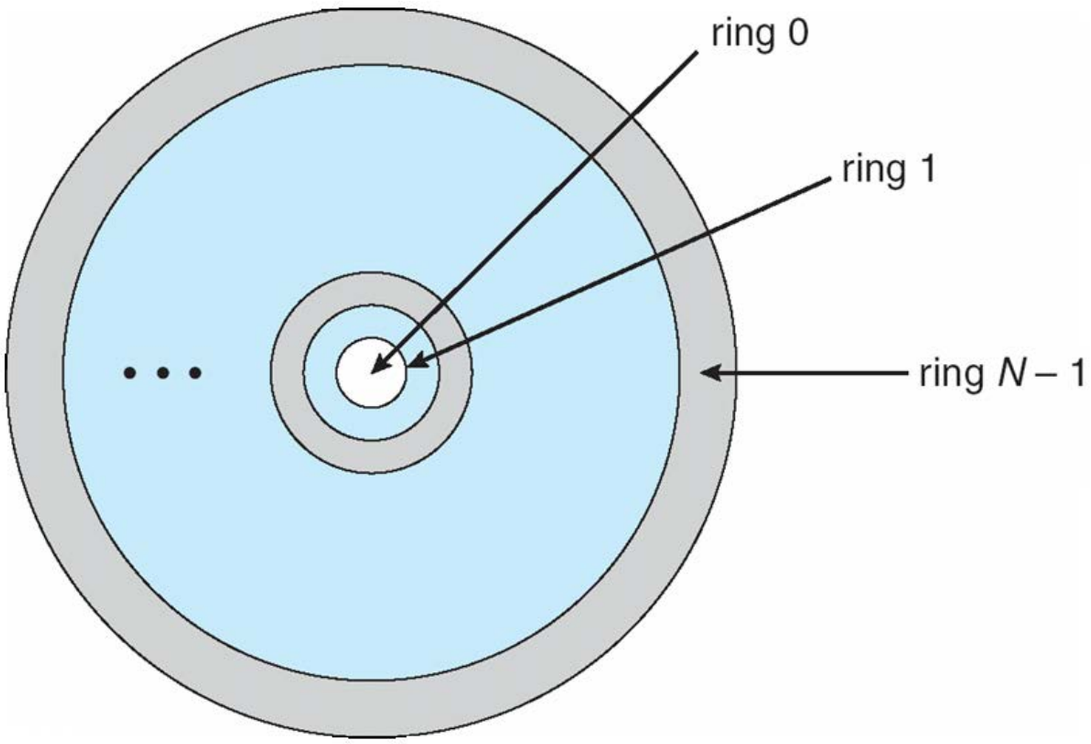
- Domain = user-id

3.Domain switch accomplished via file system

- Each file has associated with it a domain bit (setuid bit)
- When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset

Domain Implementation (MULTICS)

- **Let D_i and D_j be any two domain rings**
- **If $j < I \Rightarrow D_i \subseteq D_j$**



12.3 ACCESS MATRIX

- View protection as a matrix (*access matrix*)
  - Rows represent domains
  - Columns represent objects
  - *Access(i, j)* is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

Capability-Based Systems

1. Hydra
   a. Fixed set of access rights known to and interpreted by the system

b. Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights

2. Cambridge CAP System

   a. Data capability - provides standard read, write, execute of individual storage segments associated with object

   b. Software capability -interpretation left to the subsystem, through its protected procedures

12.4 LANGUAGE-BASED PROTECTION

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

Language-based protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

Protection in Java 2

1. Protection is handled by the Java Virtual Machine (JVM)
2. A class is assigned a protection domain when it is loaded by the JVM
3. The protection domain indicates what operations the class can (and cannot) perform
4. If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library

Credit Based Third Semester B.C.A. Degree Examination, Oct./Nov. 2014

(New Syllabus) (2013-14 Batch Onwards)

OPERATING SYSTEM

Note: Answer any ten questions from Part A and answer one full question from each Unit of Part B.

PART A

1. a) Define the different types of real time systems.

b) Define PCB. Mention the components of PCB.

c) Define the terms throughput and response time.

d) What is a critical section? Name the requirements for solution to the critical section problem.

e) Name the methods to recover from deadlock.

f) Define the terms 'Max' and 'Need' in Banker's algorithm.

g) Differentiate logical address space and physical address space.

h) List out any four file types with extensions.

i) Give the difference between absolute path name and relative path name.

j) What is the purpose of cp command in Linux ? Give example.

k) What is the purpose of shift command in Linux ?

l) Name any two directory oriented commands in Linux.

PART-B

UNIT-I

2. a) Explain any five services of operating system.

b) Define and explain the benefits of threads.

c) With a neat diagram explain process scheduling using queuing diagram. (5+4+6)

3. a) Write a note on process management and file management.

b) Explain process state transition with a neat diagram.

c) Discuss FCFS and SJF CPU scheduling policies with example and also compare the same. (5+5+5)

UNIT-II

4. a) What is readers-writers problem? Explain.

b) Explain the necessary conditions for deadlocks to occur in a system.

c) Consider the following snapshot of a system: (5+4+6)

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Answer the following using Banker's algorithm.

1) Is the system in a safe state ?

2) If a request from process P1 arrives for (0, 3, 2, 1) can the request be granted immediately?

5. a) Explain the concept of semaphores in process synchronization.

b) Explain the deadlock avoidance methods.

c) How can you detect deadlock using 'wait-for' graph? Explain.

UNIT III

6. a) Explain swapping method with a diagram.

b) Explain LRU page replacement algorithm with an example.

c) Explain the concept of equal and proportional allocation of frames in memory management. (5+5+5)

7. a) Explain demand paging with a diagram.

b) Given the following page reference string:

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

With 3 frames of memory write the steps for FIFO page replacement algorithm which shows the occurrence of page· faults and find the number of page faults.

c) Explain any five file operations. (5+5+5)

UNIT IV

8. a) Explain the different file permissions in Linux. What does -rw-r-r- -file1 mean for the file file1?

b) Explain the case statement with syntax and example.

c) What are the different options of 'cat' command? Give examples. (5+6+4)

9. a) Explain the following commands:

i) grep

ii) chmod

b) Write a note on positional parameters.

c) Write a note on different modes of vi editor. (4+5+6)