CONTENTS

SYLLABUS

OPERATING SYSTEMS WITH UNIX AND SHELL PROGRAMMING

SECTION A

Operating System Fundamentals

Introduction Concepts: Operating system function and characteristics, historical evolution of operating systems, Real time systems, Distributed systems, Methodologies for implementation of O/S service, system calls, system programs, Interrupt mechanisms.

SECTION B

I/O System, File Management and Process Scheduling

File System: Function of the system, File access and allocation methods, Directory structure, file protection mechanisms, implementation issue, hierarchy of file, disk scheduling policies.

Process Scheduling: Process, PCB, state transition, Level of Scheduling Comparative study of scheduling algorithms

SECTION C

Feature of UNIX, directory structure of UNIX, File structure of UNIX, concept of inodes. Logging into Unix, format of UNIX components, basis operations on files, filters and pipelines mail and communication commands.

SECTION D

Shell Script

Types of shells, control structure for shells and I/O for shells. Use of Editors, VI, EX & Ed.

SECTION A

UNIT 1 INTRODUCTION

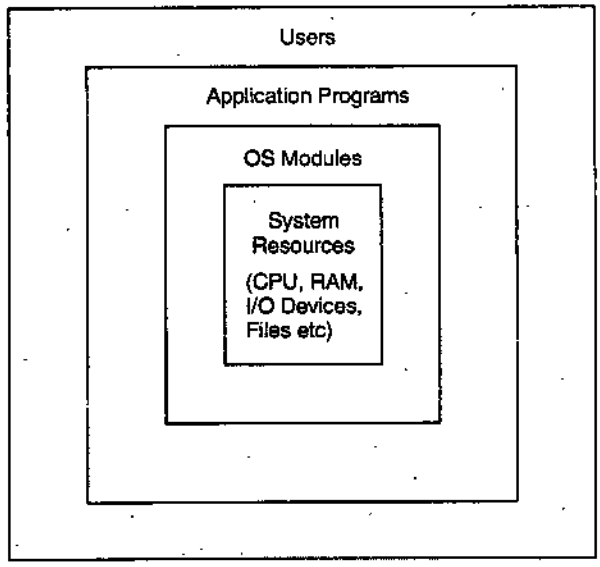
NOTES

★ LEARNING OBJECTIVES ★

- Introduction
- Major Goals of an Operating System
- Evolution of Operating Systems
- Types of Operating Systems
- Summary
- Self-Assessment Questions

INTRODUCTION

1. An operating system (OS) is an organized collection of software modules that provide an effective interface between the computer users and the computer resources. The computer resources comprise CPU, Main Memory, I/O Devices (like Keyboard, Mouse, Monitor, Hard Disk Drive, Floppy Drive, CDROM etc), Secondary Storage, Files & Network etc.



2. The Users execute Application Programs, which make System Calls to the OS modules for allocation of resources. In response, the OS modules provide the necessary resources in a systematic manner. The Users can

also interactively request resources through system commands made directly to the OS Modules. So, the Users & User Programs have to go through the Operating System modules for the allocation of needed resources. Thus, OS acts as a Resource Manager. It decides:-

- Which process should get what resources?
- In what quantity?
- At what time?
- And for how long?

MAJOR GOALS OF AN OPERATING SYSTEM

Any Operating System should meet the following major goals:-

(*a*) Optimize the Use of Computer Resources so as to maximize its throughput.

(*b*) Create a User-Friendly Computing Environment for accessing the Computer Resources.

EVOLUTION OF OPERATING SYSTEMS

Starting from the 'Bare Machine' approach (*i.e.* machine with no OS) to its present sophisticated forms, the Operating System has evolved through various stages of its development like **Serial Processing, Batch Processing, Multi-programming and Interactive Timesharing Approach etc. as traced below.**

Bare Machine Approach

In the earlier stages of computer development, there was nothing like a Operating System. A computer used to be programmed directly in machine language, without any system software support. This is now known as a 'Bare Machine' approach. Such an approach was possible on earlier machines like PDP-11, wherein a program, coded in Machine Language, could be entered into the RAM through front-panel switches and executed. The results of execution could be displayed on a set of LEDs, mounted on the front-cover itself. This approach was extremely user-unfriendly. It was very tedious to code a program in machine language (using zeros and ones) and then enter it through the toggle switches. In case of a power failure, such a program used to vanish and had to be entered all over again. The whole process was highly time-consuming, resulting in a poor throughput. This brought out a need for the following:-

NOTES

- I/O devices to enter code & data conveniently and output the results in a user-friendly manner.
- Secondary Storage to save the entered programs on non-volatile media.
- Assemblers & Compilers to translate programs coded conveniently in an Assembly Language or a High Level Language.
- Linkers to link library modules with the user developed object modules.
- Loaders to load the executable files from Secondary Storage to Random Access Memory (RAM) and execute.
- I/O Devices like printers to obtain hard copies of the outputs.

Serial Processing

With advent of I/O Devices such as punch cards, paper tapes, mag-tapes and line-printers etc, and the advent assemblers & compilers, there came the concept of serial processing. In serial processing, the programs were executed strictly in a serial manner (one after another), wherein program source code, written in a program coded in assembly language or high-level language on a pack of punched cards could be entered into the computer using punch card reader. Then, necessary assembler or compiler used to be loaded to translate the source code into an object file. Then the object file used to be linked with library routines to create an executable file, which was then executed and results were outputted. Only then next program could be taken up for execution in a strictly serial manner.

Limitations of Serial Processing

In serial processing, the OS was limited only to a Loader and some I/O Device Drivers. The system utility programs (not forming part of OS), available at this stage, comprised only assemblers, compilers, linkers, editors and debuggers. These utility programs relied on the services provided by the OS.

Although, a big improvement over the 'bare machine' approach, the serial processing mode was very inefficient. The sequencing of the operations was being done manually, which resulted in poor utilization of system resources. At a time, only a small subset of resources could be used.

The jobs used to be executed in the same sequence, as fed; and their output used be printed out on the line printer. Imagine that five jobs coded in Pascal, FORTRAN, Pascal, FORTRAN, Pascal appear in that sequence. So, the Operating System has to first Pascal Compiler for the first program, then Fortran Compiler for the second program and then re-load Pascal Compiler for the third program and so on. This brought out the concept of *Batching Jobs* having similar needs *i.e.* it would be better that first load Pascal Compiler and execute first, third & fifth program and then load the

FORTRAN compiler and execute second and fourth program. This would need loading Pascal & FORTRAN compilers only once and thus reduce the total execution time of the five programs considerably. This batching of programs was initially done manually by the Operators. It brought out the concept of Batch Processing.

Batch Processing

With the invention of hard-disk drives, the things were much better. Now, the jobs loaded through card-reader, could be stored on the disk; thus creating a pool of jobs on the Disk. The OS Commands written in a Control Language used to be embedded in the job stream, together with the user programs and data. A Memory-resident portion of the Batch OS, called 'Batch Monitor' used to read, interpret and execute these commands automatically and group (batch) the pooled jobs into separate batches placing similar jobs having identical needs in the same batch, based on pre-specified criteria. In response to the commands, the batched jobs were executed automatically, one after another, without any manual intervention. For batched jobs, the common housekeeping operations, like loading of compiler, were performed only once. Thus, overhead per program got reduced, thus improving the system utilization. With this, the concept of *Job Scheduling* got evolved.

Limitations of Batch Processing

Though a big improvement over serial processing, batch processing had following limitations:-

(*a*) With the advent of intelligent I/O subsystems, it was possible to perform I/O operations concurrently with the CPU processing, but this potential was not utilized in batch processing When a job was being read-in or its output was being printed out, the CPU used to be idling. Each job had to wait for long before execution.

(*b*) The user had no means of interacting with a job, when it was under execution. Suppose, there was some syntax error, the program would fail to compile; but the user will know this, only when he collects the printout next morning. Then he would fix up the bug and resubmit the job. This time, the program may have some run-time error. So, the program development used to be very long.

(*c*) Debugging was feasible only offline and it used be very time-consuming.

Thus, the main limitation of a batch processing system was that at any time, it used to dedicate all the system resources to a single program, being executed. However, an executing program may oscillate between computation phase and I/O phase. So, at a time, either CPU will be busy or I/O will be busy. Thus, at a time either CPU will be idling or I/O will be idling. Thus, the

degree of resource utilization used to very low; thus limiting the system throughput. Thus, as far as program development was concerned, batch processing did not provide a major improvement over serial processing. It had many limitations; like a fairly long turn-round time (i.e. the time elapsed from the time a job is submitted to the time when output is received by the user). Online Debugging was not feasible. The offline debugging was very laborious and time-consuming.

Multi-Programming

Multi-Programming refers to the concept, wherein more than one programs used to be activated concurrently; one of the active programs being in 'run-mode' utilizing the CPU and others utilizing the I/O devices at the same time or being in a 'wait state' waiting for resources to be available. This improved the utilization of system resources, thus increasing the system throughput. Such a system was called multi-programming system. Significant performance gains were achieved, by interleaving the execution of more than one program. With single CPU systems (Von Neumann Architecture), only one program could be in 'run state' (i.e. in control of the CPU) at a time, and other concurrent programs could be in other states; like performing I/O operation, or waiting for some event to occur, or ready to take control of the CPU. Since a number of programs would be active concurrently, competing for the shared resources, the Operating System became quite complex.

Interactive Time Sharing Systems

Such systems, in addition to multi-programming, support some additional features, like permitting user interaction with the system, through OS commands. The user Programs can be entered and executed in an interactive mode. Source code of a program is entered interactively, through a terminal using screen editor. A file produced in this manner is processed interactively, by a language-translator, to produce an object file. Related object files and library routines can be linked together using a linker, to produce a single executable file, which is executed interactively. Any bugs, thrown up during execution, can be cleared by re-editing the source-code file interactively, re-compiling it and executing again. This reduces drastically the program-development period.

Each interactive user is assigned a time-slice in a round-robin fashion, during which it controls the CPU. During the time-slice, the process gets control of CPU and tries to complete its computation. If the computation is not completed during the assigned time-slice, then the running process is preempted; then it has to wait for time-slice in the next cycle to resume computation from where it was preempted. The time-slice cycle is so adjusted that each user has a feel as if the CPU (with a reduced processing power) is assigned to it all the time.

STUDENT ACTIVITY

1. What do you mean by operating system?

2. Explain major goals of an operating system.

In contrast to the Batch Systems, which impose turn-around delays and support only offline debugging, the Multiprogramming Systems provide quick terminal response and allow online debugging. Each user virtually gets a feel as if he/she has the entire machine to himself/herself. So, it provides a more convenient and more productive environment for software development and execution of programs. With this approach, there was a drastic reduction in the program development time. It remained in vogue for a long time during the 70s and 80s, till the Desk-Tops became widely available.

TYPES OF OPERATING SYSTEMS

OS can be classified into various categories, with respect to the type of processing, it supports. Following are the main types of OS:-

Batch OS

This is the most primitive type of Operating System. Batch Processing required that a program, its related data and relevant control commands should be submitted together, in the form of a Job (normally coded on a bunch of punched cards). Batch OS allows no interaction between the users and the executing programs. Thus, the programs that have long execution times (like payrolls, forecasting, statistical analysis and large scientific number-crunching programs) and require little operator interaction during execution, were well served by Batch Processing. However, due to long turn-around delays and infeasibility of online debugging, the Batch Processing was not at all suitable for software development. The typical features of a batch OS were:-

(*a*) *Scheduling:*. The scheduling of jobs was normally in the order of their arrival i.e. First Come First Served (FCFS). This criteria provides a fair deal to the jobs, but suffers from long average turn-around time and average waiting time, since some short jobs may have to wait for unduly long periods to get executed, if some long jobs are already queued ahead of the short jobs. To overcome this limitation, another criteria known as "Shortest Job Next", was evolved, which offered preference to pending shorter jobs over the long jobs. This offered much better his average turn around time and average waiting time of the pending jobs.

(*b*) *Memory Management*: Memory used to be divided into two permanent partitions- one permanently occupied by the resident portion of the OS and the other dynamically used to load the transient programs for execution. When a transient program terminates, it vacates the memory partition occupied by it and the same is assigned to other waiting transient program.

(c) *I/O Management*: Since, only one program used to be under execution at a time, there was no contention for the allocation of I/O Devices. So, a simple program-controlled I/O approach was good enough to access the I/O Devices.

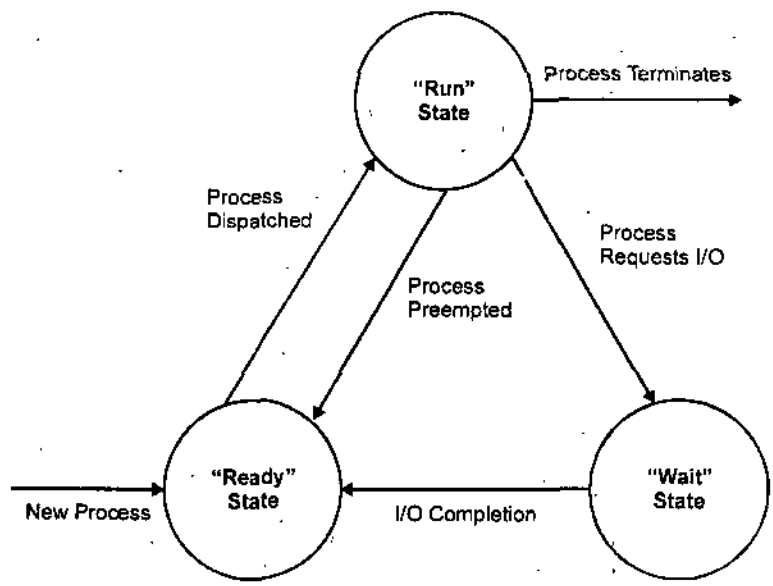
(d) *File Management:* Since, there used to be only one program accessing the files at a time, there was no need of providing concurrency control.

Thus, in a nutshell the Batch OS was limited to a Loader and some simple Device Drivers.

Multiprogramming OS

A multiprogramming system permits multiple programs to be loaded into the memory and execute the programs concurrently. A program in execution is called a "Process" or a "Task" The concurrent execution of programs has a significant potential for improving the utilization of system resources and thus enhancing the system throughput, as compared to batch and serial processing. This potential is exploited by multiprogramming OS, which multiplexes the computer resources amongst a multitude of concurrent programs. A program in execution is called a Process. When a running Process requests some I/O, it goes to 'WAIT' state. Then another Ready process (i.e. one of the processes from the 'READY' queue) is scheduled to "RUN". So, when a Process goes for I/O, the CPU is not left idling. When the I/O requested by a WAITING process is completed, it is transferred from 'WAITING' to 'READY' state and wait to be scheduled, as per the pre-decided scheduling criteria. Different states of a Process in a Multi-programming environment are depicted below:-

Process States in Multi-Programming



NOTES

Multi-tasking Vs Multi-programming

The concept of managing a multitude of simultaneously active programs, competing with each other for accessing the system resources, is called multi-tasking. A multi-tasking OS is characterized by its capability to support concurrent execution of more than one task. This is achieved by simultaneous maintenance of code and data of several processes in the memory at the same time, and by multiplexing the CPU and I/O resources amongst the active tasks. The multi-tasking OS monitors the state of all the tasks and of the system resources.

On the other hand, Multi-programming OS will have all multi-tasking features and in addition would also support sophisticated memory management.

Multi-User OS

This refers to a Multi-programming OS, that supports simultaneous interaction with multiple users. A Multi-user OS should also support:-

- User authentication by User-id and Password
- Resource usage accounting for the interactive users (i.e. the amount of usage of resources by various authorized users)
- Protection of users' environments.

Multi-tasking, without multi-user support, is usually found in real-time systems.

Multi-access OS.

This refers to an OS, which permits simultaneous access to a computer system, through multiple terminals. It may not be a multi-programming OS; for example an Airline Reservation System, which permits access through a large number of terminals, but through a single controlling program.

Multi-processor OS

Multi-Processor refers to an environment of multiple CPUs, tightly coupled through a common bus. Such OS will have multi-tasking features and support simultaneous execution of multiple tasks on multiple CPUs. The multiple processors share system bus and clock. In addition, the CPUs may also share RAM and other peripheral devices. Usually the parallel processors are executing the same code. While writing a program for such an environment, the programmer has to indicate explicitly what segments of the code can be executed in parallel.

Interactive Time-Sharing OS

The Time-Sharing Interactive systems belong to the era of large mainframe systems, that were made available to a large number of users for accessing concurrently in an interactive mode. Each user used to interact with the

NOTES

system through a separate Video Terminal. A programmer could enter its source code using sophisticated editors, compile the source code, link the object code and run the executable code interactively. At any stage, the user could interact with the execution of its program. Debugging could be performed online. This reduced the time needed for program development, drastically. The OS served the interactive users, in a round-robin fashion, allowing a fixed time-slice for each user to have control of the CPU. Since, the operators were interacting through slow I/O devices, each user had a feel as if the CPU was available solely to itself. So, the computation capacity of the CPU is multiplexed evenly, amongst the interactive users.

So, Time-Sharing systems were large multi-programmed multi-user systems, designed for program development. One of the major requirements of such a system is good terminal response. It should give an illusion to each user as if the entire computer resources are available to itself, solely. It should provide an equitable share of common resources to each user. For example, when the system is overloaded, the program with higher processing requirements is made to wait longer. Most time-sharing systems use time-sliced, non-pre-emptive scheduling. It has FCFS Queue; and a program requiring service is put at the end of the queue. When the process reaches the front of the queue, it is dispatched. If it is not completed within the time slice, it is pre-empted at the end of the time-slice and put at the end of the queue. Now, it has to wait for the next time-slice, to resume its execution from the point where it left at the end of previous time slice. A process continues to run in bursts like this, till it terminates. The main features of an Interactive Time-Sharing OS are:-

(*a*) *Memory Management.* The memory management in time-sharing systems provides for isolation and protection of co-resident programs. Since, the programs are being executed on behalf of different users, they do not involve much of inter-process communication.

(*b*) *I/O Devices Management.* I/O Management must be sophisticated enough to cope with the requirements of the multiple users, contending for multiple devices. However, due to relatively slow speed of terminals and users, processing of terminal- interrupts would not be so critical. As in most multi user environments, allocation and de-allocation of devices must be handled in a manner so as to provide the system integrity, while attempting to optimize the system performance.

(*c*) *File Management.* Since multiple users will be sharing access to multiple files, the file management must provide protection and access control.

Real-Time OS

It refers to the environment of embedded systems, with a very rigid requirement to complete the processing of input data, in a pre-specified time.

NOTES

The input data, in such systems, is invariably received from real-world sensors. The outputs may be used to control some real-world processes. For example, an Air Defense system, receiving information about aircraft movements from radars and control weapons for the elimination of hostile aircraft. For such an application, it can be well appreciated that the processing of data inputs, within specified time limits, is absolutely critical.

So, the Real Time systems are used in environments, where a large number of events, mostly external to the computer system, must be accepted and processed in a highly time-bound manner (with specified deadlines). Such applications include process control in industry (like steel, petro-chemical plants, thermal plants & nuclear plants etc), military systems for processing of sensor information and weapon control, telephone switching equipment, flight control, real-time simulations etc.

(*a*) *Scheduling.* A primary objective of a real-time system is to provide quick response to external events, so as to meet the specified deadlines of processing. Such systems have to be multi-taking (may not be multi-user). The scheduling is normally priority-based pre-emptive, in which the processes handling the external-world-inputs are assigned higher priority and whenever a higher-priority process becomes ready to run, it pre-empts a lower priority running process. User convenience and resource optimization are of secondary concern. Such systems may be expected to handle thousands of interrupts per second, with a stringent requirement of not missing even a single event.

(*b*) *Memory Management.* The memory management is relatively less demanding, since the process population is more or less static and memory is so designed that most of the processes will always be memory resident, in order to provide a quick response. In fact, the programs are normally ROM-resident. But, the processes have closer interaction. This requires features that enable sharing of memory, amongst processes and at the same time provide protection from each other.

(*c*) *I/O Devices Management.* Time critical device management is one of the major requirements of real time systems. The external events and data have to be accepted and processed, in a highly time-bound manner. This is implemented using interrupt mechanism. The system calls allow the users to connect the devices and access Interrupt Service Routines (ISRs) directly, so as to achieve a good response time.

(*d*) *File Management.* The file management is not very critical, since most of the real-time databases will be memory-resident. Some embedded systems may have program and static data on ROM and may not need any online secondary devices at run-time. The dynamic databases are maintained in ROM. If at all any files are to be accessed, as in the case of large real-time systems, the primary objective is fast access.

The Real-Time Systems can be categorized into:

(a) Hard Real Time OS

In Hard Real Time Systems, all critical tasks have to be completed strictly within the specified time limits. Not meeting the deadlines can be catastrophic; like an embedded system controlling the operation of a refinery. In such systems, all code and data must be in RAM/ ROM. All kernel delays must be bounded. So, the OS should not have any unnecessary frills. Interrupt mechanism must be optimized, so as to respond to the real-world inputs, with minimal delays.

(b) Soft Real Time OS

This is less restrictive type of real time system. If the input data is not processed within specified time intervals, the results may not be catastrophic, but the output may lose its utility. For example, in a system monitoring flight path of an aircraft, the delayed information will have no utility. In such systems, tasks are divided into two categories- Real Time Tasks & Non-Real-Time Tasks. The Real-Time tasks are accorded priority over the other tasks. Any output from such systems would be useful, only if it is produced within a specified time after receiving the corresponding inputs. So, the kernel delays must be bounded. The soft real time systems are not suitable for industrial and defense use, since the delays are not uncertain.

Combination OS

Different types of OS are optimized or geared up to serve the needs of specific environments. However, sometimes an environment may be a hybrid of such environments, for example the same computer may be used as a software development platform and as a target machine in a real-time environment. For this reason, the commercial OS like UNIX, VMS etc are designed to provide a large set of services, catering to the needs of various environments. If batch and interactive jobs are being run simultaneously, lower priority is assigned to batch jobs. In fact, the Batch Jobs may be used as filler, executed only when system resources are idling. The processes, handling real-time interrupts to receive real-world events and inputs, will be assigned higher priority. Such systems would normally employ time-sliced, priority-based pre-emptive scheduling. A combination OS will be the one which caters for the needs a wide spectrum of target environments. For obvious reasons, such OS will not be optimized for a specific environment.

Distributed OS

A Distributed OS caters for a distributed environment, wherein a collection of autonomous computer systems, capable of communicating and cooperating with each other through network, are connected to each other through a LAN/WAN. A Distributed OS governs the operation of such a distributed system and provides a virtual machine abstraction to its users. Key objective

of Distributed OS is transparency i.e. the resource distribution must be hidden from the users and the application programs, unless they need to determine this information, in which case the information is provided through system services, provided the user has authorization for the same. The Distributed OS provides means for system-wide sharing of resources like computational capacity, I/O and files etc. Thus, it should also cater for the following services:

- Global Naming of resources
- Distribution of computations
- Enabling processes to access remote resources
- Enabling processes to communicate with remote processes

SUMMARY

- An operating system is a program that manages the computer hardware and it provides the interface between user of the computer and computer hardware. The operating system has been evolved from the simple batch systems to time shared systems.
- To improve the overall performance of the computer system, developers introduced the concept of multiprogramming so that several processes can be placed in memory at the same time. Multiprogramming also allows time sharing systems. It allows many users to share the computer simultaneously. It improves the performance of computer system by allowing overlapping between CPU and I/O operations on a single system.
- Operating system provides various services and on the basis of these services structure of the operating system can be developed. The operating system structure varies depending on the type of OS and thus the services provided by the OS. It is a layered product and different functionalities may be implemented in different layers.

SELF-ASSESSMENT QUESTIONS

1. Briefly explain the four major functions of an Operating System.
2. Give main features of the following types of OS, outlining their limitations and strengths:
   (*a*) Batch OS (*b*) Multi-programmed OS
   (*c*) Interactive Time-sharing OS (*d*) Real time OS
3. What do you mean by Batch Processing? Give its limitations.
4. Explain different types of operating systems.

❑❑❑

UNIT 2 FUNCTIONS AND STRUCTURE OF OPERATING SYSTEM

★ LEARNING OBJECTIVES ★

- Introduction
- Functions of an OS
- OS Components
- OS Services
- System Calls
- System Programs
- Interrupts
- OS Structure
- Multi-processing
- Summary
- Self-Assessment Questions

INTRODUCTION

An operating system performs different functions which is divided into sub-systems in accordance with the functions performed also an operating system provides an environment for the execution of user programs and provides certain services to the programs and to the users of those programs for accessing of system resources.

Operating system varies dependently upon the type of OS and thus the services provided by OS.

FUNCTIONS OF AN OS

The Operating System is a manager of System Resources, performing the following functions:

NOTES

1. Process Management
2. Main Memory Management
3. I/O Devices Management
4. File Management
5. Secondary Storage Management
6. Network Management
7. System Protection
8. Command Interpretation
9. Accounting of Resources Usage by Users in a Time-Sharing System

Process Management

A Process (also known as Task) is an instance of a program in execution. While a program is just a passive entity, a Process is an active entity performing the intended functions of its related program. It is the smallest unit of work that is independently schedulable. To accomplish its task, a process needs certain resources like CPU time, memory, files & I/O devices. These resources are allocated to the process either at the time of its creation of when it is executing. In a multi-programming environment, there will a number of simultaneous processes existing in the system; some of these will be system processes and others will be user processes.

OS Functions related to Process Management

The OS is responsible for the following functions, related to process-management:-

(*a*) Process Creation, which involves loading the program from Secondary Storage to Memory and commence its execution.

(*b*) Process scheduling or dispatching i.e transferring a Process from the Ready State to Run State, when it controls the CPU.

(c) Suspending a Process, which involves transferring the Process from Run State to Wait State. This is done when a Running Process Requests some I/O or Requests "Wait for some Event".

(*d*) Resuming a Process i.e. transferring it from Wait State to Ready State, when the requested I/O has been completed or the Event Awaited by the process occurs.

(*e*) Providing mechanisms for process synchronization for sharing of resources amongst concurrent processes.

(*f*) Providing mechanisms for inter-process communication, amongst concurrent processes.

(*g*) Providing mechanisms for deadlock handling.

(h) Process Deletion or Process Termination. This is done when either the Process has successfully finished its execution or when it has committed a fatal error and is so terminated forcibly by the OS. The resources held by the Process are freed and its PCB is deleted.

Memory Management

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. When a program terminates, the memory space occupied by this program is declared available and the next program can be loaded and executed. In multi-programming environment, multiple programs are maintained in the memory simultaneously. The OS is responsible for the following functions related to memory management:-

- Keeping track of which segment of memory is in use and by whom
- Deciding which processes are to be loaded into memory when space becomes available
- Allocation a de-allocation of dynamic memory
- Swapping-in and swapping-out of processes

I/O System Management

An OS will have Device Drivers to facilitate I/O functions involving devices like Keyboard, Mouse, Monitor, Disk, Floppy Drive, CD ROM, Printer etc. The Device Drivers are nothing but software routines that control respective I/O Devices through their controllers. A Device Driver hides the peculiarities of its I/O device.

A Device communicates with the CPU by sending & receiving signals over a cable or by radio. The device communicates with a host processor through a connection point, called a Port. Sometimes, the devices may be connected in Daisy Chaining fashion. In this arrangement, the first device is connected to the computer port, the second device is connected to the first device and so on. In this arrangement, any command from computer is received by the first device, which may, in turn, pass it to second device and so on. A controller is an electronic device that controls a port or a bus or a device; for example SCSI Bus controller. A controller may typically contain a processor, micro-code and some local memory to perform its assigned functions. Some devices have their own built-in controllers; for example disk drives. This implements the disk-side protocols. A controller has a set of registers for exchange of data and control signals/status information between the device and the host processor. Communication between host processor & device may be performed using special I/O Instructions. An alternative to this is

NOTES

Memory-mapped I/O, wherein the registers are mapped in the main memory itself and communication with the device can be accomplished by using standard read/write instructions. For example, a graphics controller has I/O Ports for exchange of control signals, and a large memory-mapped region to hold the screen contents. The host processor updates the screen contents in its memory itself, which is much faster than sending updates though special I/O instructions, since it will involve millions of instructions per second.

File Management

Computers can store information on different media like Hard Disk, Floppy, CD, Magnetic Tape etc. All these media have different characteristics in terms of physical organization, capacity, access methods, data transfer rate etc. But for convenient accessing of the information, the OS provides a uniform logical view of the information storage. The logical storage unit is called a File. A file is a collection of related information, defined by its creator. The files are organized into directories & sub-directories. The OS abstracts the logical unit 'File' from the characteristics of the underlying media. The OS is responsible for:

- Creating & deleting files
- Creating & deleting directories
- Support manipulation of files & directories
- Mapping files onto secondary storage
- Backing up files onto media like tapes

Secondary Storage Management

Computer systems use disks as principal on-line storage medium, for both programs and data. System utility programs like compilers, editors, linkers etc are kept stored on the disk and loaded into RAM as and when required. Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the disk access speed. The OS is responsible for following functions related to disk management:-

- Storage allocation
- Free space management
- Disk scheduling

In the case of Virtual Memory Management, the disk space acts as an extension of main memory. The address space of a process will be mapped partially in RAM and partially on secondary storage. The whole address space is divided in pages. The pages are moved between secondary storage and RAM, as needed. The size of a program can be larger than the RAM size

available. The available RAM is divided in frames and size of a frame is equal to a page. So, a page in RAM will map onto a frame. Whenever, page is needed by a process and is not found in memory, the situation is called a Page Fault. The page is moved from secondary storage to RAM in an available frame. This is called paging-in. When no free frame is available to accommodate a new page, then one of the memory resident pages is thrown out to the secondary storage. This is called paging-out. When a new process is to be moved into RAM and free space is not available in RAM, a memory resident process, that might have partially executed and may be currently inactive, is moved out to secondary storage. This is called swapping out. At some later moment, when conditions for its execution are satisfied, the swapped-out process may be swapped-in. This would resume its execution from the point where it was swapped out. This paging-in, paging-out, swapping and swapping-out is done under the control of operating system.

In addition, OS also performs disk scheduling, so as to optimize the movement of disk-head, for improving the I/O response time and increasing the disk-I/O throughput.

Networking

A distributed system is a collection of processors, which do not share memory, clock or peripheral devices. Instead, each processor has its local clock, and RAM, and communicate through a network. Access to a shared resource permits increased speed, increased functionality and enhanced reliability. Various networking protocols are TCP/IP, UDP/IP, FTP, HTTP (Hyper Text Transfer Protocol) and Network File System (NFS).

System Protection

System Protection refers to the mechanism for controlling the access to computer resources by various users and processes. If a computer system has multiple users and permits concurrent execution of multiple processes, the processes would need protection from each other and also the OS would need protection from the user processes. For this purpose, the OS ensures that contending process can gain access to shared resources only through OS and that too within the well-defined parameters. Some of the System Protection Features are:

- At a time, only one process is enabled to take control of the CPU.
- At a time, only process is permitted exclusive access to an I/O Device for performing I/O. The Device Registers not made directly accessible to the user processes. These are accessible only to the Device Drivers, which are part of OS.

- A Process access memory only within the segments allocated to it by the OS. Any attempt, to access memory outside the segments allocated to a process (termed as "Memory Access Violation") is prevented by the OS.
- A User can access a shared file only to the extent specified by the Access Rights granted to that user. Any attempt to access beyond the rights is denied by the OS.

Command Interpretation

It is interface between user and OS. Many commands are given to the operating system by control statements. When a new job is started in a batch system, or when a user logs onto a time-shared system, command interpreter is executed automatically. It reads & interprets the control statements. Command Line Interpreter is sometimes called a shell. The command line interpreter should be user-friendly.

System Command Vs System Call

A System Command provides an interface between an Interactive User and the OS. Whereas, System Calls are embedded in the Programs, to provide an interface between the Processes and the OS. For each System Command, there will be equivalent System Call. The System Calls for an OS are listed in Application Programs Interface (API) manual.

OS COMPONENTS

OS is divided into sub-systems in accordance with the functions performed. Thus, it comprises of the following components:-

Process Management Component

This component is responsible for various tasks related to Process Management:-

- Mechanism for Process Creation & Deletion
- CPU Scheduler
- Tools for Process Synchronization such as Semaphores
- Mechanism to support Inter-Process Communication
- Mechanism for Deadlock Handling

Memory Management Component

This includes algorithms for Memory Allocation/ De-allocation, Swapper, Pager etc to manage the Main Memory.

I/O Management Component

This includes I/O Drivers for various devices supported by the system and various I/O Modes (like Program-Controlled I/O, Interrupt Driven I/O, DMA etc) supported by the devices. This includes mechanism for Interrupt Servicing.

File Management Component

This includes mechanism for Creation/ Deletion of Files & Directories, mechanism to support primitives for manipulation of files & directories and mechanism to Back-Up the Files etc.

Secondary Storage Component

This includes mechanism for Allocation/De-allocation of Disk Space, Management of Free Space & Disk-Scheduling etc.

Network Management Component

This includes mechanism to Send/ Receive Messages through the Network-Ports supported by the system, using Networking-Protocols like TCP/IP etc.

Protection System

This includes mechanism for the protection of resources from the User Processes, protection of User Process from each other and protection of the OS from the User Process. This is achieved by ensuring that all System Resources are accessed by the User Processes only through OS. During access, the OS ensures that a User Process does not violate the Access Rights.

Command Interpreter

This component forms the outermost layer of the OS. It provides an interface between the System Command issued by Users and the OS Routines. This component intercepts the System Commands, interprets the commands and then invokes necessary OS Routines to service the commands.

Accounting Component

This component keeps account of the system resources used by various users in a time-sharing system and generates bills at pre-specified intervals. It can also keep track of the payments made and payments due.

OS SERVICES

An OS provides an environment for the execution of user programs. OS provides certain services to the programs and to the users of those programs

for the accessing of system resources. Some of the common services are:-

NOTES

Program Execution

The MAIN purpose of OS is to provide an efficient and convenient environment for the execution of programs. So, an OS must provide various functions for loading of a program into RAM and execute it. Each executing program must terminate-either normally or abnormally.

I/O Operations

A running program would need I/O operations for reading-in of input data and for outputting of results. This I/O may be from/to a file or from/to an I/O device. For each device, some special functions may be necessary (such as rewind a tape, clear screen etc). All these operations are managed by an OS.

File Manipulations

Each executing Program would need to create, delete & manipulate files, which is managed by OS.

Communications

OS manages inter-process communication between the processes, executing on the same computer or running on different computers in a distributed/multi-processor environment. An OS would provide mechanisms for this inter-process communication; like mailboxes, shared memory etc.

Error Detection & Recovery

Errors may occur during execution of a program; like divide by zero, memory access violation etc. The OS should provide for detection such errors (or exceptions) and handle recovery (called Exception Handling).

Resource Allocation

When multiple users are logged onto the system or multiple jobs are running concurrently, resources would need to be shared amongst them. The OS would decide on the allocation of resources; like the CPU Scheduling algorithm will determine the control of CPU amongst the concurrent processes. Similarly, there will be routines for the allocation & de-allocation of other resources like memory, I/O Devices, Files etc.

Accounting

It relates to the accounting of resources used by each user in a multi-user environment. Resource usage statistics may be useful in planning the futuristic requirements of an enterprise.

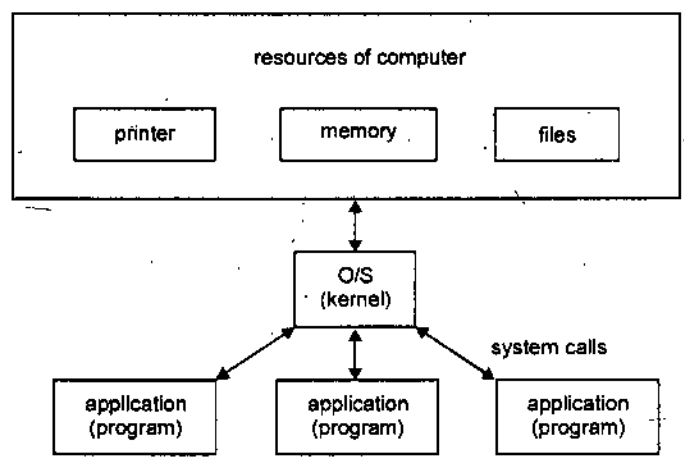
System Protection

This is also a service provided to ensure that the users and user processes are protected from each other. For example, when a process is having an exclusive access to a device allocated to it, no other process would be granted access to that device during that period. Also, no process would be permitted to access a memory location in a segment not assigned to that process. This service ensures that a process can execute safely.

SYSTEM CALLS

An operating system has two major jobs. It manages the resources of the computer and the application programs running on the computer. The user programs access the resources of the computer by calling functions that are executed by the kernel and are called as system calls. A collection of system calls is sometimes referred to as Application Program Interface(API).

System calls provide the interface between a process and the operating system. These calls are generally available as assembly language instructions. Several High level languages as C, C++, Perl allow system calls to be made directly.



When handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. For I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

There are five major categories of System calls:

(a) Process Control

- end and abort
- load, execute

NOTES

create process, terminate process

get process attributes, set process attributes

wait event, signal event

allocate and free memory

example: fork(), execl(), execv(), wait(), system() e.t.c.

(*b*) File Management

create file, delete file

open, close

read file, write file

get file attribute, set file attributes

example: open(), read(), write(), close(), creat() (yes there is no e - this is not a typo), lseek(), link() e.t.c.

(*c*) Device Management

request device, release device

read, write, reposition

get device attributes, set device attributes

logically attach or detach devices

(*d*) Information Maintenance

Get time or date, set time or date

Get system data, set system data

Get process, file or device attributes

Set process, file or device attributes

Examples: time(), setitimer(), settimeofday(), alarm() etc.

(*e*) Communications

create connection, delete connection

send message, receive message

transfer status information

attach or detach remote devices

example: signal(), pause(), kill() etc.

NOTES

SYSTEM PROGRAMS

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). System programs provide a convenient environment for program development and execution.

System programs can be divided into following categories:

File management: These programs manipulate files and directories.

Status Information: These programs can ask for the status information as date, time, available memory, number of users and so on.

File Modification: These programs can create and modify the contents of file, as text editors.

Programming-language support: These programs involve compilers, assemblers and interpreters.

Program loading and execution: The system provides absolute loaders, relocatable loaders, overlay loaders and linkage editors.

Communication: These programs create the virtual connection among users, processes or computer systems through e-mail, remote log in or transferring files.

INTERRUPTS

The occurrence of any event is signaled by the interrupt from either hardware or software. Hardware may generate the interrupt at any time by sending a signal to the CPU, by the way of system bus. Software may generate an interrupt by executing system call.

There are three types of interrupts that may cause a break in the normal execution of a program:

1. External Interrupts
2. Internal Interrupts
3. Software Interrupts

External Interrupts may cause from I/O devices, from a timing device, from a circuit monitoring the power supply or from any external source.

Internal Interrupts arises due to some illegal or erroneous use of an instruction or data as overflow, attempt to divide by user, stack overflow, some protection violation, due to premature termination of the instruction execution.

The difference between internal and external interrupts is that the internal interrupt is generated due to some exceptional condition caused by the program and not by the external events. Internal interrupts are synchronous

NOTES

with the program while external interrupts are asynchronous. If the program is rerun the internal interrupts will occur at the same place always and external interrupts are independent from the program execution.

External and internal interrupts are generated by the signals that occur in the hardware of the CPU. Software interrupt is initiated with the execution of the program. It can be used by the user for generating interrupt at any desired position in the program. For example when a user want to switch from a user mode to supervisor mode. When an input or output is required the supervisor call instruction is initiated to request the supervisor mode. This instruction causes the software interrupt.

When the CPU is interrupted, it stops what is doing and transfers the execution to the starting address where the service routine for the interrupt is allocated. Each computer design has its own interrupt mechanism. The interrupt must transfer control to the interrupt service routine by invoking a generic routine to examine the interrupt information and then the routine will call the interrupt specific handler. A table of pointers to interrupt routines can be used, as it is given that only a predefined number of interrupts is possible for handling the interrupts quickly. These locations hold the addresses of interrupt service routines for various devices. This array of addresses is indexed by a unique device number. It provides the address of the interrupt service routine for the interrupting device.

OS STRUCTURE

The OS varies depending upon the type of OS and thus the services provided by the OS. Normally Operating System is a layered product. Different functionalities may be implemented in different layers. The critical routines are implemented in the innermost layer, called Kernel. The Kernel is in direct control of the hardware. When the system is executing in the Kernel Mode (ie. Executing Kernel Routines), it will be manipulating critical system tables; so the system will impose many restrictions like disabling of Interrupts. Since, such restrictions should be imposed for minimal period, the routines in the Kernel are highly optimized.

Examples:-

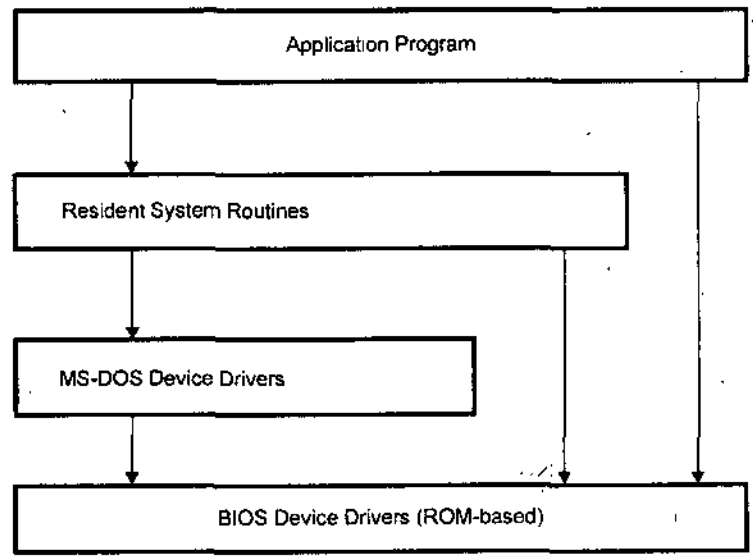
MS DOS Structure

The BIOS (Basic I/O System) is ROM resident and it contains I/O Drivers for basic devices like Keyboard, Monitor & Disk etc. The Bootstrap routine is also part of BIOS. The remaining MS-DOS Drivers and Resident System Routines are loaded during booting. Since, MS-DOS is designed as a single-

NOTES

user OS and does not need elaborate System-Protection features, the BIOS Routines can be directly accessed from Application Programs.

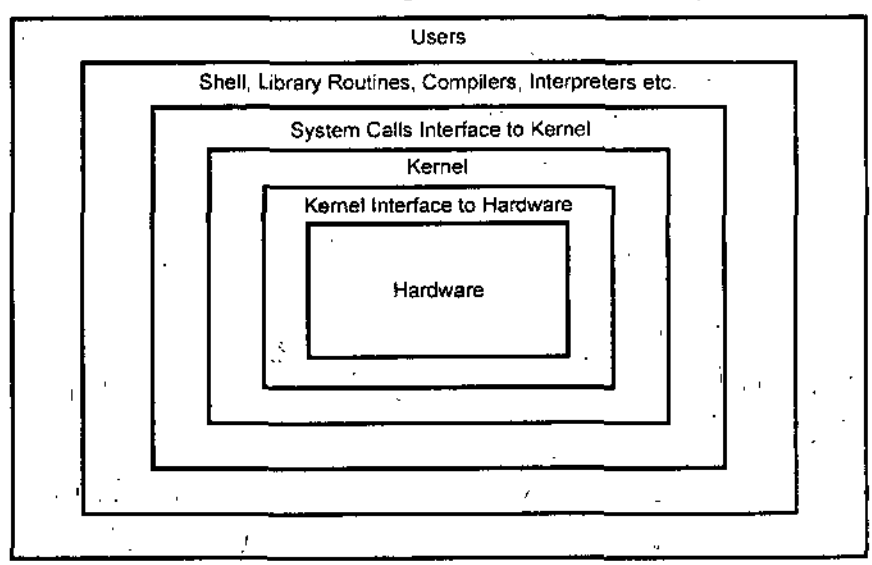
Some Designers design the System in such a way that BIOS is transferred to RAM at the time of booting, since RAM is faster than ROM. Then, access to BIOS is made only in the RAM. This technique is known as **"Shadowing"**.



UNIX STRUCTURE

UNIX is a layered Operating System. The Outermost Layer, which provides interface of System Commands/ System Calls with the OS, is called Shell (a Command Interpreter). It interprets the System Commands/ Calls and invokes necessary OS Routines to service the Commands.

The OS Kernel comprises I/O Drivers, CPU Scheduler, Pager, Swapper etc, which are interfaced with the Computer Hardware.



MULTI-PROCESSING

NOTES

It refers to the environment of multiple tightly-coupled processors exploiting the potential of parallel processing algorithms. Usually the parallel processors are executing the same code. While writing a program, the programmer has to indicate explicitly what segments of the code can be executed in parallel. The constructs for this mechanism are provided in the parallel-processing languages.

Some Examples of Parallel Processing Algorithms

Problem 1

The following expression describes the serial/parallel precedence relationship amongst six processes P_1 ... P_6.

$$P (S(P_2 , (P (P_3, S (P_1 , P (P_6, P_5)))), P_4)$$

where P implies Parallel & S implies Serial.

Transform the expression into program using:-

(*i*) Fork-Join Constructs (*ii*) Par-begin Par-end Constructs

Solution

First construct a Precedence Graph for the expression, as follows:-

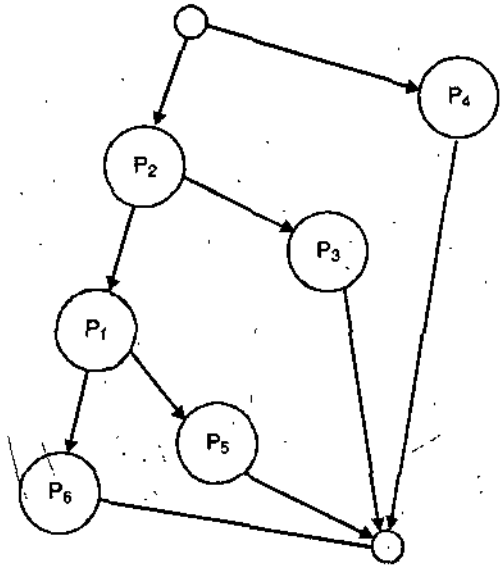
Precedence Graph

Program using Fork-Join Constructs

The Fork-Join Construct makes use of Fork & Join Instructions, which function as follows:-

FORK

When an executing Process P_1 executes instruction **FORK P_2** ; (where P_2 is



NOTES

of type Address), **a new sub-process is created, which starts executing in parallel with P_1, starting from Address P_2.**

JOIN

When a Process executes instruction *JOIN F;* (where F is of type Integer), the value of *F is decremented by 1; after decrementing if (F > 0) then the Process is deleted, but if (F= 0) then the process continues with execution of the next instruction.*

The above program can be coded using Fork-Join Construct as follows:

```
        F:= 7;
        fork P2 ;
        fork P4 ;
        goto FINISH;

P2:     <CODE OF Process P2>
        fork P1 ;
        fork P3 ;
        goto FINISH;

P4:     <CODE OF Process P4>
        goto FINISH;

P1:     <CODE OF Process P1>
        fork P6 ;
        fork P5 ;
        goto FINISH;
P3:     <CODE OF Process P3>
        goto FINISH;

P5:     <CODE OF Process P5>
        goto FINISH;

P6:     <CODE OF Process P6>
```

FINISH: join F;
end.

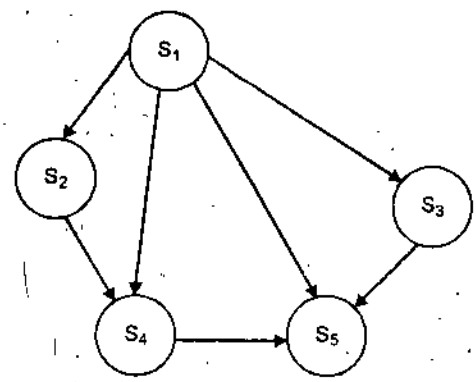
Program using Par-begin and Par-end constructs. The Par-begin and Par-end construct works as follows:

NOTES

The Processes P_1 and P_2 enclosed in a Par-begin and Par-end pair will execute in parallel.

So, the above program can be coded using Par-begin and Par-end as follows:

```
Parbegin
{       P2;
        Parbegin
            {   P1;
                Parbegin
                        P6;
                        P5;
                Parend;
            };
                P3;
                  Parend;
                };
                  P4 ;
Parend;
```



NOTES

Problem 2 (UPTU Exam - May 2002)

Write a concurrent program using Par-begin Par-end to represent the following precedence graph:-

Program using Par-begin Par-end Constructs

S_1;

Par begin

{ S_2 ;

S_4 ;

}

S_3 ;

Par end;

S_5;

Problem 3. *Write Pseudo-code for multiplication of two matrices A[M,N] and B[N,L] and output the resultant matrix C[M,L].*

Solution.

```
Read (A[M,N]);/* Read in Matrix A */
Read (B[N,L]);/* Read in Matrix B */
For I:= 1 to M do
   For J:= 1 to K do
            Parbegin
                     C[I,J]:= 0;
                     For K:= 1 to L do
                       C[I,J]:= C[I,J] + A[I,K] * B[K,J];
             Parend;
Write (C[M,L]);
```

STUDENT ACTIVITY

1. Describe functions of operating system.

2. What are the operating system components?

NOTES

SUMMARY

- A Process (also known as Task) is an instance of a program in execution. While a program is just a passive entity, a Process is an active entity performing the intended functions of its related program. It is the smallest unit of work that is independently schedulable.
- The Device Drivers are nothing but software routines that control respective I/O Devices through their controllers. A Device Driver hides the peculiarities of its I/O device.
- Computers can store information on different media like Hard Disk, Floppy, CD, Magnetic Tape etc. All these media have different characteristics in terms of physical organization, capacity, access methods, data transfer rate etc.
- A distributed system is a collection of processors, which do not share memory, clock or peripheral devices. Instead, each processor has its local clock, and RAM, and communicate through a network.
- If a computer system has multiple users and permits concurrent execution of multiple processes, the processes would need protection from each other and also the OS would need protection from the user processes.
- A System Command provides an interface between an Interactive User and the OS. Whereas, System Calls are embedded in the Programs, to provide an interface between the Processes and the OS.
- An operating system has two major jobs. It manages the resources of the computer and the application programs running on the computer.
- The occurrence of any event is signaled by the interrupt from either hardware or software. Hardware may generate the interrupt at any time by sending a signal to the CPU, by the way of system bus. Software may generate an interrupt by executing system call.

SELF-ASSESSMENT QUESTIONS

1. Describe the main components of an OS?
2. What types of OS will meet requirements under the following environments?
   (*a*) A large mainframe computer, made available for software development to a team of about 100 programmers.
   (*b*) A system receiving inputs from sensors and controlling Air Defence Weapons.

NOTES

(c) A system used for process control in an oil refinery.

(d) Multi purpose fuel injection system used in vehicles.

(e) A system used for processing of information stored on magnetic tapes and producing hard copies of output.

3. Explain Interrupt Servicing mechanism, clearly bringing out the role of OS in that.
4. Explain the salient features of Program-controlled I/O, Interrupt-Driven *I/O* and *DMA, highlighting their relative merits & demerits.* What is spooling?
5. Explain the concept of a "Process". With the help of a State Transition Diagram, explain various states of a Process, during its execution in a Multi-Programming environment. What is Daemon?
6. Explain the concept of Threading? How does a "Thread" provide a more efficient mechanism as compared to a full-fledged "Process"?
7. *Write Pseudo Code, using Parbegin-Parend, to exploit inherent* parallelism in the following problem (Don't care about the Library Routines):-

   Read in 1000 Integers into an array A[1000]

   Call a Library Routine to determine Avg (A)

   Call a Library Routine to determine Max (A)

   Determine Ratio = Avg (A) / Max (A)

   Write out Ratio

8. The following expression describes the serial/parallel precedence relationship amongst four processes P_1 ... P_4.

   P (S (P_2 , P (P_3, P_4)), P_1)

   where P implies Parallel & S implies Serial.

   (a) Draw Process-Precedence Graph.

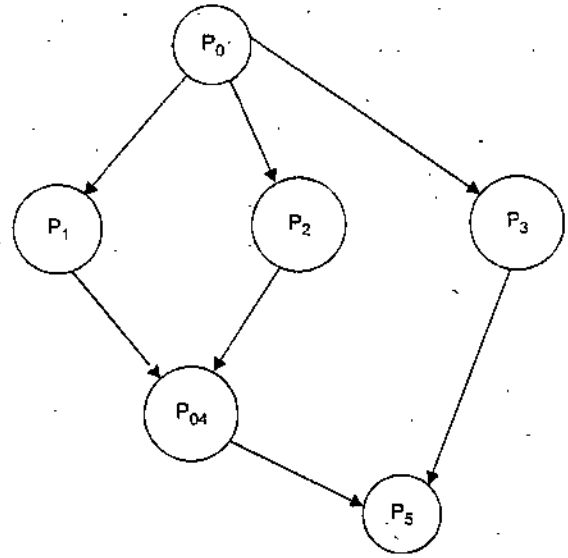
   (b) Transform the expression into program using:-

   (i) Fork-Join Constructs

   (ii) Par-begin Par-end Constructs

9. Write programs using "Fork-Join" and "Parbegin-Parend" to represent the following Process Precedence Graph.

NOTES



SECTION B

UNIT 3 I/O SYSTEM MANAGEMENT

NOTES

★ LEARNING OBJECTIVES ★

- Introduction
- Concept of Hardware Interrupt
- Concept of Software Interrupt (Trap)
- Various Types of I/O Modes
- Concept of Spooling
- Disk I/O
- Disk Scheduling
- Various Disk Scheduling Algorithms
- Summary
- Self-Assessment Questions

INTRODUCTION

An OS will have device drivers to facilitate I/O functions involving devices like Keyboard, Mouse, Monitor, Disk, Floppy drive, CD ROM, Printer etc. The device drivers are nothing but software routines that control respective I/O devices through their controllers. A device driver hides the peculiarities of its I/O device.

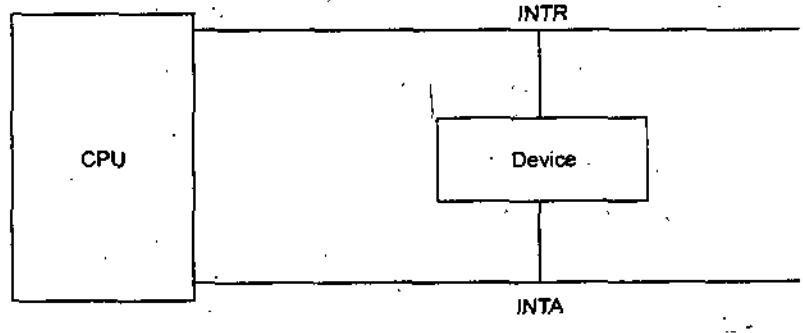
A Device communicates with the CPU by sending and receiving signals over a cable or by radio. The device communicates with a host processor through a connection point, called a Port. Sometimes, the devices may be connected in Daisy Chaining fashion. In this arrangement, the first device is connected to the computer port, the second device is connected to the first device and so on. In this arrangement, any command from computer is received by the first device, which may, in turn, pass it to second device and so on. A controller is an electronic device that controls a port or a bus or a device; for example SCSI Bus controller. A controller may typically contain a processor, micro-code and some local memory to perform its assigned functions. Some devices have their own built-in controllers; for example disk drives. This implements the disk-

side protocols. A controller has a set of registers for exchange of data and control signals/status information between the device and the host processor. Communication between host processor and device may be performed using special I/O Instructions. An alternative to this is memory-mapped I/O, wherein the registers are mapped in the main memory itself and communication with the device can be accomplished by using standard read/write instructions. For example, a graphics controller has I/O ports for exchange of control signals, and a large memory-mapped region to hold the screen contents. The host processor updates the screen contents in its memory itself, which is much faster than sending updates though special I/O instructions, since it will involve millions of instructions per second.

CONCEPT OF HARDWARE INTERRUPT

It is a mechanism by which a device draws the attention of CPU. A Device is connected to CPU through a Bus. The Bus comprises address lines, Data Lines and Control Lines. One of the control lines is called Interrupt Request (INTR) Line, which is "Active Low" and an Interrupt Acknowledge (INTA) Line, which is "Active High".

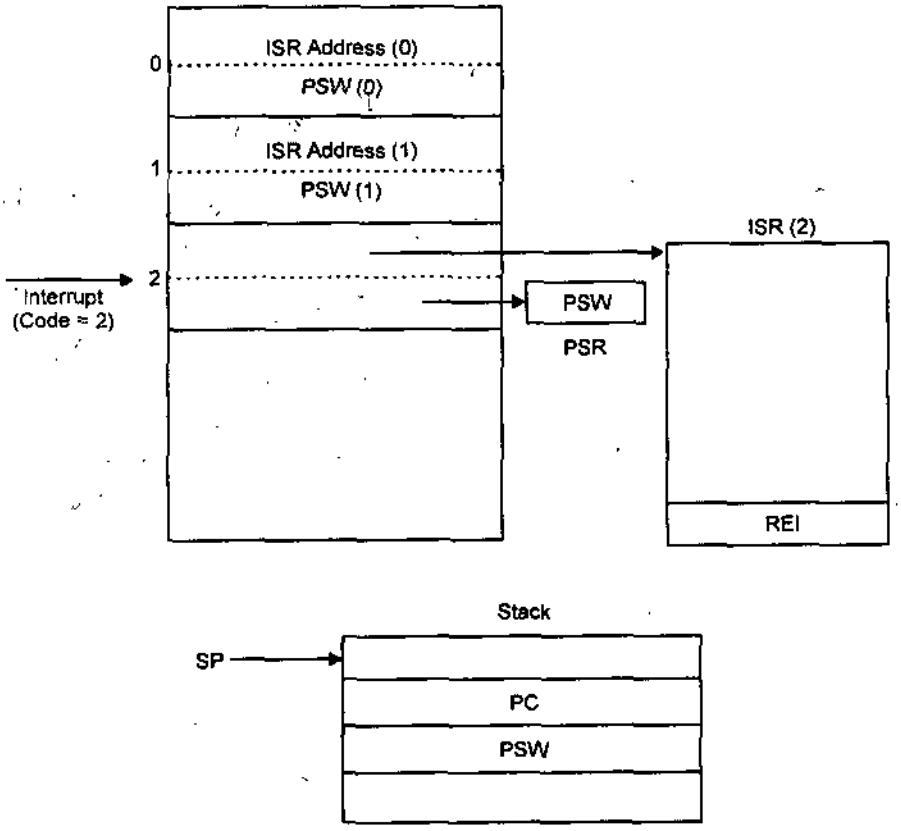
Initially, when the device has nothing to convey to the CPU, the INTR is "High" (1) and the INTA line is "Low" (0). When the Device needs to convey something to the CPU, it will bring the INTR line to Low (0). The CPU will be monitoring the INTR line after execution of each instruction. When the INTR line is found to be "Low", it means the Device wants to "Interrupt" the CPU. The CPU pauses to service the Interrupt. It reads the Data Bus to determine the "Code" of the Interrupt. The "Code" will be used to determine the respective Interrupt Service Routine (ISR) as explained in the "Interrupt Servicing" below. Then the CPU will acknowledge the Interrupt by raising INTA to "High" (1). When the Device monitors that the INTA line has gone "High", it indicates that the Interrupt has been attended to by the CPU. Then, the device turns the INTR line to "High" and INTA line to "Low" - so, it is back to normal.



Interrupt Servicing Mechanism

NOTES

The OS maintains a table called Interrupt Vector Table (IVT), which is stored at the lower end of main memory (location 0 onwards). Each entry in the IVT contains an ISR address and a Processor Status Word (PSW). The PSW contains processor status that should when the respective ISR is being executed. The IVT is indexed by the Interrupt Codes.



So with the help of Interrupt Code, the OS locates the respective ISR Address and PSW in the IVT. The current values of Program Counter (PC) and Processor Stats Register (PSR) are saved on the stack. Then the PSW value from the respective vector of IVT is loaded into the PSR and ISR Address is loaded into PC. Thus, the program control is now transferred to the ISR. The execution of the ISR results in servicing of the Interrupt Request. The last instruction of ISR is REI (Return from Interrupt). This results in popping the PSW & PC value from the stack and loading to PSR & PC respectively. So, the program control returns to the process that was running at the time of Interrupt Request.

CONCEPT OF SOFTWARE INTERRUPT (TRAP)

This refers to the Interrupt caused by a System Call executed by a running process or by an exception occurred during execution of a process. When a

running process makes a System Call (say to perform an I/O) it results in a Software Interrupt (Trap). Just like Hardware Interrupt, the Trap provides a Code for the accessing the ISR through the IVT. Further processing is exactly similar to Hardware Interrupt.

I/O Port

I/O Port provides necessary interface between CPU and the Device Controller. An I/O Port is a set of registers through which a Device Driver on the CPU side interacts with a Device Controller on the Device Side. A typical I/O Port comprises of the following four registers:-

(*a*) *Command Register.* This is used by the host processor to pass I/O commands (like read, write etc) to the connected I/O Device.

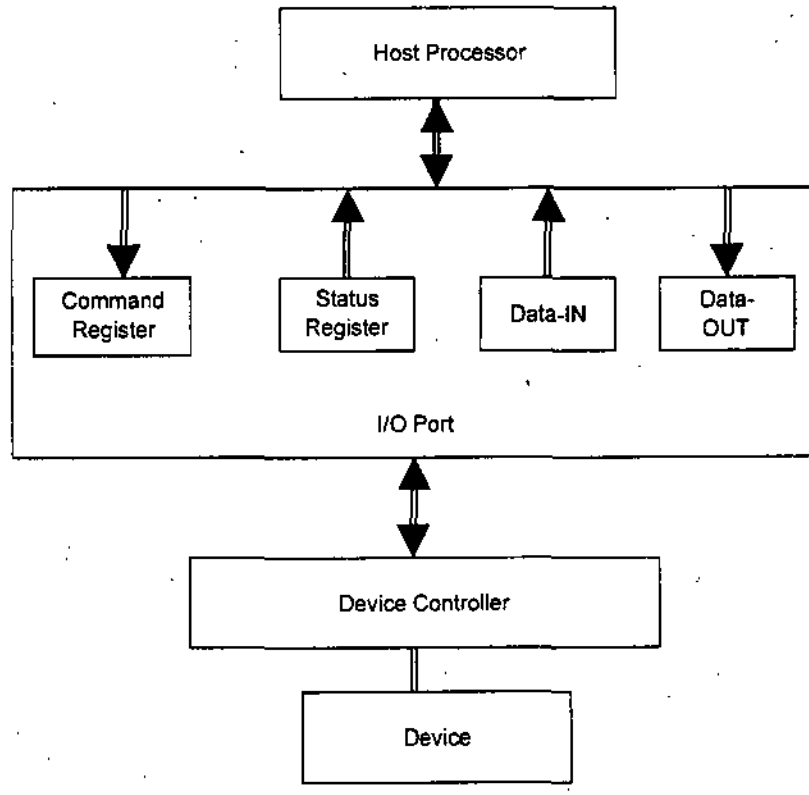
(*b*) *Status Register.* This is used to obtain the status (like device-busy, device-free etc) of connected device.

(*c*) *Data-in.* This register accommodates the data being read-in from the I/O Device.

(*d*) *Data-out.* This register accommodates the data being written-out to the I/O Device.

Memory Mapped I/O Vs I/O Mapped I/O

If the I/O Registers are mapped as part of the main memory itself then it is called Memory Mapped I/O. But if the I/O Registers are with the Device



Controller then it is termed as I/O Mapped I/O. The accessing of Memory Mapped I/O Registers is performed by standard Memory Read/write Instructions and it is very efficient. However, the accessing of I/O Mapped Registers is done using special IN/OUT instructions which is comparatively much slower.

NOTES

VARIOUS TYPES OF I/O MODES

There are three types of I/O:-

1. Program-Controlled I/O
2. Interrupt Driven I/O
3. Direct Memory Access (DMA) Transfer

1. Program-Controlled I/O

When a Running Process issues an I/O Call (say to write onto a Device), the Context of the Process (i.e. PC & PSW) are saved on the stack and the program control is transferred to the ISR. The ISR running on the host CPU will proceed as follows:

(*a*) The host will repeatedly monitor the BUSY bit of the Device STATUS Register, till the bit is CLEAR. It indicates that now the device is free to start next I/O.

(*b*) The host then sets the WRITE bit in the Device COMMAND Register and writes the Byte of Data to be written out into the Device DATA-OUT Register.

(*c*) The host then sets the COMMAND READY bit in the Device COMMAND Register.

(*d*) During this time, the controller was continuously monitoring the COMMAND READY bit. Moment, the COMMAND-READY bit is found to be set, the controller knows that there is some command to initiate I/O. So, the Controller responds by setting the BUSY bit in the STATUS register. This indicates that now the device is busy performing an I/O.

(*e*) The Controller then reads the COMMAND register to determine the Command. So, it will notice the WRITE bit having been set.

(*f*) Then, the Controller reads the DATA-OUT register and outputs the Byte of Data onto the Device.

(*g*) Now, the controller clears the COMMAND-READY in the COMMAND Register & also clears the BUSY bit in the STATUS register. In case of I/O error, the Controller will set the ERROR bit in the Device STATUS Register.

(*h*) During this time, the Host will be monitoring the STATUS register. When BUSY bit is cleared and if ERROR bit is not found set, it indicates I/O Success, else it indicates I/O Failure.

It is called Program-Controlled I/O as the entire is performed under the direct control of Requesting Process. The ISR functions as a sub-routine of the Requesting Process. This type of I/O is extremely in-efficient on account of the following:-

(*a*) The CPU wastes its cycles waiting for the "Busy" bit of the Status Register to be cleared- both at the time of initiating an I/O or waiting for the completion of an I/O. This is called "Polling" or "Busy Waiting", which wastes the CPU time.

(*b*) The CPU performs I/O byte-by-byte. For each byte of data transfer, it has to initiate a fresh I/O. Thus it is not at all suitable for transfer of a large block of data between memory and a Device.

2. Interrupt-Driven I/O

This solves the "Busy-Waiting" problem of Program-Controlled I/O. It works as follows:

(*a*) When a User Process issues an I/O Call, the User Process is put to Wait State.

(*b*) CPU Control is transferred to a System Process, which executes the ISR.

(c) *Initiation of I/O.* The System Process will sense the "Busy" bit of Status Register. If it is free, the System Process will initiate I/O and go to Wait State. If the bit is set, still the System Process goes to Wait State; it will initiate I/O when device gets free and intimates the CPU through Interrupt. After Initiating I/O, the system process goes to Wait State and another Ready Process is scheduled to Run.

(*d*) *Completion of I/O.* When I/O is completed, the Device Controller will intimate the event to the CPU. Then the System Process responsible for the I/O is scheduled to Run. If it is Read operation, then it will transfer the data read from the Device to the local buffer of the User Process.

(*e*) Now since I/O has been completed, the requesting User Process is brought out of the Wait State and put in the Ready Queue. Now, it waits for assignment of CPU.

So, in Interrupt Driven I/O, the problem of Busy-Waiting of CPU has been eliminated. But, still the CPU has initiate I/O for each byte of data to be transferred.

NOTES

3. Direct Memory Access (DMA) Transfer

In the Programmed I/O or Interrupt Driven I/O, the CPU has to initiate the I/O for each byte of data to be transferred between a Device and Memory. This keeps the CPU continuously engaged during the entire I/O operation. This method is unsuitable for transferring large data between a Device and Memory. Imagine transferring a long file from secondary storage to RAM. This will keep the CPU tied up to the transfer from an unduly long time. A viable alternative is to go for a mechanism, wherein the CPU just initiates the transfer and then the data transfer is performed by some intelligent controller, without any further intervention of the CPU and when the transfer is completed, the CPU is intimated accordingly, by the controller through an Interrupt. This is precisely called Direct Memory Access (DMA) transfer.

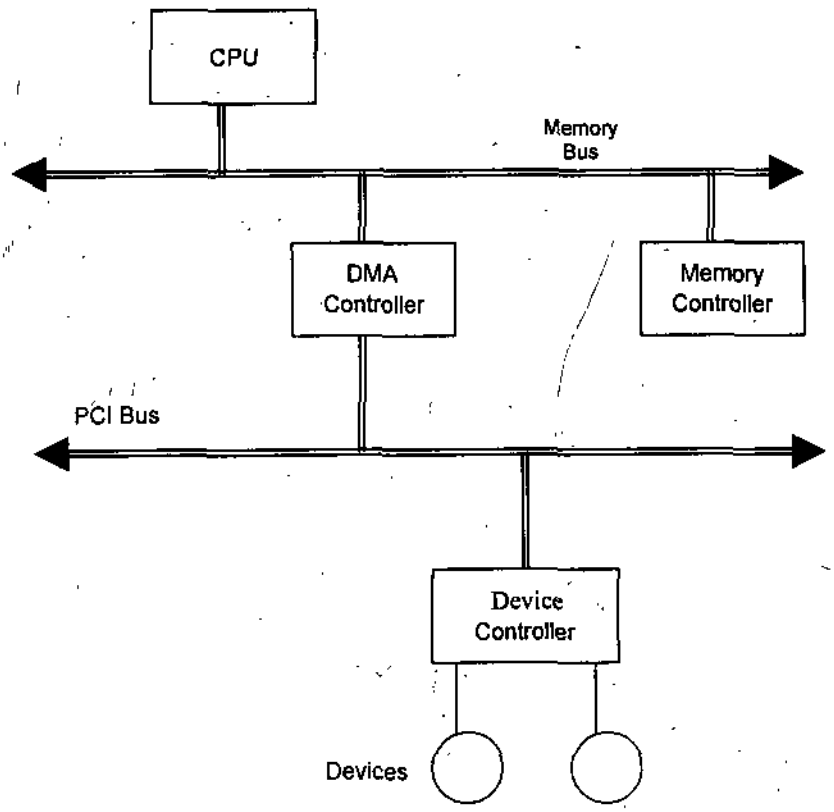
In DMA transfer, a device can directly access the memory, without intervention of CPU. Whenever, the host has to transfer a block of data between the memory and a device, it proceeds as follows:

(a) The Host writes a DMA Command Block into the memory. The DMA Command Block has following information:-

  (*i*) Memory Start Address for DMA transfer

  (*ii*) Number of Bytes to be transferred

  (*iii*) Direction of transfer, whether from Memory to Device or vice-versa.



NOTES

(b) Then the Host passes the start address of DMA Command Block to the DMA Controller.

(c) On receiving the start address of DMA Command Block, the DMA Controller reads the information and proceeds to progress Data Transfer, independent of CPU. For this, it shares the control of Memory Bus with the CPU. This is called 'Cycle-Stealing' and will slightly affect the CPU performance during the DMA transfer.

(d) When the intended Data Transfer is completed, the DMA Controller interrupts the Host Process to intimate the DMA completion. Then, the Host Processor can initiate another DMA transfer. Thus, the CPU does not remain tied to the device and is free to perform other tasks, when data transfer is in progress in parallel.

CONCEPT OF SPOOLING

SPOOLING is acronym for Simultaneous Peripheral Operations Online. Suppose a Line Printer is to be used to accept the print outputs of concurrently running processes in a Multi-Programming Environment, the resulting print-out will contain mixed-up output of the processes. Such an output will be of no use to the users. This problem is solved by the concept of SPOOLING. Under this scheme, a separate spool file is created for each process. The print outputs of a process are directed to its respective spool file (not to the printer). When a process is terminated or it explicitly indicates that it has finished its print commands, a Printer Daemon gets activated. The Printer Daemon closes the respective file of the terminated Process and queues it for printing out by the printer. The queued files will be printed out by the printer, one-after-another, without mixing up the outputs of different processes.

DISK I/O

Magnetic disks form the main secondary storage media for computer systems. A Hard Disk Drive consists of a number of round disk platters mounted on a spindle. Both surfaces of the platter are covered with a magnetic material capable of storing information. The information stored is not volatile. Each disk platter has a number of concentric tracks for recording of data and each track is divided into a number of sectors. A read-write head is positioned above/below each surface. The heads are attached to a disk arm, which moves all the heads together. The set of tracks that are at same radius form a cylinder. So, the number of disk cylinders will be equal to the number of tracks on each platter.

STUDENT ACTIVITY

1. Explain the function of I/O Port.

2. Describe various types of I/O modes.

When the disk drive is ON, the spindle rotates at a high speed (of the order of 12,000 RPM). To read/write some data from/to a sector, its address will consist of (Cylinder Number, Track Number, Sector Number). A Read/Write wll involve following steps:

(*a*) First moving the disk arm such that heads are positioned at the specified Cylinder.

(*b*) Wait till the specified sector is directly above/below the read/write head.

(*c*) As the sector of the specified track is rotating over/below the specified head, read/write the data.

So, disk read/write access time has following components:-

(*a*) Time T_1 to align the head with the cylinder by moving the arm in or out. This time wi" depend on the extent of movement required. This movement is measured as the number of cylinders to be traversed by the read/write heads. This is known as Seek Time.

(*b*) Time T_2 required by the platters for movement of the specified sector above/below a read/write head, after the arm has been positioned at the desired cylinder. This time, on the average will be equal to the time taken by half rotation i.e. 0.5/Rotation Rate. This time is called rotational latency.

(*c*) Time T_3 taken for actual transfer of data, after the head has moved exactly above/below the start of specified sector. This will also be inversely proportional to the rotation rate.

DISK SCHEDULING

As indicated above, the T_2 and T_3 components of disk access time depend purely on the underlying hardware technology and are beyond the control of OS. However, if pending Disk I/O Requests are scheduled in such a way that the total movement of disk arm is minimal, the average value of Seek Time T_1 will be reduced. This will reduce the average access time. This is precisely the spirit behind the concept of Disk Scheduling.

VARIOUS DISK SCHEDULING ALGORITHMS

1. First Come First Served (FCFS)
2. Shortest Seek Time First (SSTF)
3. SCAN
4. Circular SCAN (C-SCAN)

5. Seek (or Look)
6. C-Seek (or C-Look)

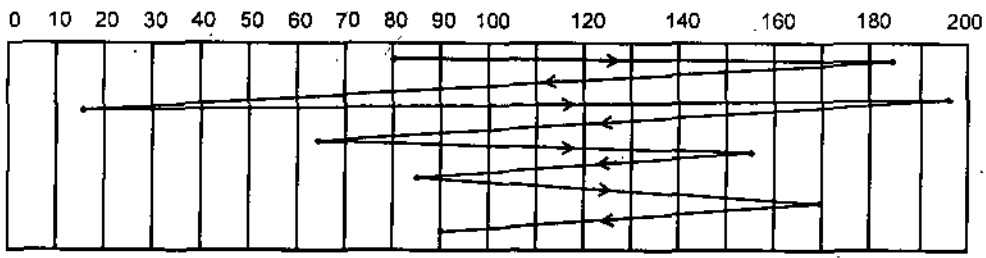
NOTES

1. FCFS Scheduling

The I/O Requests are serviced strictly in the save order as they are received. This algorithm is most fair, but the average seek time will be worst.

Example. Suppose the disk has maximum 200 cylinders (numbered from 0 to 199) and the disk arm is currently at cylinder number 80. Suppose the pending requests are for cylinder numbers 185, 15, 195, 65, 155, 85, 170, 90 received in that order. (For comparison of performance of various algorithms, we would use the same data for all.)

Disk arm movement pattern for FCFS

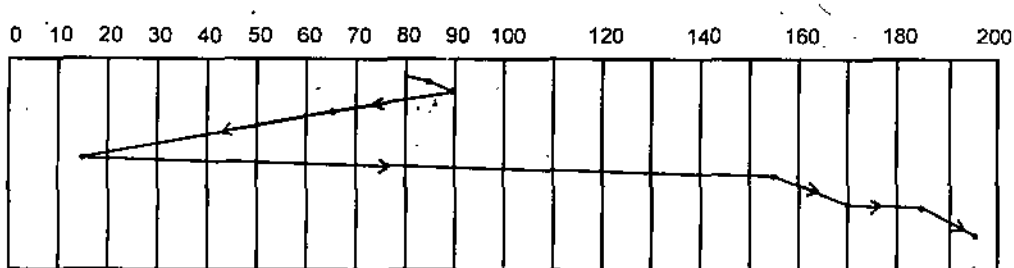


Total Head Movement = (185 – 80) + (185 – 15) + (195 – 15) + (195 – 65)
+ (155 – 65) + (155 – 85) + (170 – 85) + (170 – 90)
= 105 + 170 + 180 + 130 + 90 + 70 + 85 + 80
= **900 cylinders**

2. SSTF

The next request to be serviced will be the one that involves minimum seek time from the current position of the disk arm.

Disk arm movement pattern for SSTF



Total Head Movement in SSTF = (90–80) + (90–15) + (195–15)
= 10 + 75 + 180 = **265 cylinders**

So, its performance is much better than FCFS.

3. SCAN

The disk arm keeps scanning between the two extremes (cylinder 0 to 199 and back) continuously. It keeps servicing the requests encountered on the way. It is also called "Elevator Algorithm" since disk arm keeps moving like an elevator.

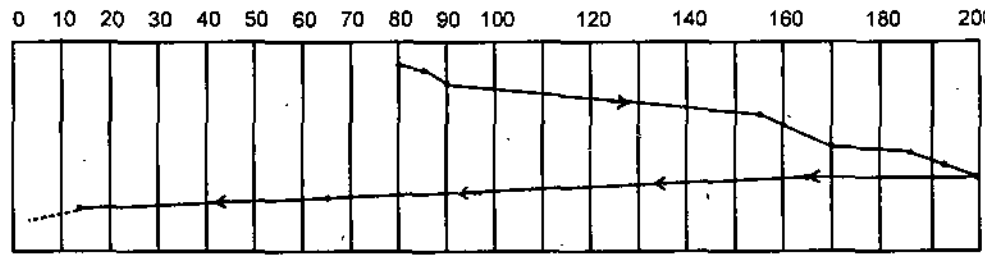
It has following disadvantages:-

(*a*) The disk arm keeps scanning between the two extremes, irrespective of the fact whether there are any requests pending or not. So, it would cause extensive wear and tear of the disk assembly.

(*b*) The requests that arrive just ahead of the arm position would get serviced immediately; but the requests that arrive just behind the arm position would have to wait for the arm to return back. So, this algorithm is not fair. This limitation is removed in the C-SCAN algorithm.

Disk arm movement pattern for SCAN

Assuming the head is initially moving towards track number 200.



Total Head Movement in SCAN = (199–80) + (199–15)

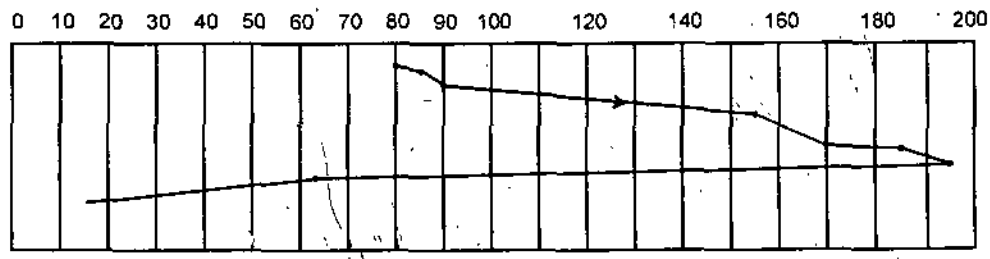
= 119 + 184 = **303 cylinders**

4. SEEK

Here also, the disk arm moves once from cylinder 0 side to cylinder 199 side and then back to cylinder 0 side and keeps servicing requests on the way. But there is one difference with SCAN that arm goes only as far as there is a request to be serviced (not up to the extremes).

Disk arm movement pattern for Seek

Assuming the head is initially moving towards track number 200.



Total Head Movement in SEEK = (195–80) + (195–15)

= 115 + 180 = **295 cylinders**

This algorithm is much superior to SCAN, since disk arm will move only if there are requests to be serviced, not otherwise.

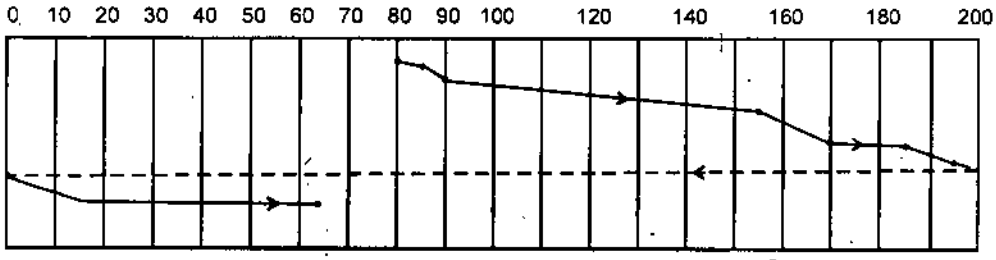
NOTES

5. C-SCAN

This algorithm is a modification of SCAN algorithm, wherein the arm keeps scanning between the two extremes, but it services requests only in one direction (say while going from 0 to 199). On the return path, the arm swings back to the other extreme, without servicing any requests on the way. So, the requests which just miss the arm during the previous scan would get serviced earlier than in the case of SCAN algorithm. So, it is more fair as compared to SCAN.

Disk arm movement pattern for C-SCAN

Assuming the head is initially moving towards track number 200.



Total Head Movement in C-SCAN = (199–80) + (65–0)

= 119 + 65 = **184 tracks**

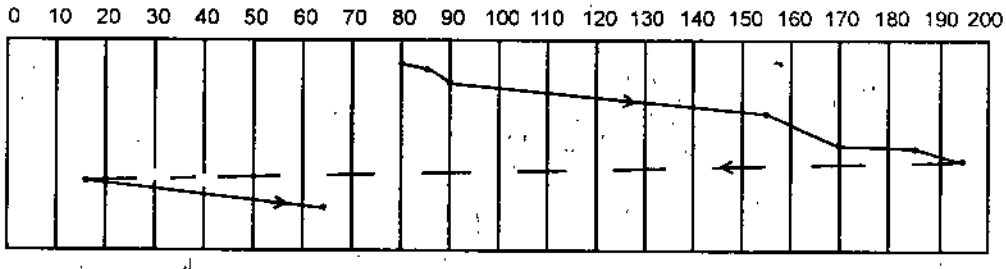
(the time taken for back swing has been ignored)

6. C-SEEK

This is a modification of seek algorithm, similar to C-SCAN. Here, the arm goes only as far as the requests to be serviced and requests are serviced only in one direction.

Disk arm movement pattern for C-SEEK

Assuming the head is initially moving towards track number 200.



NOTES

Total Head Movement in C-SEEK = (195–80) + (65–15)

= 115 + 50 = **165 cylinders**

C-SEEK is the best algorithm, since in involves minimum movement of the disk head to service all the requests. Disk head will not move, if there is no pending request, thus reducing wear & tear of the disk.

SUMMARY

- A Device is connected to CPU through a Bus. The Bus comprises address lines, Data Lines and Control Lines. One of the control lines is called Interrupt Request (INTR) Line, which is "Active Low" and an Interrupt Acknowledge (INTA) Line, which is "Active High".
- The OS maintains a table called Interrupt Vector Table (IVT), which is stored at the lower end of main memory (location 0 onwards).
- When a running process makes a System Call (say to perform an I/O) it results in a Software Interrupt (Trap).
- I/O Port provides necessary interface between CPU and the Device Controller. An I/O Port is a set of registers through which a Device Driver on the CPU side interacts with a Device Controller on the Device Side.
- In DMA transfer, a device can directly access the memory, without intervention of CPU.
- SPOOLING is acronym for Simultaneous Peripheral Operations Online. Suppose a Line Printer is to be used to accept the print outputs of concurrently running processes in a Multi-Programming Environment, the resulting print-out will contain mixed-up output of the processes. Such an output will be of no use to the users. This problem is solved by the concept of SPOOLING.
- A Hard Disk Drive consists of a number of round disk platters mounted on a spindle.

SELF-ASSESSMENT QUESTIONS

1. Explain the concept of Spooling. Why are output files for the printer normally spooled on disk before being printed, instead of being directly from the application program?
2. Briefly explain the various disk scheduling policies.
3. What is the Interrupt Service mechanism? Explain the concept of Software interrupt.

4. Consider the process requesting to read from the following tracks:

   98, 183, 37, 122, 14, 124, 65, 67

   Assuming the head is initially at track number 98

   (*a*) Draw track chart for FCFS, SSTF, SCAN, SEEK, C-SCAN & C-SEEK algorithms for disk scheduling.

   (*b*) Determine total head movement in tracks in each case.

   (*c*) Which is the best algorithm?

❑❑❑

NOTES

NOTES

UNIT 4 FILE MANAGEMENT

★ LEARNING OBJECTIVES ★

- Introduction
- Major OS Functions in File Management
- *File Operation*
- Directory Structure
- Disk Space Allocation Methods
- Free Disk-Space Management
- File Access Methods
- Sharing of Files
- File Protection Mechanism
- Summary
- Self-Assessment Questions

INTRODUCTION

For all types of secondary storage media, OS provides a uniform logical view of information storage. The logical unit for such storage is called a **File,** which is abstracted from the physical properties of the underlying device, on which it is stored. A File can be viewed as a named collection of related information, mapped onto a secondary storage device. All secondary media are non-volatile storage of information, where the contents persist, even when power is turned off. A file could contain any kind of information; like source code, object code, executable binary code, textual data, numeric data, graphics information or audio data etc.

File Attributes

Necessary Attributes of a file are:

NAME. It is a string of alphanumeric characters and some special characters like underscore. Most of the systems expect an alphabet as the first character of a file name and some systems are case sensitive to the file names. It is a displayable parameter, which is used for referencing a file.

NOTES

IDENTIFIER. It is a unique identification of a file, which is internal to the system. OS uses it for making references to the related file.

TYPE. It is normally expressed as an extension to file name and indicates the type of file; like A.cpp indicates that it is a C++ Source Code File, A.obj indicates that it is an object file and A.doc indicates that it is MS WORD file.

LOCATION. This is expressed as access path of the file for locating it on the device like C:/WINDOWS/all files/A.cpp

SIZE. It indicates current size of the file in bytes or blocks.

PROTECTION. This is Access Control Information. It indicates 'WHO' has got 'WHAT' Access Rights on the file i.e. who is owner of the file, who all are allowed to only read the file and who all are permitted to modify, delete or execute etc.

TIME, DATE, USER IDENTIFICATION. Records of information pertaining to the creation and last update of file.

VERSION NUMBER. This indicates version number of a file, along with creation/ update date/time.

File System

The file system consists of two distinct sub-components:-

(*a*) Collection of files, each storing related data.

(*b*) Directory Structure under which the files are organized. This provides information regarding all the files in a system.

File Organization

File organization refers to the manner in which the records of a file are organized on the secondary storage.

- A file is a set of logical records.
- It is allocated a disk storage space in terms of physical blocks (block size is normally 512 bytes).
- All basic operations like read, write etc are in terms of blocks.
- Last block of a file may not be fully occupied, resulting in loss of some storage space, called **Internal Fragmentation.** Average disk space lost due to internal fragmentation is of the order of half block per file. So, larger the block size, larger will be the disk space lost due to internal fragmentation. There is no way of recovering this loss.

File Control Block (FCB)

This contains complete information about a file; like its name, ownership, size, pointer to the storage blocks where the file is stored etc. Initially, it will

be residing on secondary storage, but when the file is opened, its FCB is loaded into memory.

File-Organization Schemes

Sequential. The file records are stored in the same order as they occur physically in the file.

Direct. The records are placed in any order, which is suited for application. The system supports random or direct access of any record in the file.

Indexed. The records are arranged in a logical sequence according to a 'key' contained in each record. The system maintains an index, which contains the physical address of some of the records.

Partitioned. This refers to a set of sequential sub-files, which together form a partitioned file. Each sequential sub-file is called a member of the partitioned file.

MAJOR OS FUNCTIONS IN FILE MANAGEMENT

1. Creation, manipulation and deletion of files as well as directories.
2. Protection of File System. OS controls the Access Rights of files & directories
3. Control sharing of files
4. Support backup & recovery of files
5. Support Encryption & decryption of sensitive files

Implementation of a File System

File system implementation has two major components- one is disk-based and the other is memory-based.

1. *On-disk structure.* The on-disk structure includes the following:
   (*a*) *Partition Control Block.* For each partition, it contains details such as the number of blocks in the partition, size of each block, number of free blocks, pointer to the free blocks' list, number of free File Control Blocks (FCBs) and pointer to the free FCBs.
   (*b*) *Directory Structure.* The structure, in which the files are organized.
   (*c*) *File Control Blocks (FCBs).* Each file has a File Control Block (FCB), which contains necessary information about the file; like its ownership, access rights and pointers to the disk blocks occupied by the file etc.

(*d*) *Boot Control Block*. It contains information needed by the system to boot an OS from that partition. It is typically the first block of a partition.

2. *In-memory Structure*. The in-memory structure contains following information:

   (*a*) *Partition Table*. It contains information about each mounted partition.

   (*b*) *Directory Structure*. It contains information about the recently accessed directories, along with the pointers to partition table.

   (*c*) *System-wide 'Open-File-Table'*. Contains a copy of the FCB of each open file. Each entry in the table will also contain a *"File-Open-Count"*, which indicates the number of processes, currently accessing the file.

   (*d*) *Per-process 'Open-File-Table'*. This table contains a pointer to the appropriate entry in the System-wide 'Open-File-Table'. It also contains a *Pointer* that points to the file location, where it would be accessed next. All operations are performed through this pointer only. Note that different processes accessing a file concurrently will have different pointers. But these operations must be compatible with each other.

FILE OPERATIONS

1. *Creation of a file*. A new file can be created either by a System Call made from a process or by a System Command issued by an interactive user. The file system will respond to the call/command, as follows:-
   - Assign a new FCB to the file to be created
   - Allocate necessary disk-storage-space to the newly created file
   - Load the appropriate directory, under which the new file is to be created, from disk to memory.
   - Update the directory by linking the new FCB to the directory
   - Write back the updated directory onto the disk.
2. *Opening a file for access*.When OS receives a call to Open a File, it would respond as follows:
   - Search the directory structure for the given file name. To speed up this search operation, some parts of directory structure are always cached into the memory.
   - Once the file is found, its FCB is copied into the System-wide 'Open-Files-Table'. The 'Open-File-Count' is incremented by one. Then, an entry is made

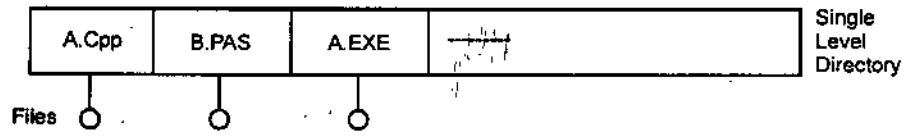
- Make entry in the Per-process 'Open-Files-Table'. This will have a pointer to the corresponding entry in the System-wide 'Open-File-Table'. This table also contains file pointer for accessing the file. Initially, it will point to the Beginning of File.

3. *Read.* To read data from a specified location in a file. In sequential read, the file pointer will be automatically repositioned at the next record, after the read operation is completed.
4. *Write.* To write at the current position of the file pointer. If file pointer is at End Of File (EOF), then the file-size is automatically increased.
5. *Seek.* This repositions the file pointer to the specified location. This is done in direct access mode.
6. *Append.* To append the information at the end of file
7. *Close File.* When a process issues a Close-File call, the OS responds by removing the file entry from the per-process open file table. Also, the 'open-file-count', in the System-Wide 'Open-File-Table', is decremented by one. If this count becomes zero, the file entry is removed from the System-wide 'Open-File-Table'.
8. *Delete File.* When a file is deleted, the blocks allocated to it are returned back to the system. Its FCB is de-linked from the directory and FCB is declared to be free.

DIRECTORY STRUCTURE

Single Level Directory

This is the simplest structure. All the files are contained in the same directory structure.



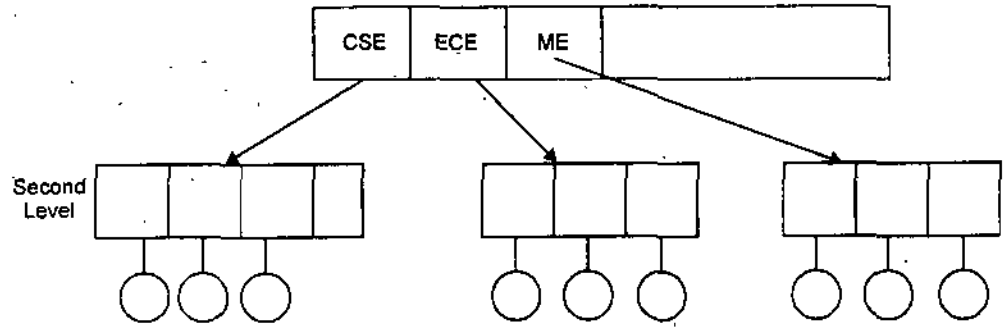
Limitations of Single-Level Directory

1. It becomes highly unwieldy when system has more than one users or the numbers of files is large.
2. Two files by different users, with same name, cannot be accommodated; one of the files will need to be renamed.

NOTES

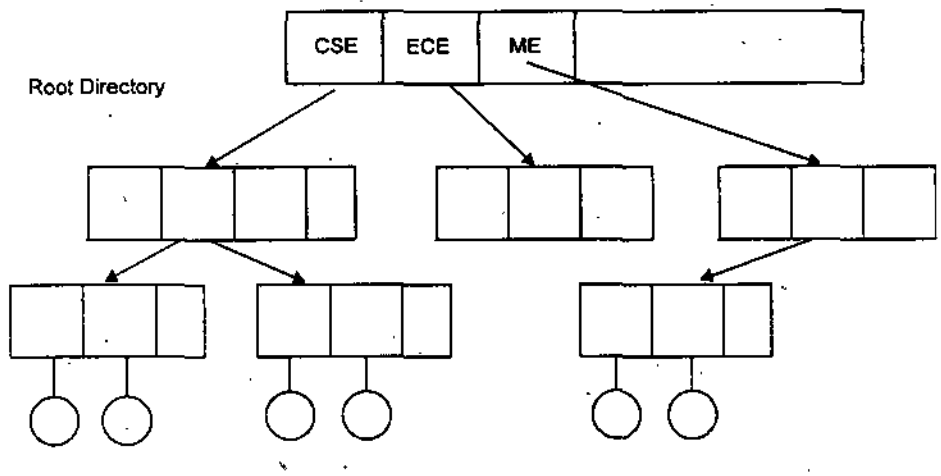
Two Level Directory

This solution can be used to create multiple User File Directories (UFDs), one for each user, under a Master File Directory (MFD). The MFD is indexed by the User Name or User Account Number, to provide an access to the relevant UFD. When a user references any of its files, a search is carried out only in the User File Directory (UFD) of that particular User.



Tree Structured Directory

The two-level directory is in fact a two-level tree. The Tree Structured Directory is a generalization of the Two Level Directory and it forms a tree of an arbitrary height. This permits the Users to create their own sub-directories under the respective UFDs. A directory may contain a set of files and a set of sub-directories. *Each file in the system has a unique access path.* Users may use absolute path name or relative path name (relative to the current directory being accessed by the user) to access sub-directories. New sub-directories can be created and existing sub-directories can be deleted. Some systems insist that a sub-directory can be deleted only if it is empty.



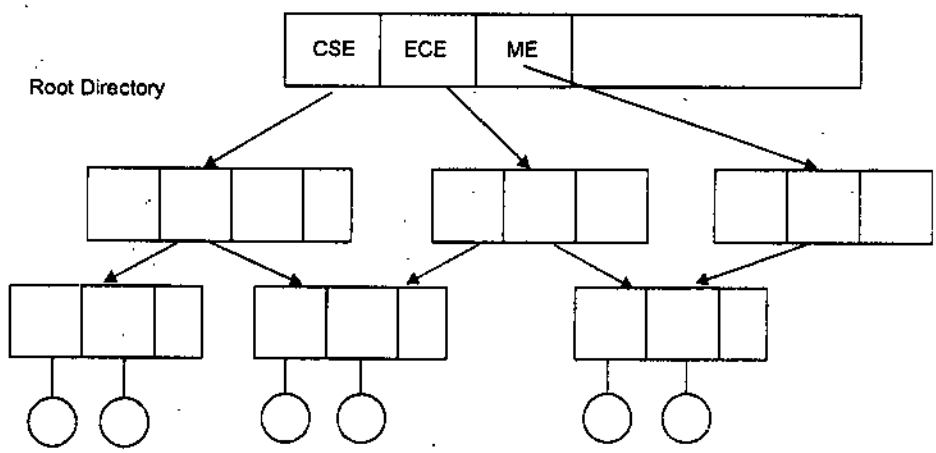
A-cyclic Graph Directory

Suppose multiple users are working on a project, the project files can be stored in a common sub-directory of the multiple users. This type of directory

is called A-cyclic Graph Directory. The common directory will be declared a shared directory. Same way, system may provide shared files. Unlike the Tree Structure, a file may have multiple access paths. However, the graph will contain no cycles. With shared files, changes made by one user are made visible to other users. The major advantage of a-cyclic graph directory is support for shared files and directories. The issues to be resolves are:

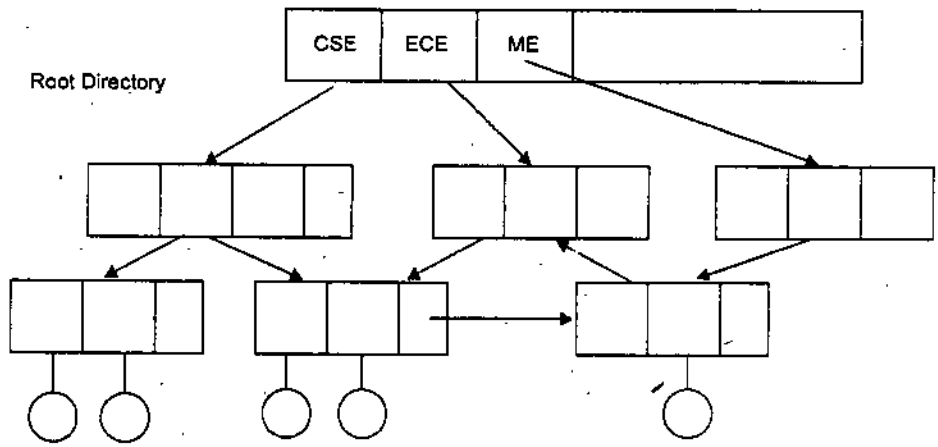
(*a*) A file may now have multiple absolute paths. So, distinct file names may refer to the same file.

(*b*) Whenever, a shared directory/file is deleted, all pointers to the directory/file are to be removed.



General Graph Directory

General Graph Directory permits cycles. One major disadvantage is that a poorly designed search algorithm may get into infinite loop while searching for a file.



DISK SPACE ALLOCATION METHODS

NOTES

Contiguous Allocation

A file is allocated a set contiguous set of blocks.

Advantages

1. The number of disk-seeks to access all blocks of a file is minimal, since the blocks reside on same or neighboring tracks. So, access will be more efficient.

Disadvantages

1. Finding contiguous space of a required size space for a new file will be time consuming.
2. Subsequent extension of a file will be difficult. The whole may have to be relocated elsewhere on the disk.
3. The scheme suffers from external fragmentation. System will have to perform de-fragmentation quite often, to recover the disk space rendered un-usable by external fragmentation.

Dis-Contiguous Space Allocation

The schemes used, for dis-contiguous space allocation on disk, are:

1. Linked Blocks Allocation
2. Indexed Allocation, using:-
   (*a*) Linked Index Blocks
   (*b*) Hierarchical Indexing
   (*c*) I-Nodes

Linked Blocks Allocation

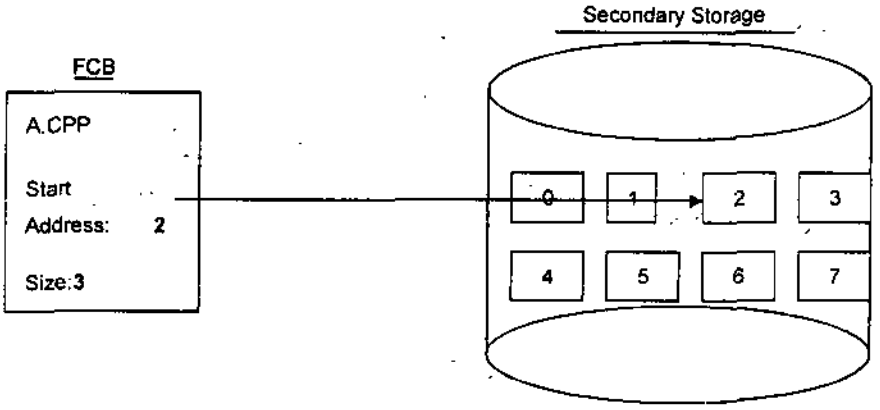
The disk blocks allocated to a file are linked to each other; the address of next block being contained in the previous one. The FCB contains pointer to first block and last block in the link list. The pointer to last blocks is handy at the time of appending new blocks to the list. The disk blocks, allocated to a file, may be scattered all over the disk space.

The FCB indicates the start Address as 2 and size of the file as 3 in the above case. So, the file A.CPP occupies three contiguous blocks i.e. 2, 3 & 4.

STUDENT ACTIVITY

1. What are the most common file organisation schemes?

2. What are the major functions of file system?



Advantages

1. This method does not suffer from external fragmentation; only last block allotted to a file may not be fully occupied (internal fragmentation). So, disk storage space is optimally utilized.

Disadvantages

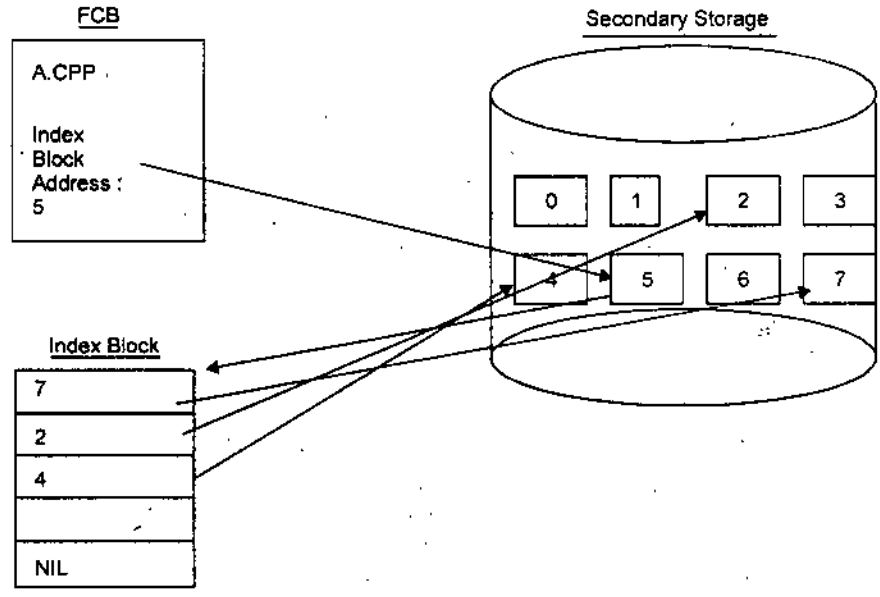
1. Accessing such files is more time consuming, since address of next block needs to be determined from the previous block.
2. Number of disk seeks to access all blocks of a file may be large. Sophisticated disk scheduling will be required to optimize the head movement during seeks.
3. The allocated blocks cannot be accessed randomly.

Indexed Allocation

The method supports random access of the allocated blocks. Each file has its own index block, which is an array of disk block addresses. Indexing can be of the following types:

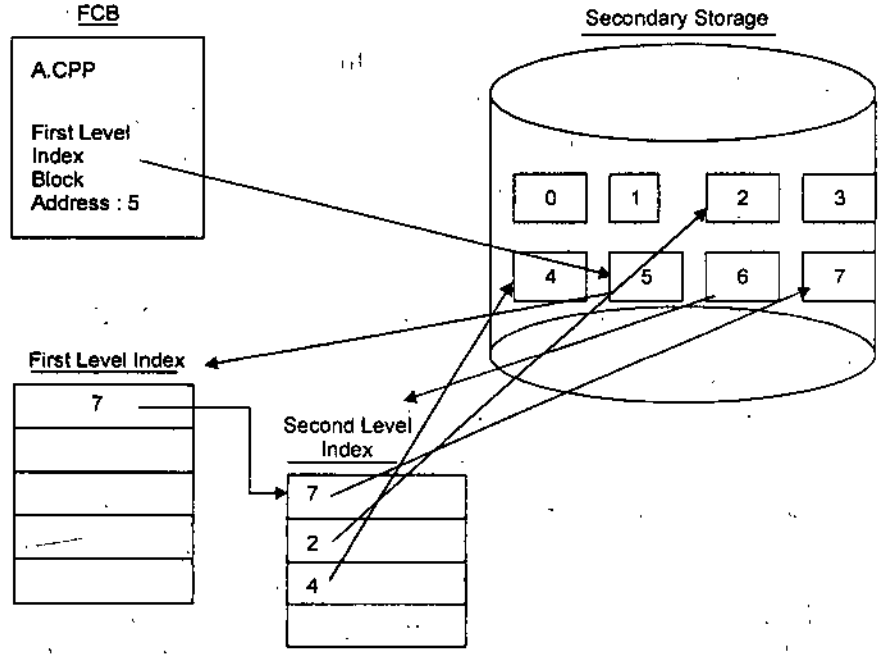
(*a*) *Indexed Allocation, using Linked Index Blocks.* **The FCB contains the Disk Address of the first Index Block.** More than one index blocks may be used by linking together the index blocks. For this, the last location in the Index Block indicates the disk address of next Index Block. This address will be NIL in the last Index Block.

The above diagram indicates that the Index Block of file A.CPP is in disk block number 5. First, the Index Block is loaded and then the file is accessed via the disk addresses contained in the Index File. The addresses are in the same sequence as the logical sequence of file blocks. The above Index Block indicates that the file is stored in blocks no. 7, 2 & 4 in that order. Also, the last address being NIL indicates that there are no more Index Blocks.



(*b*) *Hierarchical indexing.* This method uses multiple level indexing. The FCB contains the disk address of the First Level Index Block. The first level index block contains pointers to the disk blocks containing Second Level Index Blocks. The second level index block point to the disk blocks allocated to the file.

The depiction indicates that the first level Index Block is stored in Disk Block # 5. First this block is loaded. The first entry in this block indicates



NOTES

that one of the second level index blocks is stored in disk block # 6. Then, this second level index block is loaded. This block indicates the file is stored in disk blocks 7, 2 & 4 in that order. Then, these file blocks are accessed.

(c) *Index Nodes (I-Nodes)*. This scheme is used in UNIX. Each file has an I-Node stored on the disk. When a file is opened, its I-Node is loaded from disk to main memory. The I-Node contains file attributes and some addresses of the disk blocks allocated to the file. For small files, addresses of all the allocated disk blocks are accommodated in the I-Node itself. However, for medium and large sized files, it is not possible to accommodate all the disk addresses in the I-Node itself.

The single indirect pointer, when not set to NIL, will point to index blocks, which will contain addresses of file blocks.

The double indirect pointers, when not set to NIL, will contain disk addresses of first level index blocks, which will further contain the disk addresses of second level index blocks and that will further contain the addresses of file blocks.

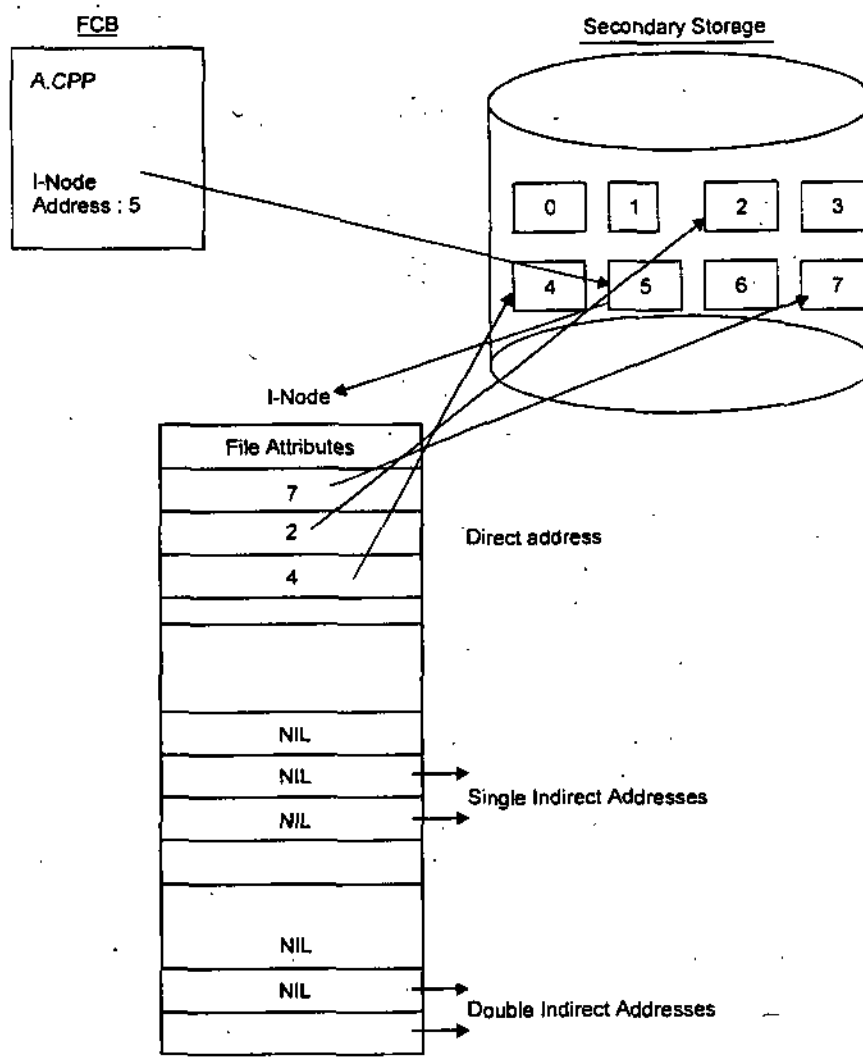
For small files, the direct addresses are sufficient to address the disk blocks allocated to a file. The single indirect pointers and double indirect pointers would be set to NIL.

However, for medium size files, the single indirect pointers are also used along with direct pointers. The access time via single indirect pointers will be larger as compared to direct pointers, due to one level of indirection.

For large files, it will also use double indirect pointers, which have still larger access time.

Advantages of Indexed Allocation

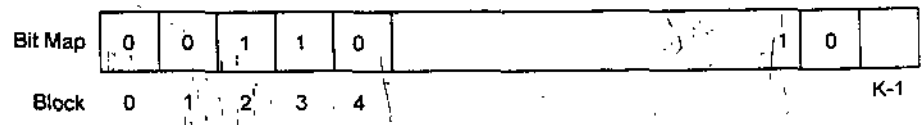
Indexed allocation supports direct access of allocated random blocks, without suffering from external fragmentation.



FREE DISK-SPACE MANAGEMENT

The OS maintains a pool of free disk blocks. Whenever, a new file is created, blocks are allocated from the free blocks' pool. Whenever, a file is deleted, its allocated blocks are declared free and put in the free blocks pool. Free blocks pool may be implemented as:

(a) *Bit Map*. The Bit Map is stored at a known location on the disk. The Bit Map has a bit associated with each block. It can be set to '1', if the block is free; else to '0' if the block is already allocated. For accessing the information, the Bit Map is loaded into memory.



Assume 1: Free

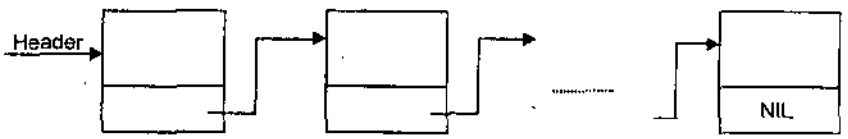
0: Already Allocated

A System can use opposite convention also i.e. 0: Free, 1: Allocated

The advantage is that information about each block can be accommodated in a single bit. It requires special bit manipulation routines to maintain and access the bit map.

NOTES

(*b*) *Free Blocks' Link List.* This method maintains a linked list of all free blocks on the disk. For accessing the information in the linked list, it is loaded into physical memory. Whenever, any allocation is to be done, blocks are removed from the head of the list and allocated. It occupies large space. Also, traversing the link list will be time-consuming.

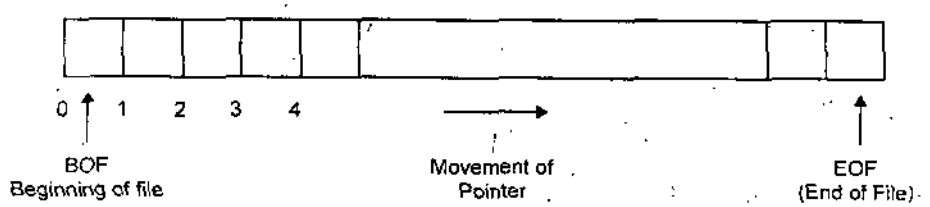


FILE ACCESS METHODS

File access methods depend on the underlying file organization. Commonly used file-access methods are:-

Sequential Access

It is based on a tape-model of file. When a file is opened for 'Read' operation, the File Pointer is positioned at the beginning of file. When a block has been read, the pointer is automatically shifted to the next record. When a file is opened for 'Write' operation, the pointer is positioned at the EOF. A 'Write' operation appends a record to the EOF and advances the pointer to the end of newly appended record, which now is EOF. Some systems may support skipping of some records forward and backward. When File Pointer is pointing to the EOF, it is sensed by the pointer and returned to the calling process.



Advantages 1 Simple implementation

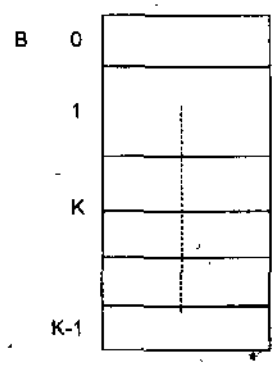
Disadvantages 1 The access, being sequential, is not efficient.

NOTES

2. Random access of a Record is not feasible. Average time to access a particular record will be equal to the time required to access half the file.

Direct Access or Random Access or Relative Access

This access is possible, only if a file is made up of fixed-length logical records. A read/ write request will include a record number. The record numbers are kept sequential. Such files are indexed by record number and support random or direct accessing of records, jumping from one record to another forward or backward. It supports absolute or relative indexing of records. This access method is most suitable for DBMS files.



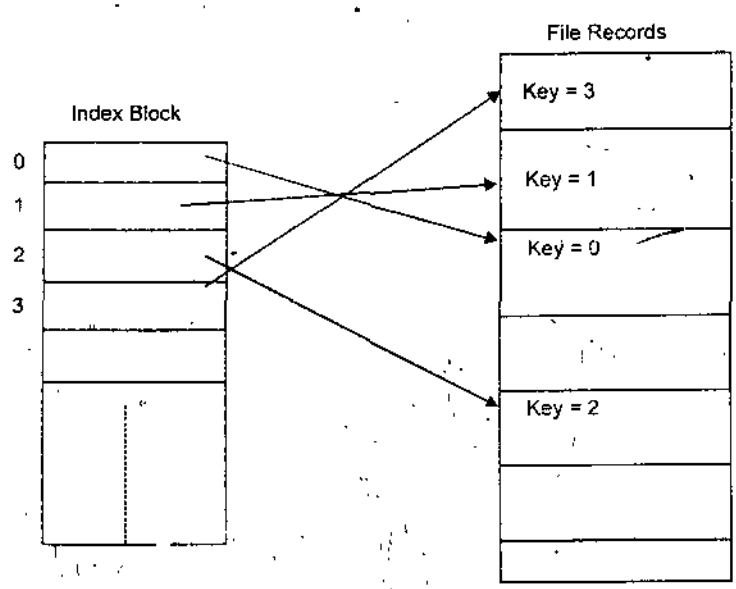
Suppose B = Base Address of a file, with records numbered 0.. N-1

S = Size of each record, the address

Then, Address of Record "K" = B + K * S

Indexed Access

The records are arranged in a logical sequence according to a key contained in each record. The system maintains an index containing the physical address



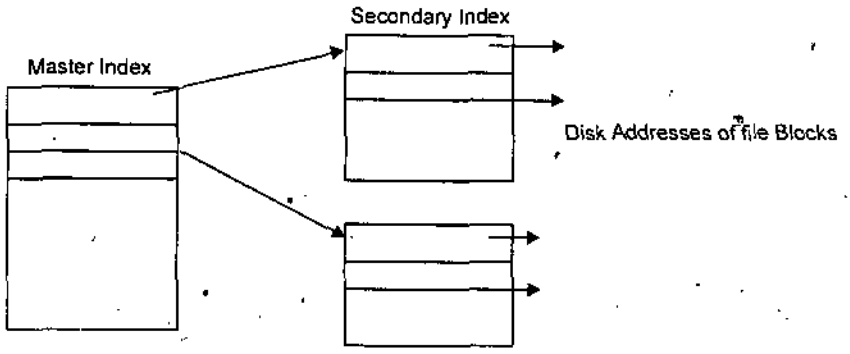
of certain records. By using the key attribute, the records can be indexed directly. This is most commonly used method in DBMS files.

NOTES

Indexed Sequential Access Method (ISAM)

It uses a small master index that points to disk blocks of a secondary index. The secondary disk blocks in turn point to the actual disk blocks allocated to a file. The file is kept sorted on a defined key. To find match for a key, we first make a binary search in the master index table, which provides a block number of the secondary index. The secondary block is read into memory and a binary is made in the secondary index to determine address of the file block containing the key value. Finally, the block is searched sequentially to determine the record containing the key value.

The pointers in the Master Index and Secondary Index Blocks are sorted as per the key values. Each File Block contains a number of records, which are also sorted as per the key value.



The FCB contains the disk address of Master Index Block.

So, while searching for a key value, perform the following steps:-

- First load the Master Index Block
- Carry out binary search for the key value in the Master Index Table and locate the disk address of desired Secondary Index Block.
- Then load the desired Secondary Index Block.
- Carry out binary search for the key value in the Secondary Index Table and get the disk address of desired File Block.
- Then load the desired file block.
- Finally, carry out a sequential search with the key value in the file block to locate the desired record.

NOTES

SHARING OF FILES

Sharing of Files by Multiple Users

The OS implements the concept file/directory **owner (or user)** and **group**. The **owner** is the user who can **change the file/directory attributes, grant access rights** to other users and has the maximum access rights over the file/directory. **Group** is a sub-set of users, who are granted a sub-set of access rights over the file/directory. Other users (other than the owner & group users) may also have a small sub-set of access rights over the file/directory. The access rights may be to **read, list, write, append, delete, execute, change attributes** etc. Whenever, a user makes a call to access a file/directory, OS will check the access rights of that user before granting access.

File Sharing on Remote File Systems

It is common in distributed systems. The OS may support a **Distributed File System**, in which remote directories & files are made visible to the user via the underlying networking. Users may be granted access rights on remote files & directories.

Client-Server Model

It is common in networked systems like world wide web (www). The servers declare the files which are available to the clients. A server serves multiple clients. So, a file at a server may be accessed simultaneously by multiple clients. Clients are identified at the servers by IP Addresses. Additional authentication methods may be used to validate the clients.

FILE PROTECTION MECHANISM

Any real secure system must have some protection mechanism or mechanisms to enforce the access rights required by the system. In the case of Unix, we rely on the memory management unit to enforce access rights for memory segments, and we rely on the file system to enforce the access rights for files.

In this section we will cover the techniques that are used in operating systems to protect file systems.

Protection Domains

A computer system may have objects that need protection. These objects may be hardware (as disk drives, memory segments, printers e.t.c) or software (as processes, semaphores, files, databases etc).

NOTES

Each object has a unique name by which it can be identified and a set of operations that can be carried out on it.

A Protection Domain is a set of (object, rights) pairs. At any given time this pair specifies an object and a set of operations, having rights to perform on it. It is possible for a single object to be in multiple domains at the same time and objects can also switch from one domain to another during execution, which is highly system dependent.

Let us take an example of UNIX. In UNIX , the domain of a process is defined by its uid and gid. For any combination, it is possible to make a complete lists of all objects that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same combination will have access to exactly the same set of objects and with different combination will have access to a different set of objects, although overlapping may be possible in most of the cases.

Access Control Matrices

There are two practical ways to implement the access matrix, *access control lists* and capabilty lists. Each of these allows efficient storage of the information in the access matrix.

Access Control Matrices

	Data 1	*Data 2*	*Prog 1*	*Prog 2*
Alice	RW		E	
Bob	R	RW	RWE	
Carol		R		E
David	RW	R	E	RWE

In a large system the matrix will be enormous in size and mostly sparse.

Access Control List

It is the column of access control matix

Advantage

Easy to determine who can access a given object.

Easy to revoke all access to an object.

Disadvantage

Difficult to know the access right of a given subject.

Difficult to revoke a user's right on all objects.

Used by most mainstream operating systems.

Capability List

It is the row of access control matrix.

A capability can be thought of as a pair (x, r) where x is the name of an object and r is a set of privileges or rights.

Advantage

Easy to know the access right of a given subject.

Easy to revoke a user's access right on all objects.

Disadvantage

Difficult to know who can access a given object.

Difficult to revoke all access right to an object.

A number of capability-based computer systems were developed, but have not proven to be commercially successful.

Protection Models

Protection matrices are not static. They frequently changes as new objects are created, old objects are distroyed and owners decide to increase or restrict the set of users for that object.At any time the matrix determines that what are the access rights of a process in a given domain.

SUMMARY

- The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary storage device is disk. Physical disks may be segmented into partitions to allow multiple file systems per spindle,and these file systems are mounted onto a logical file system to make them available for the further use.
- The various files can be allocated on the disk in three ways: through contiguous, linked or indexed allocation strategies.
- File access methods depend on underlying file organization. The most commonly used file-access methods are: sequential access, Direct access and indexed sequential access methods.

SELF-ASSESSMENT QUESTIONS

1. Explain various operations possible on File.
2. Discuss various File Allocation Strategies for disk space management.

NOTES

3. A sequential access file has fixed size 15 byte records. Assuming the first record is record 1, the first byte of record 5 will be at wht logical location?
4. Which allocation scheme woul work best for a file system implemented on a device that can be accessed sequentially, a tape drive, for instance.
5. Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particlar file?
6. How do caches help to improve performance? Why do systems not use more or larger caches if they are so useful?
7. Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

   86,1470,913,1774,948,1509, 1022, 1750, 130

   Starting from the current head position, what is the total distance(in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk- scheduling algorithms?

   (*a*) FCFS (*b*) SSTF

   (*c*) SCAN (*d*) C-SCAN

   (*e*) C-LOOK
8. Consider a file currently consisting of 100 blocks. Assume that FCB (and index block in the case of indexed allocation) is already in the memory. Calculate how many disk I/O operations are required for contiguous, linked and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous allocation, assume that there is no room to grow in the beginning, but room to grow at the end. Assume that the block information to be added is stored in the memory.

   (*a*) The block is to be added at the beginning.

   (*b*) The block is to be added in the middle.

   (*c*) The block is to be added at the end.

   (*d*) The block is to be removed from the beginning.

   (*e*) The block is to be removed from the middle.

   (*f*) The block is to be removed from the end.
9. Consider a file system on a disk that has both logical & physical block sizes of 512 bytes. Assume that the information about each file is

(c) *Process Control Block (PCB).* The PCB refers to a data structure, created in the system to maintain the complete information about a process. The PCB is kept constantly updated, to reflect the current status of the related process. The intended contents of PCB vary from Operating System to another. But, some of the fields are common:-

(*i*) Process Id number (System Assigned. At any instant, this has to be unique for a process. This is analogous to primary key of a process)

(*ii*) Process Name (User Assigned)

(*iii*) Saved Program Counter (PC) value *i.e.*, address of the next instruction to be executed, when the process goes to run state. Initially, this points to the first instruction of the code. When a process goes to run sate, this value is loaded to a hardware register called PC, for faster access during process execution. When the current instruction has been executed, the PC is automatically updated to point to the next instruction. When a process goes to wait state, or is pre-empted, the PC value is saved in the PCB, which will be reloaded when the process comes back to run state.

(*iv*) Saved Stack Pointer (SP). Initially, it points to the bottom of the stack. When a process goes to run sate, this value is loaded to a hardware register called SP, for faster access during process execution. When a process goes to wait state, or is pre-empted, the SP value is saved in the PCB, which will be reloaded when the process comes back to run state.

(*v*) Saved Processor Status Word (PSW)- This indicates the value of hardware status to be loaded in the Processor Status Register (PSR), when the process goes to run state. It contains information like Condition Codes, Process Priority, Operation Mode (User or Supervisory or Kernel) etc. When a process goes to wait state, or is pre-empted, the PSR value is saved in the PCB, which will be reloaded when the process comes back to run state.

(*vi*) Saved contents of other general-purpose-registers, which are to be loaded into the general-purpose Registers when the process goes to run state.

(*vii*) Current Process State, like Running, Waiting, Ready to Run, Suspended, Swapped-Out etc.

(*viii*) Necessary memory-management information.

To accomplish its task, a process needs certain resources like CPU time, memory, files & I/O devices. These resources are allocated to the process either at the time of its creation of when it is executing. In a multi-

programming environment, there will a number of simultaneous processes existing in the system; some of these will be system processes and others will be user processes.

NOTES

PROCESS CONTROL BLOCK

A process is an instance of a program in execution. PCB is a data structure to maintain complete information of a process. The current data in a PCB will indicate current status of related process (also called 'task'). The extent of this information varies from OS to OS. However, some important status information, as stored in a typical PCB, will be:

Process-Id (Unique)
Process Name
Executable File Name
Saved Registers
Program Counter (PC)
Stack Pointer (SP)
Processor Status Word (PSW)
Memory Management Information
Status of Open Files
Status of Assigned Devices
Accounting Information
Pointer to Previous PCB
Pointer to Next PCB

The memory management information may comprise information about Memory Limits, Page Tables, Segment Tables etc, depending on the type of memory management used. The accounting information may comprise the consumption of execution time, memory space, disk space etc. The pointers to Previous PCB and to Next PCB are meant for linking the PCB in a doubly-linked-queue.

Process States

Typical process states in an OS are:

New

A Process just created. It is entered into the Ready Queue.

Ready

A Process that is ready to run. It is waiting in the Ready Queue for dispatching *i.e.*, to take control of the CPU. There can be many Ready Processes at a time.

Running

A process from Ready Queue is *dispatched* i.e. assigned control of the CPU. This Process is called in Running State i.e. executing its instructions. In a Von-Neumann-Architecture, only one process can be in this state at a time.

Waiting/ Blocked/ Suspended

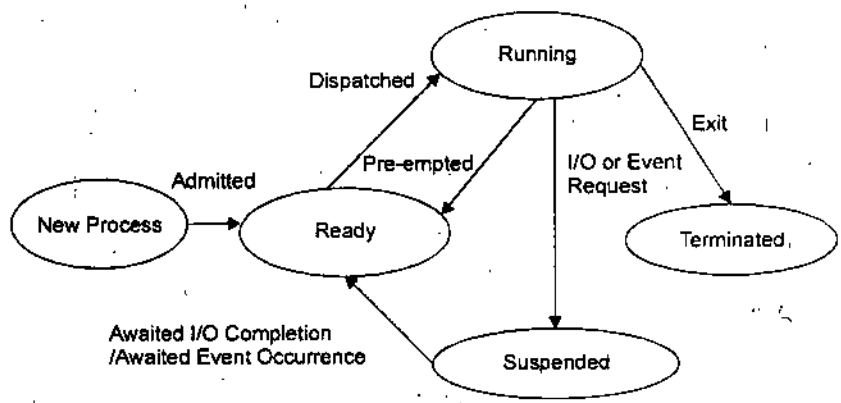
A Process waiting for completion of I/O or waiting for an event to occur. There can be many suspended processes at a time.

Terminated

A process, that has been deleted, due to completion of its execution or on account of causing a fatal exception.

PROCESS STATES TRANSITION

A process, during its execution cycle, undergoes various state transitions from a New Process at the time of creation to a Terminated Process after its deletion. A typical State Transition diagram of a Process is depicted below:-



The names of various states may vary from OS to OS.

JOB/PROCESS SCHEDULING

Scheduling refers to the set of policies and mechanisms that an OS supports for determining the order of execution of pending Jobs and Processes. A

Scheduler is an OS Module, that determines the next pending job to be admitted into the system for execution or next ready process to be dispatched to RUN state.

NOTES

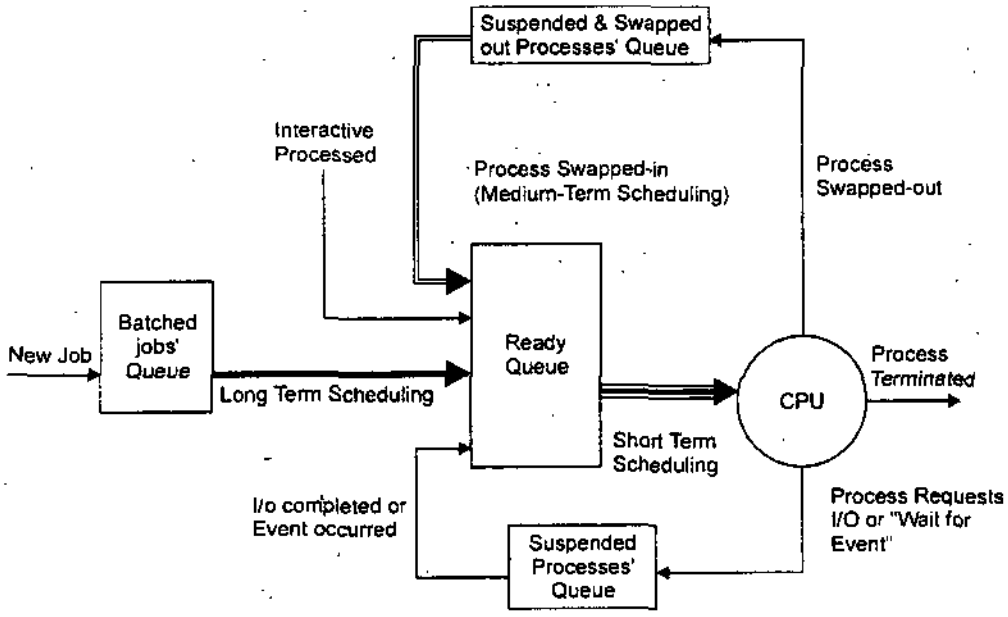
Goals of Scheduling

(*a*) To optimize the utilization of system resources.

(*b*) To provide as fair deal as possible to all pending jobs & processes.

(*c*) To ensure that more critical processes get priority over others.

TYPES OF SCHEDULING



Long Term Scheduling

Long term scheduling refers to the OS decision of admitting (loading into memory and taking up for execution) the pending Batch Jobs from the Batch Queue. Batch Queue is normally reserved for resource-intensive and low-priority jobs that could act as fillers when interactive activity is low. Each Batch job contains programmer-specified or system-assigned parameters of resource needs, in terms of expected execution time, memory and I/O requirements. These specifications enable OS to decide their order of execution. **The primary goal of long-term-scheduling is to achieve a proper mix of IO-bound and CPU-bound jobs, so as to keep the CPU and I/O devices as occupied as possible.**

Medium Term Scheduling

When a process is suspended after making an I/O request, it cannot make any progress till the requested I/O is completed by the OS. Sometimes, a suspended process is removed from the main memory and shifted to secondary-storage (swapped-out), to make space for another process. Medium Term Scheduler is charged with the handling of swapped-out processes. It determines when a swapped-out process can be moved back into the memory (swapping-in), and resumed from the same point, where it was swapped-out.

Short Term Scheduling

It allocates the CPU amongst the pool of Ready Processes, resident in the memory. Assigning the CPU to a process is also called 'Process Dispatching'.

The Short Term Scheduler is the most frequently invoked OS module. It will be invoked every time, when any of the following events occurs:

(*i*) When the running process requests an I/O or makes a System Call. The running process is suspended i.e. transferred from Running State to Suspended State (also called Waiting State or Blocked State).

(*ii*) The running process completes its execution. The running process is terminated.

(*iii*) A new process is created. The new process is entered into the Ready-Queue. In the case of priority-based-preemptive scheduling, a higher-priority process getting ready will preempt a lower-priority running process. The preemption means that the currently running process will be transferred to the Ready Queue in a partially executed state and the higher priority process will be transferred to Running State. The preempted process will have to wait for its turn (as per its priority) to complete its pending execution.

(*iv*) A Fatal Exception occurs, like 'Divide-by-Zero'. The running process is forced to Exit. If the exception is too severe, like 'Memory Access Violation in Kernel Mode', then the OS may force the system to be rebooted.

(*v*) An Interrupt occurs, signaling 'Completion of an I/O' or 'Completion of a System Call'. The process, that had requested the I/O or System Call, will be brought out from Suspended State and transferred to Ready State. If the priority of this process happens to be higher than the currently-running-process, in a priority-based-preemptive scheduling, this process will preempt the currently running process.

(*vi*) Occurrence of Timer Interrupt. The Timer Interrupt may signal 'Expiration of assigned Time Slice' of the running Process. In that eventuality, the running process will be transferred from Running State to Ready State.

NOTES

A running process could be suspended (by issue of I/O request or System Call) and thus transferred to Suspended Queue; OR could be pre-empted (by a higher priority process getting READY or expiration of its time-slice or by receipt of an INTERRUPT) and thus transferred to READY Queue, OR could be terminated (due to completion or on account of causing a fatal exception). Under all these situations, a running process will relinquish control of the CPU, causing 'Context Switching'.

CONTEXT SWITCHING

Context of a process is saved in its PCB. Switching of CPU from one process to another requires saving of the state of the old process in its PCB and loading of the saved state of the new process. This task is performed by OS and is known as Context Switching. The context switch involves the following steps:-

(*i*) Saving the 'Context' ('State') of the Running Process in its Process Control Block (PCB). In case, the Running Process is being terminated, 'Context Saving' will not be necessary. In that eventuality, the process will be deleted, causing deletion of its PCB and freeing the memory occupied by the running process.

(*ii*) Shifting of the PCB of the running process to newly assigned Queue. In case of suspension of the running process, its PCB will be shifted to the 'Blocked or Suspended Processes Queue'. In case pre-emption of running process, its PCB will be shifted to the 'Ready Queue'.

(*iii*) Selection of a process from the 'Ready Queue' (provided a process exists in the Ready Queue) to take control of the CPU (called Process Dispatching). The Context of the newly selected process will be loaded into the system registers. This is called 'Context Un-saving' or 'Context Loading'. The new process will be enabled to Run by loading the address of its next instruction in the Program Counter (PC).

The 'Context Switching' is a pure overhead. The extent of this overhead depends on the size of the 'Process-Context' *i.e.*, the extent of process-state saved in the PCB. Larger the Process-Context, higher will be the Context-Switching-Overhead. The goal is to keep this overhead as low as possible. One of the techniques to reduce this overhead is use of multiple sets of hardware registers. The context of an active process can be kept in a separate set of registers. Context can be switched by switching from one set of registers to other set of registers, without necessity of saving and un-saving of process context.

NOTES

How to evaluate a Scheduler?

A Scheduler algorithm is evaluated against some widely accepted performance criteria, as discussed as follows:

PERFORMANCE CRITERIA OF A SCHEDULER

(*a*) CPU Utilization

(*b*) Throughput of the System

(*c*) Average Turnaround Time of a Process

(*d*) Average Waiting Time of a Process

(*e*) Response Time.

The above criteria are defined below:-

(*a*) *CPU Utilization.* It is the average fraction of time, during which CPU is busy, executing either user programs or system modules. Higher the CPU utilization, better it is.

(*b*) *System Throughput.* It is the average amount of work completed per unit time. One way of expressing throughput is by the average number of user jobs completed per unit time. Higher the throughput, better it is.

(*c*) *Turnaround Time of a Job / Process.* Turnaround time of a process or a job is the total time elapsed from the time the job is submitted (or process is created) to the time the job (or process) is completed.

Turnaround Time = Process/Job Finish Time - Process/Job Arrival Time

Lower the average Turnaround Time, better it is.

(*d*) Waiting Time of a Job/Process. Waiting Time of a job (or process) is defined as the total time spent by the job (or process) while waiting in Suspended State or Ready State, in a multiprogramming environment.

Waiting Time = Turnaround Time - Actual Execution Time

Lower the average waiting time, better it is.

(*e*) *Response Time.* This parameter has relevance for interactive time-sharing systems and for real-time systems. In interactive systems, response time is defined as the time elapsed from the moment last character of a command-line is typed-in by an interactive user, to the time when first response to that command appears on the terminal. For real-time systems, it is defined as the time elapsed from the time an event is reported in the system, to the time when first instruction of its Interrupt Servicing Routine (ISR) is executed to service the interrupt caused by the event.

CPU-Bound Jobs

The jobs, that spend comparatively more time performing number-crunching operations (*i.e.*, CPU operations) and comparatively less time performing I/O operations are called CPU-bound jobs.

NOTES

I/O Bound Jobs

The jobs that spend comparatively more time performing I/O operations and comparatively less time doing CPU operations are called I/O bound jobs.

> The Long-Term-Scheduler, while loading jobs for execution, ensures a proper mix of CPU-bound Jobs and I/O-bound jobs so as to optimize the utilization of both CPU and I/O devices.

CPU Bursts & I/O Bursts

A process, during execution, will keep on alternating between CPU-Bursts and I/O bursts. Normally, the CPU bursts will comparatively be short (of the order of 4 ms or so) & I/O bursts will be comparatively longer.

Prediction of next CPU Burst

There are some algorithms like 'Shortest Remaining Time Next', which operate based on the predicted value of the next CPU burst. The value of the next CPU burst can be predicted by using the Exponential Average of the previous measured values of CPU bursts. This method works on the premise that next CPU burst of a process will be similar to the previous ones.

Let τ_n be the predicted value of n^{th} CPU burst

t_n be the measured value of n^{th} CPU burst

Then, the predicted value of the next CPU bursts

$$\tau_{n+1} = a\, t_n + (1-a)\, \tau_n$$

$$= a\, t_n + a\,(1-a)\, t_{n-1} + a\,(1-a)^2\, t_{n-2} + \ldots + a\,(1-a)^n\, t_0 + (1-a)^{n+1}\, \tau_n$$

where $0 < a \leq 1$

t_n is the just previous measured value and the value of t_n represents the previous history of measured values. If the value a is closer to 1, it implies more weightage is assigned to the nearest history. Else, if the value of a is closer to 0, it implies that more weightage is assigned to distant history.

VARIOUS SCHEDULING ALGORITHMS

1. First Come First Served (FCFS)
2. Shortest Job First (SJF)
3. Shortest Remaining Time Next (SRTN)
4. Priority Based Non-Preemptive 'Scheduling
5. Priority Based Preemptive Scheduling
6. Round Robin Scheduling
7. Multi Level Queues (MLQ) Scheduling
8. Multi Level Queues (MLQ) Scheduling with feedback

STUDENT ACTIVITY

1. What is the concept of a process and PCB?

2. What is CPU scheduler?

NOTES

FCFS Scheduling

The jobs or processes are dispatched (scheduled to run) strictly in the same order, as those have arrived in the system. This algorithm is fairest, in the sense that a job/process gets executed strictly in the same sequence, as it has arrived in the system.

Example :

Process	*Arrival Time ms*	*Next Burst Ms*
P_0	0	10
P_1	1	6
P_2	3	2
P_3	5	4

FCFS Gantt Chart:

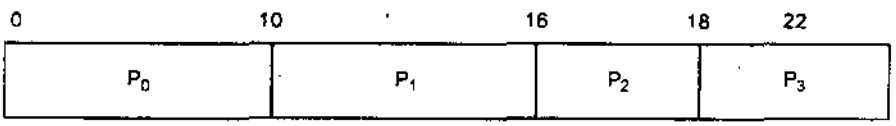


Table depicting performance of FCFS Algorithm:

Process	*Arrival Time* (T_0) *ms*	*Next Burst* (Δt) *ms*	*Finish Time* (T_1) *ms*	*Turnaround Time* $TAT = T_1 - T_0$	*Waiting Time* $WT = TAT - \Delta t$
P_0	0	10	10	10	0
P_1	1	6	16	15	9
P_2	3	2	18	15	13
P_3	5	4	22	17	13

Average Turnaround time = (10 + 15 + 15 + 17) / 4 = 57/ 4 = 14.25 ms

Average Waiting Time = (0 + 9 + 13 + 13) / 4 = 35/4 = 8.75 ms

Shortest Job First (SJF) Scheduling

The next job to be dispatched will be the one, which happens to be the shortest amongst the pending lot of jobs. This algorithm is non-preemptive; so a job, once scheduled, is permitted to complete its next burst.

Let us see the performance of this algorithm against the above set of processes.

Process	Arrival Time ms	Next Burst Ms
P_0	0	10
P_1	1	6
P_2	3	2
P_3	5	4

SJF Gantt Chart

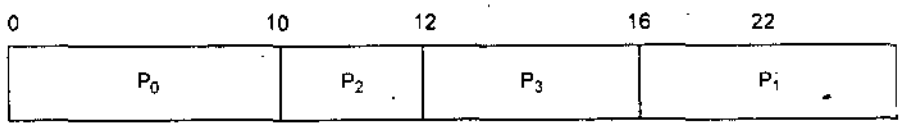


Table depicting performance of SJF Algorithm:

Process	Arrival Time (T_0) ms	Next Burst (Δt) ms	Finish Time (T_1) ms	Turnaround Time $TAT = T_1 - T_0$	Waiting Time $WT = TAT - \Delta t$
P_0	0	10	10	10	0
P_1	1	6	22	21	15
P_2	3	2	12	09	07
P_3	5	4	16	11	07

Average Turnaround time = (10 + 21 + 09 + 11) / 4 = 51/ 4 = 12.75 ms

Average Waiting Time = (0 + 15 + 07 + 07) / 4 = 29/4 = 7.25 ms

SJF produces Average Turnaround Time and Average Waiting Time better than the FCFS algorithm.

Shortest Remaining Time Next Algorithm

This is a preemptive algorithm, wherein the next job/process to be dispatched will be the one that happens to be shortest amongst the pending jobs, at the time of making the decision. However, if a process arrives later, whose next burst happens to be shorter than the remaining burst time of the currently running process, the new process will preempt the currently running process. The preempted process will be later re-dispatched, when its remaining burst happens to be shortest amongst the pending processes.

Let us see the performance of this algorithm against the set of processes considered in the above example.

NOTES

Process	Arrival Time ms	Next Burst Ms
P_0	0	10
P_1	1	6
P_2	3	2
P_3	5	4

SRTN Gantt Chart

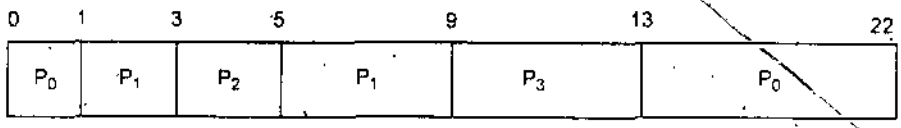


Table depicting performance of SRTN Algorithm:

Process	Arrival Time (T_0) ms	Next Burst (Δt) ms	Finish Time (T_1) ms	Turnaround Time $TAT = T_1 - T_0$	Waiting Time $WT = TAT - \Delta t$
P_0	0	10	22	22	12
P_1	1	6	09	08	02
P_2	3	2	05	02	00
P_3	5	4	13	08	04

Average Turnaround time = (22 + 08 + 02 + 08) / 4 = 40/ 4 = 10 ms

Average Waiting Time = (12 + 02 + 00 + 04) / 4 = 18/4 = 4.5 ms

In terms of average Turnaround Time and Average Waiting Time, SRTN is even better than SJF.

But, SRTN involves more number of context-switching events, since it is a preemptive algorithm. The additional events of context switching will cause some additional overheads. So, the actual average Turnaround Time and waiting time will be slightly higher than as indicated above.

One major limitation of SJF and SRTN is likelihood of starvation of a long job. Theoretically, there exists a distinct probability that a long job may get unduly delayed (or in extreme cases, may get even starved), if relatively shorter jobs keep arriving one after another.

Priority-based Non-preemptive Algorithm

More critical processes are assigned a priority higher than the less critical ones. At the time of scheduling, a process dispatched will be the one that has highest priority amongst the processes waiting in the Ready Queue. Once

NOTES

dispatched, a process P_1 is allowed to complete its next burst, even if another process of higher priority becomes ready to run during the execution of P_1.

Let us assume the following set of processes:

Process	*Arrival Time ms*	*Next Burst Ms*	*Process Priority*
P_0	0	10	5
P_1	1	6	4
P_2	3	2	2
P_3	5	4	0

Suppose, lower the priority value, higher is the process priority. i.e. Priority of process with priority-value 0 will be higher than the priority of process with priority-value 1.

So, Priority of the above processes is: $P_3 > P_1 > P_2 > P_0$

Gantt Chart (Priority-based non-preemptive algorithm):

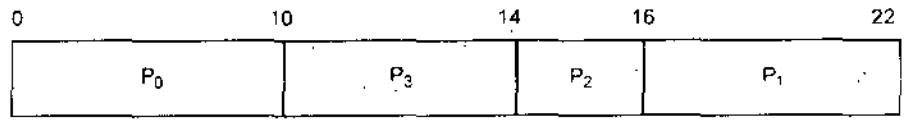


Table depicting performance of Priority based Non-preemptive Algorithm:

Process	*Arrival Time (T_0) ms*	*Next Burst (Δt) ms*	*Finish Time (T_1) ms*	*Turnaround Time $TAT = T_1 - T_0$*	*Waiting Time $WT = TAT - \Delta t$*
P_0	0	10	10	10	00
P_1	1	6	22	21	15
P_2	3	2	16	13	11
P_3	5	4	14	09	05

Average Turnaround time = (10 + 21 + 13 + 09) / 4 = 53/ 4 = 13.25 ms

Average Waiting Time = (00 + 15 + 11 + 05) / 4 = 31/4 = 7.75 ms

The Average Turnaround Time & Average Waiting Time, achieved with this algorithm, is much worse than SRTN. But, these parameters are not of much relevance here, since the main focus is to service the more critical processes earlier than the less critical ones.

NOTES

Priority-based Preemptive Algorithm

At the time of scheduling, a process dispatched will be the one that has highest priority amongst the processes waiting in the Ready Queue. When a process P_I is executing, if another process P_J of higher priority becomes ready to run during its execution, then P_i will be preempted by P_J.

Let us assume the set of processes, as considered in the last example:-

Process	*Arrival Time ms*	*Next Burst Ms*	*Process Priority*
P_0	0	10	5
P_0	0	10	5
P_1	1	6	4
P_2	3	2	2
P_3	5	4	0

Priority of the above processes is: $P_3 > P_2 > P_1 > P_0$

Gantt Chart (Priority-based Preemptive algorithm)

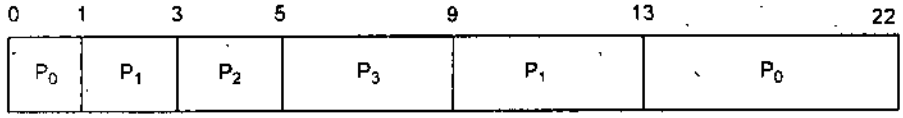


Table depicting performance of Priority based Preemptive Algorithm :

Process	*Arrival Time (T_0) ms*	*Next Burst (Δt) ms*	*Finish Time (T_1) ms*	*Turnaround Time TAT = $T_1 - T_0$*	*Waiting Time WT = TAT – Δt*
P_0	0	10	22	22	12
P_1	1	6	13	12	06
P_2	3	2	05	02	00
P_3	5	4	09	04	00

Average Turnaround time = (22 + 12 + 02 + 04) / 4 = 40/ 4 = 10 ms

Average Waiting Time = (12 + 06 + 00 + 00) / 4 = 18/4 = 4.5 ms

In this algorithm, higher the priority of a process, lower will be its waiting period.

Here, the Process priority will be in the order: $P_3 > P_2 > P_1 > P_0$

And Process waiting time is in the order : $P_3 = P_2 < P_1 < P_0$

Limitation of Priority based algorithms

The priority-based algorithms suffer from the inherent possibility of starvation of low-priority processes. One remedy lies in enhancing the priority of a process in proportion to the time spent by the process in Ready Queue. This is known as priority adjustment, in relation to the age of a process. This would exclude the probability of starvation of a low-priority process, since as it ages in the Ready Queue, its priority will grow, finally enabling it to be dispatched.

Average Context Switching overhead of a preemptive algorithm will be higher than the average context switching time of the corresponding non-preemptive algorithm. Thus, the throughput of a preemptive algorithm will be lower than its non-preemptive counterpart.

Round Robin Scheduling

This scheduling algorithm is specially tailored for Interactive Time-Sharing Systems. A small unit of time, called Time-Slice or Time-Quantum is defined. The time slice normally varies from 10 to 100 ms. The Ready Queue is implemented as a FCFS Queue. A new process is linked to the tail of Ready Queue. The process at the head of Ready Queue is dispatched. The process is assigned an execution time, equal to the specified Time-Slice. Now, there are three possibilities:-

(*a*) The dispatched process completes its execution within the assigned time slice and terminates.

(*b*) The dispatched process requests I/O within the assigned time-slice and is blocked on the Suspended Queue.

(*c*) The process is not able to finish its next burst within the assigned time-slice. At the expiry of time-slice, a timer interrupt occurs and the process is preempted. It is put at the tail of the Ready Queue.

Under all the above conditions, the next process from the head of the Queue is dispatched for the Time-Slice.

A preempted process, when it reaches the head of the queue, it will again be dispatched for the Time-Slice. This continues till a process completes its execution and terminates.

Let us consider the same set of processes as considered for FCFS, SJF and SRTN algorithms:

Let Time Slice = 4 ms

Context Switching Time = 1 ms

NOTES

Process	Arrival Time ms	Next Burst Ms
P_0	0	10
P_1	1	6
P_2	3	2
P_3	5	4

Gantt Chart for RR Algorithm:

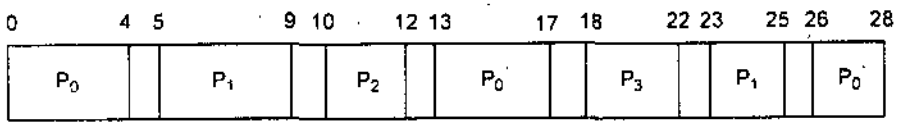


Table depicting performance of SJF Algorithm:

Process	Arrival Time (T_0) ms	Next Burst (Δt) ms	Finish Time (T_1) ms	Turnaround Time $TAT = T_1 - T_0$	Waiting Time $WT = TAT - \Delta t$
P_0	0	10	28	28	18
P_1	1	6	25	24	18
P_2	3	2	12	09	07
P_3	5	4	22	17	13

Average Turnaround time = (28 + 24 + 09 + 17) / 4 = 78/ 4 = 19.5 ms

Average Waiting Time = (18 + 18 + 07 + 13) / 4 = 56/4 = 14 ms

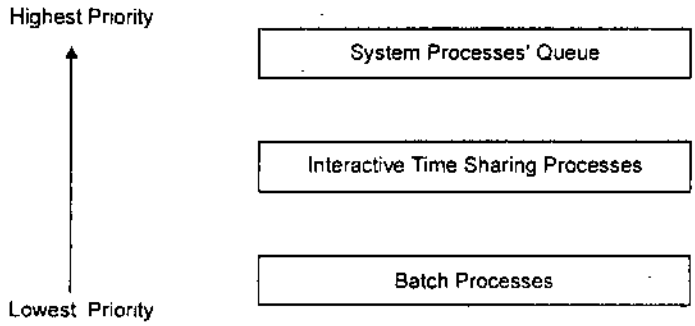
In Round Robin algorithm, the average Turnaround Time and the average Waiting Time are very large.

If time slice is chosen to be longer than the longest possible process, then Round Robin algorithm will become same as FCFS algorithm.

If the Time-slice is chosen to be very small (closer to the context switching period) then context switching overheads will be very high, thus affecting the system throughput adversely. So Time-slice has to be carefully chosen. It should be small enough to give a good response to the interactive users. At the same time, it should be large enough to keep the context-switching overheads low.

Multi-Level Queue Scheduling

The Ready Processes are partitioned into a number of categories like System Processes, Inter-active Time Sharing Process & Batch Processes etc. Also, the Ready Queue is partitioned into the same number of sub-queues, with each sub-queue accommodating a category of processes. There is an Inter-Queue priority as shown below:

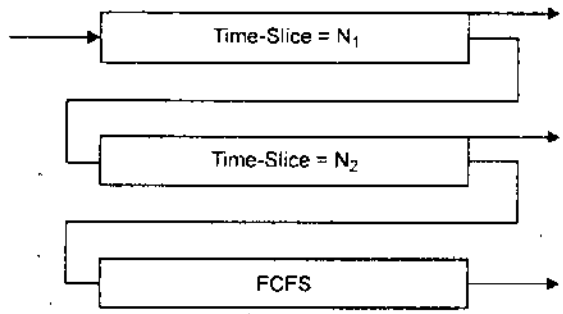


First processes will be scheduled from the highest priority queue *i.e.*, System Processes Queue. When there is no process in the System Processes Queue then processes will be taken up from the next priority queue *i.e.*, Interactive Processes Queue & so on. Each queue can have a different scheduling algorithm; for example System Processes Queue can have Priority-based Preemptive Scheduling, Interactive Processes can have Round-Robin Algorithm & Batch Processes Queue can have Shortest Job First Algorithm.

Advantage of this arrangement is a great degree of flexibility, catering to the specific needs of different types of processes.

Multi-Level Queue (MLQ) with Feedback

The disadvantage of simple MLQ is that a certain type of process is assigned to a particular queue only. The processes in a lower priority queue may face starvation.



In MLQ with Feedback, processes can be moved from one queue to another. For example, in the above arrangement, there are 3 queues. When a process becomes Ready-to-Run, it enters at the tail of the top queue. When a process reaches the front of this queue, it is scheduled with a time-slice of N_1 (say 5 m sec). If it is completed within the assigned time-slice of 5 m sec then it is terminated; else it enters at the tail of next queue. When it comes to the front of the queue and there is no process waiting in the top queue, then it is scheduled with a time slice N_2 (say 10 m sec). If it is completed within 10 m sec, then it is terminated else it enters at the tail of the next queue *i.e.*, FCFS Queue. Now, when it comes to the from of FCFS queue and there is no process

waiting in the top two queue then this process is scheduled and it runs to completion. Obviously, in this arrangement, shorter processes get higher priority. System processes and interactive processes are normally smaller than batch processes.

NOTES

SUMMARY

- CPU scheduling refers to the set of tasks that an OS supports for selecting a waiting process from the ready queue and allocating the CPU to it. Many scheduling algorithms are known for this purpose as first come first serve, shortest job first, round robin, priority scheduling, multilevel queues and multilevel feedback queue scheduling.
- FCFS scheduling is the simplest one but it may cause short processes to wait for very long processes. SJF scheduling provides the shortest average waiting time but the implementation of SJF is difficult because predicting the length of next CPU burst is difficult. Round robin scheduling is appropriate in time-shared systems. It allocates the CPU to the first process in the ready queue for a small unit of time, called a time quantum. After this time quantum if the process has not relinquished the CPU, it is preempted and the process is placed at the tail of the ready queue. In the case of RR scheduling, the problem is selection of time quantum. If the quantum is too large then RR policy is same as FCFS policy. If the quantum is too small, scheduling overhead in the form of context switch time becomes excessive.
- Multilevel queue scheduling allows different algorithms to be used for different classes of processes. Multilevel feedback queue scheduling allows process to move from one queue to another. The FCFS algorithm is nonpreemptive, RR algorithm is preemptive, priority and SJF algorithm may be either preemptive or nonpreemptive.

SELF-ASSESSMENT QUESTIONS

1. With reference to the following set of processes, determine Average Waiting Time & Average Turnaround Time, using the following scheduling algorithms:
   (*a*) First Come First Served
   (*b*) Shortest Job First
   (*c*) Shortest Remaining Time Next (SRTN)
   (*d*) Priority based (Non preemptive)

NOTES

(e) Priority based (Pre emptive)

(f) Round Robin

Note 1. Make use of Gantt Charts.

2. Lower number means higher priority *i.e.*, process with priority 1 has higher priority than process with priority 2.
3. In case of a tie, use FCFS to break the tie.
4. Time slice for Round Robin scheduling is 4 ms

(i)

Process	*Arrival Time Time (ms)*	*Next CPU Burst (ms)*	*Priority*
P1	0	10	3
P2	0	1	1
P3	0	2	4
P4	0	5	2

(ii)

Process	*Arrival Time Time (ms)*	*Next CPU Burst (ms)*	*Priority*
P1	0	24	5
P2	3	7	3
P3	5	6	2
P4	10	10	1

(iii)

Process	*Arrival Time Time (ms)*	*Next CPU Burst (ms)*	*Priority*
P1	0	14	7
P2	1	7	1
P3	3	2	3
P4	5	8	2

5. Suppose a system is using SJF algorithm for CPU scheduling and it predicts next CPU burst from the exponential average of the previous CPU burst. If the first prediction of CPU burst T_0 = 20 ms, weight factor a = 0.6, and the previous CPU burst are 08, 16, 24, 16 ms in that sequence. Determine the next predicted CPU burst.
6. Suppose SJF Queue is implemented as doubly linked circular Queue. Explain advantages of this implementation over a singly linked Queue implementation.

NOTES

7. Explain the degree to which the following algorithms discriminate in favour of short processes:-

   FCFS

   RR

   Multi level Queue with feedback

8. Compare the following CPU Scheduling algorithms, highlighting the strengths and limitations of each algorithm:-

   (*a*) FCFS

   (*b*) SJF (non pre-emptive)

   (*c*) SRTN or SFF (Preemptive)

   (*d*) Priority (Non Pre-emptive)

   (*e*) Priority (Preemptive)

   (*f*) Multi level Queue

   (*g*) Multi level queue with feedback

9. What is common between:-

   (*a*) Priority Scheduling & SJF Scheduling

   (*b*) FCFS & Priority Scheduling

   (*c*) Round Robin & FCFS

   (*d*) SJF & Round Robin Scheduling

10. Consider the following pre-emptive, priority-scheduling algorithm, based on dynamically changing priorities. Larger Priority number implies higher priority. It has following features:

- As a process enters Ready Queue, it is assigned an initial priority 0.
- As a process waits in the Ready Queue, its priority increases at a rate a.
- Highest priority process is dispatched to CPU.
- As a dispatched process is running, its priority increases at a rate $\beta > \alpha$.
- A higher priority process in Ready Queue can pre-empt a running process. Will a running process be ever pre-empted? Justify your answer.

❑❑❑

SECTION C

NOTES

UNIT 6 UNIX OPERATING SYSTEM

★ LEARNING OBJECTIVES ★

- Overview of UNIX Operating System
- Versions of UNIX
- Brief History of UNIX
- Features of UNIX
- Filters
- Pipes
- Summary
- Self-Assessment Questions

OVERVIEW OF UNIX OPERATING SYSTEM

Simply stated, UNIX is an operating system. It is by and large the most popular operating system existing today. The features and flexibility of UNIX is so immense that it has become a standard for great many operating systems. It is multi-user system, which means that more than one user can work at the same computer system at the same time. UNIX also supports multi-tasking. Multitasking means that more than one program can be made to run at the same time. For example, you can initiate a program and leave it by itself to go on and in the meantime you can work on some other program. Multi-tasking and multi-user are the two most important characteristics of UNIX, which have helped it gain widespread acceptance among a large variety of users.

The UNIX OS files consume 40 MB of the 80 MB disk space. Another 10–20 MB of disk space is eaten up as swap space. This swap space is used at that point of time when UNIX falls short of memory. So, the contents that are not immediately required are stored in the swap space. Any time when the program needs these contents, they are read from the swap space.

NOTES

VERSIONS OF UNIX

The original version of UNIX actually came from AT&T. Because of the great deal of flexibility offered by UNIX, many new companies emerged and brought out their own variations. Some of the popular versions of UNIX are given in Table below.

Version	*Developed by*
UNIX	AT&T
AIX	IBM
XENIX	SCO
ULTRIX	DEC (Digital Equipment Corporation)
UNICOS	Cray Research
Sun OS	Cray Research
BSD	University of California at Berkeley
Dynix	Sequent
HP/UX	Hewlett –Packard

BRIEF HISTORY OF UNIX

The original version of UNIX came in the late 1960's. It was designed by Ken Thompson at AT&T Bell Laboratories. At that point of time, Bell Labs were busy in designing a very big operating system called Multics. Their objective was to provide a very sophisticated and complex multi user system which had support for many advanced features. However, Multics failed because the state of art provided by it at that time was too complex.

Therefore, Bell Labs had to withdraw themselves from the Multics project. Ken Thompson then started working on a simpler project and he named it UNIX. This version of UNIX was rewritten in the year 1973. The source code of UNIX operating system was rewritten in C language by Dennis Ritchie, the inventor of C. In order to make UNIX popular among users, AT & T came up with a unique marketing strategy. They started distributing source copies of UNIX to different universities at a very nominal price. This resulted in the widespread popularity of UNIX. In 1974, Thompson and Ritchie described the UNIX System and got it published in a newspaper named Communications of the ACM. This helped in increasing the acceptance level of the UNIX system even more.

By the year 1977, the UNIX system was installed at around 500 different sites. UNIX system found its major contribution in the telephone companies, providing a good environment for program development, network transaction services and real time services. A large number of institutions and universities were provided licenses of UNIX system. In the year 1977, the UNIX system was first ported from a PDP to a non-PDP machine.

So, as the popularity of UNIX grew, many other companies came out with their own versions of UNIX and ported it onto other new machines. From the year 1977 to 1982, Bell Laboratories combined many AT & T variants into a single system and gave it a name UNIX System III. Many new features and advancements were brought out by Bell Laboratories in this version. It was given the name UNIX System V. The people at University of California at Berkeley developed a variant to the UNIX System. Its recent version is called 4.3 BSD for VAX machines. It provided many new and interesting features. By the beginning of 1984, UNIX system was installed at about 1,00,000 different computer sites. It ran on a wide range of computers ranging from a mini computer to a mainframe. No other operating system can make such a claim. Many of the programs of the UNIX Operating System are written in C. However, UNIX system can support many other languages also like Fortran, Basic, Pascal, Ada, Cobol, Lisp and Prolog.

UNIX Architecture

The interaction between the user and the hardware happens through the operating system. The operating system interacts directly with the hardware. It provides common services to programs and hides the hardware intricacies from them. The high level architecture of the UNIX system has been shown in Figure 2.1.

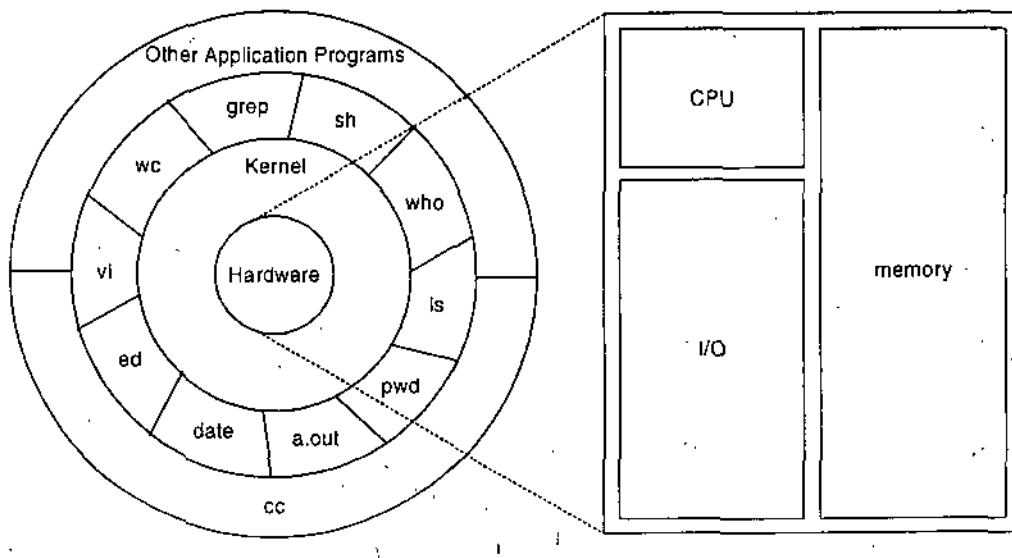


Figure 2.1. *System Architecture of UNIX.*

NOTES

The hardware of a UNIX system is present in the centre of the diagram. It provides the basic services such as memory management, processor execution level, etc. to the operating system. The UNIX system seems to be organised as a set of layers. The system Kernel is also called the operating system. The user programs are independent of the hardware on which they are running. Therefore, it becomes very easy to run programs on UNIX system running on different hardware if the programs do make special assumptions. The programs such as the shell and editors like (ed and vi) interact with the Kernel by invoking a well defined set of system calls. The system calls get various actions done from the Kernel for the calling program. They interchange data between the Kernel and the program. There are many other programs in this layer which from a part of the standard system configurations. These programs are known as commands. But there are several other user created programs present in the same layer. It is shown by the program whose name is a.out. 'a. out' is the standard name for all the executable files produced by the C compiler. The outer most layer contains other application programs which can be build on top of lower level programs. For instance, the C compiler, cc, appears in the outermost layer of the figure. It invokes a C preprocessor, compiler, assembler and link loader. These are all separate lower level programs. The programming style offered by the UNIX system helps us to fulfill a task by combining the existing programs.

File System and Internal Structure of Files

A file in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the Kernel does not impose any structure on files, and no meaning is attached to its contents – the meaning of bytes depends solely on the programs that interpret the file. This is not true of just disc files but of peripherals devices as well. Magnetic tapes, mail messages, character typed on the keyboard, line printer output, data flowing in pipes– each of these is just a sequence of bytes as far as the system and the programs in it are concerned.

Files are organized in tree structured directories. Directories are themselves files that contain information on how to find other files. A path name to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically it contains of individual file name elements separated by the slash character. For example in /usr/rohit/ data, the first slash indicates the root of the directory tree, called the root directory. The next element usr is a sub directory of the root, rohit is a sub directory of usr and data is a file or a directory in the directory rohit.

The UNIX file system supports two main objects : files and directories. Directories are nothing but files which have a special format. So, let us first learn the representation of a file.

Representation of Data in a File

All the data entered by the user is kept in files. Internally the data blocks take up most of the data that has been put in files. Each block on the disk is addressable by a number. Associated with each file in UNIX is a little table called inode which contains the table of contents to locate a file's data on disk. The table of contents consists of a set of disk block numbers. An inode maintains the attributes of a file, including the layout of its data on disk. Disk inodes consists of the following fields.

Field	*Description*
Mode	Specification of access permissions for the owner and other users
UID	ID of the user creating the file
GUID	Ids of user group having permissions on this file
Length in bytes	Number of bytes contained in the file
Length in blocks	Number of blocks to implement the file
Last modification date	Time the file was written to
Last access date	Time the file last read
Last inode modification	Time the file last modifeid
Reference count	Number of directories in which the file appears; this field is used to detect when all the file refernces are deleted from all the directoriesso that the space may be released
Block reference	Pointer and indirect pointer to blocks in the file

The data on a file is not stored in a contiguous section of the disk. The reason behind is that the Kernel will have to allocate and reserve continuous space in the file system before allowing operations that would increase the file size. For instance, let us suppose that there are three files A, B and C. Each file consists of 10 blocks of storage and suppose the system allocated storage for the three files contiguously as shown in Figure 2.2.

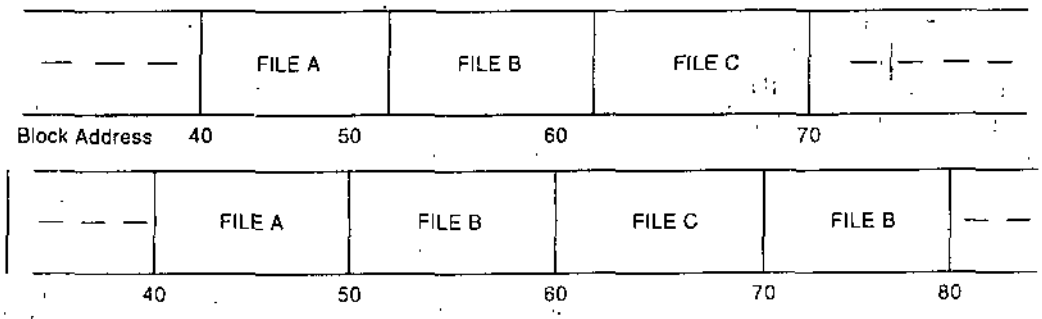


Figure 2.2

NOTES

However, if the user now wishes to add 5 blocks of data in the file B, then the Kernel will have to copy the file to such a place where a file of 15 blocks can be accommodated. Moreover, the previously occupied disk block by file B's data can only be used in a case where the files have data less than 10 blocks.

The Kernel allocates the file space of one block at a time. This allows the data to be spread throughout the file system. In this case, locating the data of a file becomes a complicated process. If a block contains 10K bytes, then such a file would need an index of 100 block numbers and a block of 100K bytes would need an index of 1000 block numbers. Thus, the size of the inode would keep varying according to the size of the file.

The UNIX system contains 13 entries in the inode table of contents as shown in Figure 2.2. The blocks which are marked as "direct" in the figure contain the number of disk blocks that contain the data. The "single indirect" block contains a list of direct block addresses. Thus, if the data has to be accessed through the indirect block, then the Kernel first finds out the appropriate direct block entry from the indirect block and there after reads the data present in the direct block. The block marked as "double indirect" contains a list of indirect block numbers and the block marked as "triple indirect" contains a list of double indirect block numbers. Let us suppose that a logical block on the file system holds I K bytes and a block number is addressable by a 4 byte integer. In such a case, a block can hold upto 256 block numbers. The maximum number of bytes that a file can hold is found out to be 16 gigabytes. It will make use of 10 direct blocks, 1 indirect, 1 double indirect and 1 triple indirect block in the inode.

Directories

The directories are files that give the file system a hierarchical structure. In a directory the data is put in a sequence of entries. Each such entry contains an inode number and the name of a file present in the directory. The pathname is a null terminated character string. The pathname is divided into separate parts by the / (slash) character. Each component of the pathname should hold the name of a directory. However, the very last component can be a non-directory file. The component names can have a maximum of 14 characters, with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes. Each directory contains the file names dot and dot-dot ("." and ".."). The inode numbers of these directories are those of the directory and its parent directory respectively. The inode number of "." in "\etc" directory is present at offset 0 in the file and its value is 83. The inode number of ".." is present at the offset 16 and its value is 2. Any directory entry can also be kept empty. Its inode number is indicated by 0.

The data stored by the Kernel for a directory is similar to the data stored for an ordinary file. For the directory also the Kernel makes use of the inode structure and direct and indirect blocks. The access permission of a directory have the following meaning : the read permission allows a process to read a directory, write permission allows a process to create new directory entries and remove the old directory entries. It accounts for altering the contents of a directory. The execute permission allows a process to search the directory for a filename.

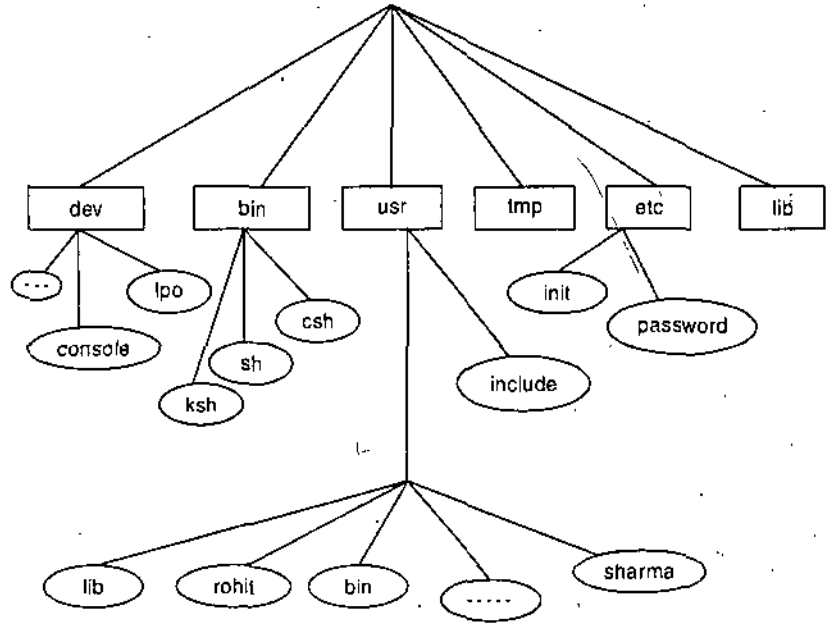


Figure 2.3

Figure 2.3 shows a typical UNIX File System. The file system is organized as a tree with a single root node called the root (written " / "); every non-leaf – node of the file system structure is a directory, and leaf nodes of the tree are either directories or regular files or special devices. /dev contains device files, such as /dev/console, /dev/lp0, /dev/mnt and so on.

The /bin directory contains the executable files for most UNIX commands. UNIX commands are either executables 'C' programs or shell scripts, which can perform a task at our request. Shell scripts are a sequence of UNIX commands grouped under a single name. This can be executed by typing in the name of the shell script. We shall deal with Shell Scripts in later sessions.

The /etc directory contains other additional commands that related to system maintenance and administration. It also contains several files which store the relevant information about the users of the system, the terminals and devices connected to the system.

The /lib directory contains all the library functions provided by UNIX for the programmers. Programs can be written in UNIX using these library functions by making system calls.

NOTES

The /dev directory stores files that are related to the devices. UNIX has a file associated with each of the I\O devices. We shall see more about this in the coming sessions.

The /usr directory is created for each user to have a private workarea where the user can store his files. This directory can be given any name. Here it is named as "usr". This is called user's HOME directory. Thus every user has a HOME directory. Within the "usr" directory, there is another directory, "bin" which contains additional UNIX commands.

The /tmp directory is the directory into which temporary files are kept. The files stored in this directory are deleted as soon as the system is shutdown and restarted. This directory does not have that much importance when compared to the others.

Create, open, read, write are system calls, which are used for basic file manipulation. The create system call; given a path name creates a empty file. An existing file opened by the open system call, which takes a path name and a node and returns a small descriptor which may then be passed to a read or write system call to perform data transfer to or from the file.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each read or write updates the current offset into the file, which is associated with file table entry and is used to determine the position in the field for the next read or write.

Blocks and Fragments

Most of the file system is taken up by data blocks, which contain whatever the users have put in their files.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file system usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1 BSD file system was limited to 1024 – byte block.

The 4.2 BSD solution is to use two block sizes for files which have no indirect blocks : all the blocks of the file are large block size except the last. The last block is an appropriate multiple of a smaller fragment size to fill out the file. Thus, a file of size 18000 bytes would have two 8K blocks and one 2K block fragment.

The block and fragment sizes are set during the file – creation according to the intended use of the file system : if many small files are expected, the fragment size should be small; if repeated transfer of large files are expected, the basic block size should be large.

FEATURES OF UNIX

UNIX is such an operating system that can be run on a wide range of machines, from microcomputers to mainframes. There are a variety of reasons which have made UNIX an extremely popular operating system.

Portability

As of today, there are innumerable computer manufacturers all-over the world. Therefore, the hardware configurations keep varying from one vendor to another. The positive and strong thing about UNIX is that it is running successfully on all these computers. The reason behind UNIX's portability is that it is written in a high-level language which has made it easier to read, understand and change. Its code can be changed and compiled on a new machine. PCs, Amigas, Macintoshes, Workstations, Minicomputers, Super Computers and Mainframes run the UNIX operating system with equal case successfully.

Machine Independent

The UNIX system does not expose the machine architecture to the user. Thus, it becomes very easy to write applications that can run on micros, minis or mainframes.

Multi-user Capability

As discussed earlier, UNIX is a multi-user system. A multi-user system is a system in which the same computer resources like hard disk, memory etc can be used or accessed by many users simultaneously. Each user is given a terminal (a keyboard and a monitor). Each terminal is an input and an output device for the user. All the terminals are connected to the main computer. So, a user sitting at any terminal can not only use the data or the software of the main computer but also the peripherals like printers attached to it. In UNIX terminology, the main computer is called the server or the console. The number of terminals that can be connected to the server depends upon the number of parts present in the controller card. For instance, an 8-port controller card in the host machine can support 8 terminals.

Multitasking Capability

UNIX has the facility to carry out more than one job at the same time. This feature of UNIX is called multitasking. You can keep typing in a program in its editor while at the same time execute some other command given earlier like copying a file, displaying the directory structure, etc. The latter job is performed in the background and the earlier job in the foreground Multitasking is achieved by dividing the CPU time intelligently between all

the jobs that are being carried out. Each job is carried out according to its priority number. Each job gets appropriately small timeslots in the order of milliseconds or microseconds for its execution giving the impression that the tasks are being carried out simultaneously.

Software Development Tools

UNIX offers an excellent environment for developing new software. It provides a variety of tools ranging from editing a program to maintenance of software. It exploits the power of hardware to the maximum extent of effectiveness and efficiency.

Built in Networking

UNIX has got built in networking support with a large number of programs and utilities. It also offers an excellent media for communication with other users. The users have the liberty of exchanging mail, data, programs, etc. You can send your data at any place irrespective of the distance over a computer network.

Security

UNIX supports a very strong security system. It enforces security at three levels. Firstly, each user is assigned a login name and a password. So, only the valid users can have access to the files and directories. Secondly, each file is bound around permissions (read, write, execute). The file permissions decide who can read or modify or execute a particular file. The permissions once decided for a file can also be changed from time to time. Lastly, file encryption comes into picture. It encodes your file in a format that cannot be very easily read. So, if anybody happens to open your file, even then he will not be able to read the text of the file. However, you can decode the file for reading its contents. The act of decoding a coded file is known as decryption.

What makes UNIX Unique

- UNIX and its variants are the only operating systems that are written in a high-level language. This gives it the benefit of machine independence and portability. It becomes very easy to understand, change and move it to other machines.
- It was the first operating system to bring in the concept of hierarchical file structure. It becomes very easy to organise and search for different files.
- It uses a uniform format for files. This makes the application programs to be written easily. This file format is called the byte stream. UNIX

treats every file as a stream of bytes. Therefore, the user can manipulate his file in the manner he wants.

- It provides primitives that allow more complex and complicated programs to be built from simpler ones.
- It has a very simple user interface that has the power to provide all the services the users want. It supports both character-based and graphical based user interfaces.
- It hides the machine architecture from the user. This helps the programmer to write different programs that can be made to run on different hardware configurations.
- It provides a simple, uniform interface to peripheral devices.
- It is a multi-user, multiprocess or operating system.

Steps to Login

1. Logging in is a procedure that tells the UNIX System who you are; the system responds by asking you the password. So, in order to log in, first, connect your PC to the UNIX system. After a successful connection is established, you would find the following prompt coming on the screen.

   Login :

   Each user on the UNIX system is assigned an account name which identifies him as a unique user. The account name has eight characters or less and is usually based on the first name or the last name. It can have any combination of letters and numbers.

2. Thus, if you want to access the UNIX resources, you should know your account name first. If you don't as yet have an account name, ask the system administrator to assign you one. Now, at the login prompt, enter your account name. Press Enter Key. Type your account name in lowercase letters. UNIX treats uppercase letters differently from lowercase letters.

   Login : Pankaj

   Password :

3. Once the login name is entered, UNIX prompts you to enter a password. While you are entering your password, it will not be shown on the screen. This is just a security measure adopted by UNIX. The idea behind is that people standing around you are not able to look through the secret password by looking at the screen. Be careful while you are typing your password because you will not be able to see what you have typed. However, if you give either the login name or the password wrong, then UNIX denies you the permission to access its resources. The system then shows an error message on the screen which is given below :

NOTES

```
Login : pankaj
Password :
Login incorrect
Login :
```

Many UNIX systems give you three or four chances to enter your login and password correct. So, key in your correct login name and the password again.

4. Once you have successfully logged on by giving a correct login and password, you are given some information about the system, some news for users and a message whether you have an electronic mail or not.

```
Login : shefali
Password :
Last login : Sun May 14 19 : 00 : 29 on ttyAe
You have mail
$
```

The dollar sign is the UNIX's method of telling that it's ready to accept commands from the user. You can have a different prompt also in a case where your system is configured for showing a different prompt. By default a $ is shown for the Korn or Bourne Shells.

5. At this point, you are ready to enter your first UNIX command. Now, when you are done working on your UNIX system and decide to leave your terminal-then it is always a good idea to log off the system. It is very dangerous to leave your system without logging out because some mischievous minds could tamper with your files and directories. They can delete your files. They can also read through your private files. Thus, logging off the system is always a better idea than turning off you terminal. In order to log off the system, type the following command :

```
$ exit
login :
```

The above command will work if you are using a Bourne or a Korn shell. However, if you are working on C shell, exit will work or you can give another command to log off.

```
$ logout
login :
```

What can go Wrong When Logging In ?

In order to maintain security, UNIX system is very particular not to allow the unauthorized users to access the system. So, when a message like 'Login

NOTES

denied' comes on the screen, it does not tell you what was wrong with your login the login name or the password. You can say that a UNIX system is a little vague when it denies you access to the system. There could be a few things you could keep in mind while logging in.

1. UNIX is very much case sensitive. Entering SHEFALI is different from Shefali in UNIX. So, keep in mind the case of characters while entering otherwise it can really frustrate you.
2. While entering a username or password, you could have accidentally mistyped a character. This can change the meaning of the information you have typed in. Thus, in order to remove the mistyped character, make use of the Backspace key. It will help you to erase off the misspelled character.
3. You can also have a slow system. If you have entered the login information and nothing happens, then the system might have slowed down especially when many people are trying to log on simultaneously. However, if your system does not respond after several minutes, then you need to check up with your system administrator to make sure that the system is functioning properly.

Sometimes, it might happen that you have keyed in the correct information and still you are told that the information given by you is incorrect. Under such circumstances, you are required to check up with your system administrator. Possibly, he could have changed your login name or password.

Changing your Password

You can change your password with the 'passwd' command. It is advisable to keep changing your password every few days. This helps in maintaining security. UNIX is very careful about the whole procedure of changing passwords. First UNIX wants you to enter your old password to make sure that a valid user is changing the password. The procedure of changing passwords is very simple. In order to change your password, you first have to log on to the UNIX system. Then issue the 'passwd' command at the UNIX prompt as shown below :

```
$passwd
Changing password for shefali
Enter old password:
Enter new password:
Re-type new password:
$
```

Thus UNIX wants you to type in your old password. Then it asks for the new password. Finally, UNIX confirms your new password by asking you to

type in the new password again. If by any means, any mismatch happens, then the UNIX system warns you that the information provided by you is inconsistent as shown below :

```
$ passwd
Changing password for shefali
Enter old password:
Enter new password:
Re-type new password:
Mismatch-password unchanged
$
```

As stated earlier also, UNIX doesn't show the passwords typed by you on the screen because of security reasons.

Before you proceed to the next section, answer the following questions.

1. Can you change Passwords in UNIX.
2. Give procedure to change Password in UNIX.

If your answers are correct, then proceed with the next section.

Choosing a New Login Name and Password

It is said to be a good practice to choose good secret passwords or combinations and keep them secure. Here are a few tips for selecting the right login name and password.

1. Your login name should not be less than two characters and normally should not exceed eight characters. Moreover, it should begin with a lowercase.
2. Choose a password that is greater than at-least six characters. The smaller the password, the more vulnerable it is to be leaked out.
3. Don't at all use easily guessable words like sun, moon, stars, etc.
4. Don't choose a password based on personal information like your spouses name, your middle name, your surname, your job title, etc.
5. The password should contain two alphabets and one numeric or a special character. It cannot include spaces in it.
6. Never choose a password that is very complicated and not easily memorizable. In such a case, the chances of leaving a copy of the password on a piece of paper near the computer rise. Therefore, keep a password that is neither too short, nor too difficult to remember.
7. You should avoid picking up a password from the computer dictionary.

You should make sure that you remember your password. When you enter your password, UNIX encrypts it and saves it in a specific file. In a case, if

NOTES

you forget your password, then even the system administrator cannot decrypt it. Therefore, he will have to create an entirely new account for you. Now, you must have realized how important security is on a UNIX system. UNIX also offers a variety of tools to maintain security. One such tool is the usage of the 'lock' command. This command is found on many systems. The lock command locks your keyboard till the time you enter a valid password, as shown below :

```
$lock
Password :
Sorry
Password :
```

If the correct password is not entered, then the lock command does not give control of the keyboard. If the correct password is not entered in a certain number of chances, then the lock command logs you off the system automatically. It is a very powerful tool to use if you are leaving your system just for a few minutes and want to make sure that no mischievous hands fiddle with your files and directories.

UNIX Command Structure

There are a few of UNIX commands, that you can type them stand alone. For example, Is , date, pwd, logout and so on. But UNIX commands generally require some additional options and/or arguments to be supplied in order to extract more information. Let us find out the basic UNIX command structure.

The UNIX commands follow the following format :

Command [options] [arguments]

The options/arguments are specified within square brackets if they are optional. The options are normally specified by a '-' (hyphen) followed by letter, one letter per option.

Some Commonly Used UNIX Commands

There are many commands you will use regularly. Let us discuss some of these commands, but please make a note that the options that are specified for the commands may not necessarily work on all look-alike UNIX. There might be some variations, (which can be ignored) here and there. The commands covered in this unit are :

banner – To display information.

cal – To display calendar on the screen.

date – To display and set the current system date and time.

passwd – To install or change the password on the system.

who – To determine the currently logged users on the system.

finger – Gives specific information about a user.

NOTES

The banner Command

This command displays information. It displays its argument exploded to a bigger size, onto the standard output. The banner command splits up the long arguments on individual word boundaries. It displays the argument up to 10 characters in large letters.

Options : None

Example : $ banner VICKY

Output :

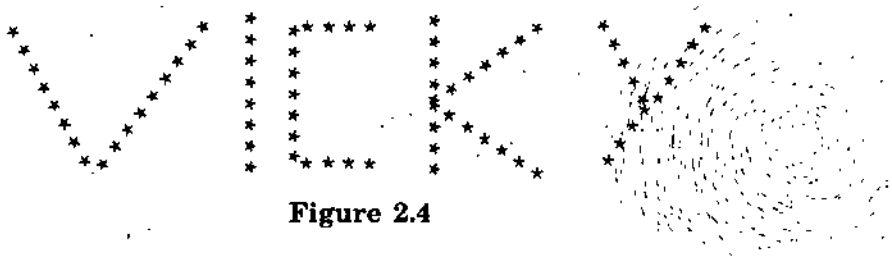


Figure 2.4

The cal Command

The cal command creates a calendar of the specified month for the specified year. If you do not specify the month, it creates a calendar for the entire year. By default this command shows the calendar for the current month based on the system date. The cal writes its output to the standard output.

Syntax : cal [[mm] yy]

where mm is the month, an integer between 1 and 12 and yy is the year, an integer between 1 and 9999. For current years, a 4-digit number must be used, '98' will not produce a calendar of 1998.

Options : None

Examples :

(*i*) $ cal

Output :

May 1999

S	M	T	W	TH	F	S
						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	29
30	31					

NOTES

(*ii*) $ cal 1998

The above command displays the calendar for entire year, 1998. The entire year will whip by, month by month.

(*iii*) $ cal 1998/lpr

The above command prints the calendar for the entire year onto the printer.

The date Command

It shows or sets the system date and time. If no argument is specified, it displays the current date and the current time.

Syntax : date[+options]

Options : %d – displays date as mm/dd/yy

%a – displays abbreviated weekday (Sun to Sat)

%t – displays time as HH :MM :SS

%r – displays time as HH :MM :SS(A.M/P.M.)

%d – displays only dd

%m – displays only mm

Examples :

(*i*) $date

It will display date as – Mon June 9 04 : 50 :24 EDT 1998.

(*ii*) $ date +%d

It will display date as – 11/12/98

(*iii*) $date +%r

It will display time as – 07 : 20 :50 PM

(*iv*) If you are working in the superuser mode, you can set the date as shown below :

$ date MMddhhmm[yy]

where MM = Month (1-12)

dd = day (1-31)

hh = hour (1-23)

mm = minutes (1-59)

yy = Year

It sets the system date and time to the value specified by the argument.

The passwd Command

The passwd command allows the user to set or change the password. Passwords are set to prevent unauthorized users from accessing your account.

Syntax : passwd [user-name]

Options : -d Deletes your password

-x days This sets the maximum number of days that the password will be active. After the specified number of days you will be required to give a new password.

-n days This sets the minimum number of days the password has to be active, before it can be changed.

-s This gives you the status of the user's password.

The above options can be used only by the superuser.

Example :

$ passwd -x 40 bobby.

The above command will set the password of the user as 'bobby' which will be active for a maximum of 40 days.

Also note, that passswd program will prompt you twice to enter your new password. If you don't type the same thing both the times, it will give you one more chance to set your password.

$passwd : bobby

Old password :

New password :

Re-enter new password :

The who Command

The who command lists the users that are currently logged into the system.

Syntax : who [options]

Options : -u – lists the currently logged-in users.

-t – gives the status of all logged-in users.

am i – this lists login-id and terminal of the user invoking this command.

Examples :

(i) $who –t

Output :

Shefali	+	ttyo1	Jan	12	9:50	
Bobby	-	ttyo2	Jan	12	10:10	

NOTES

The second column here shows whether the user has write permission or not.

(*ii*) $who -u

Output :

Shefali ttyo1 Jan 12 9:50 1235

Bobby ttyo2 Jan 12 10:10 2401

The last column here denotes the process-id.

(*iii*) $who am i

Output :

Shefali ttyp6 Jan 12 14:34

This command shows the account name, where and when I logged in. It also shows the computer terminal being used.

The Finger Command

In larger system, you may get a big list of users shown on the screen. The finger command with an argument gives you more information about the user. The finger command followed by an argument can give a complete information for a user who is not logged onto the system.

Syntax : finger [user-name]

Options : none

Examples :

(*i*) $ finger shefali

This command will give more information about shefali's identity as shown below :

Login name : shefali

(512) 222-4444

Directory : /home/shefali

Last login Fri May 16 12 : 14 : 40 on ttyo1

Project : X window programming

Shefali ttyol May 18 20 :05

(*ii*) If you want to know about everyone currently logged onto the system, give the following command :

$finger

File System, Commands, Permissions Changing Order and Group

As we have already discussed, file, in a UNIX system is a unit of storing information. All utilities, applications and data are represented as files.

The file may contain executable programs, texts or databases. They are stored on secondary memory storage such as a disk or magnetic tape.

Naming Files

NOTES

You can give filenames upto 14 characters long. The name may contain alphabets, digits and a few special characters. Files in UNIX do not have the concept of primary or secondary name as in DOS, and therefore file names may contain more than one period(.).

Therefore, the following file names are all valid filenames :

mkt.c name2.c .star a.out

However, UNIX file names are case sensitive. Therefore, the following names represent two different files in UNIX.

mkt.c Mkt.c

File Types

The files under UNIX can be categorized as follows :

1. Ordinary files.
2. Directory files.
3. Special files.
4. FIFO files.

We are discussing about these files below.

Ordinary Files

Ordinary files are the one, with which we all are familiar. They may contain executable programs, text or databases. You can add, modify or delete them or remove the file entirely.

Directory Files

Directory files, as discussed earlier also represent a group of files. They contain list of file names and other information related to these files. Some of the commands which manipulate these directory files differ from those for ordinary files.

Special Files

Special files are also referred to as device files. These files represent physical devices such as terminals, disks, printers and tape-drives etc. These files are read from or written into just like ordinary files, except that operation on these files activates some physical devices. These files can be of two types Character device files and block device files. In character device files data is handled character by character, as in case of terminals and printers. In block device files, data is handled in large chunks of blocks, as in the case of disks and tapes.

FIFO Files

FIFO (first-in-first-out) are files that allow unrelated processes to communicate with each other. They are generally used in applications where the communication path is in only one direction, and several processes need to communicate with a single process. For an example of FIFO file, take the pipe in UNIX. This allows transfer of data between processes in a first-in-first-out manner. A pipe takes the output of the first process as the input to the next process, and so on.

File Names and Metacharacters

In UNIX, we can refer to a group of files with the help of METACHARACTERS. These are similar to wild card characters in DOS. The valid Metacharacters are *, ? and [ ].

* replaces any number of characters including a null character.

? used in place of one and only one character.

[ ] brackets are used to specify a set of a range of characters.

Examples :

(*i*) $ ls *.c

It will list all files starting with any character or characters and ending with the character c.

(*ii*) $ robin*

It will list all the files starting with robin and ending with any character.

(*iii*) $ ls x ?yz*

It will list all those files, in which the first character is x, the second character can be anything, the third and fourth characters should be respectively y and z and that the rest of the name can be anything.

(*iv*) $ ls l[abc] mn

It will list all those files, in which the first character is l, the second character can be either a, b or c the last two characters *i.e.,* 3rd and 4th should be m and n respectively.

Alternatively the above command can also be given in the following manner.

$ ls l[a-c]mn

You should be very careful in using these metacharacters while deleting files, they may unintentionally erase a lot many files at one time.

File Security and Ownership

NOTES

The data is centralized on a system working with UNIX. However, if you do not take care of your data, then it can be accessed by all other users who login. And there cannot be anything private to a person or a group of persons. The first step towards data security is the usage of passwords. The next step should be to guard the data among these users. If the number of users are small, it is not much of a problem. But it can be problematic on a large system supporting many users. UNIX can thus differentiate a file belonging to an individual, the owner of a file or group of users or the others with different limited accesses, as the case may be. The different ways to access a file are :

Read (r) – You can just look through the file

Write (w) – You can also modify it

Execute (x) – You can just execute it

Therefore if you have a file called vendor.c and that you are the owner of it, you may provide yourself with all the rights rwx [read, write and execute]. You can provide rx (read, and execute) rights to the members of your group and only the x (execute) right to all others.

Normally, when you create a file, you are the owner of the file and your group becomes the group id for the file. The system assigns a default set of permissions for file, as set by the system administrator. The user can also change these permissions at his will. But only a superuser can change these permissions (rwx), ownership and group id's of any file in the system.

Giving the execute permission to the data file is absolutely meaningless. Similarly, giving the write (w) permission to an executable file does not carry any sense. The execute (x) permission on directories mean that you can search through the directory and the write (w) permission means that you can create or remove files in the directory. The read (r) permission means that you can list the files in the directory.

Some Commonly Used Commands

Now, let us discuss some of the commonly used file and directory commands as listed in the following table.

ls	Listing of files and directories
cp	Copying files
mv	Moving or renaming a file
ln	Creating a link to a file
rm	Removing a file

cat	Displaying the contents of files
pwd	Printing current working directory
mkdir	Making a new directory
cd	Changing a directory
rmdir	Removing a directory
chmod	Changing access permission for files
chown	Changing owner
chgrp	Changing group

The ls Command

The ls command is used for listing information about files and directories.

Syntax : ls [-options] [filename]

Here, filename can be the name of the directory or file. If it is directory, it lists information about all the files in the directory and if it is a file, it lists information about the file specified. You can also use metacharacters to choose specific files.

Options :

-a – List all directory entries including. (dot) entries.

-d – Give the name of directories only.

g – Print group id only in the long listing.

i – Print inode number of each file in the first column.

l – Lists in the long or detailed format owner's id only in the long listing.

r – Reverse the order in which file names are sorted.

s – This lists the disk blocks (of 512 bytes each), occupied by a file.

u – Sort file names by time since last access.

t – Sort file names by time since last modified.

C – Display files in columns.

R – Recursively lists all subdirectories.

F – Marks type of each file.

You can make use of more than one option at a given time, just group them together and precede them with "-".

Examples :

(*i*) $ ls-lu /usr/*.c

Output : mkt

(*ii*) $ ls -l /usr/mkt

The above command gives a long listing of the mkt directory, which is inside the usr directory as illustrated below :

drwxr-x- -3 bobby engineer 79 may 15 15 :22 bin

-rwxr-x- -2 bobby engineers 422 jan 12 10 :38 sale.c

(*iii*) $ ls-l/dev

The above command gives a typical listing of the dev directory in the root as illustrated below :

crw- -w- -w 1 root 0,0 Feb 19 12 :05 console

brw- - - - - - 1 root 0,1 Feb 19 11 :08 hpob

brw-rw- - - 1 root 6,2 Jun 06 11 :18 ntl

crw-rw- - - 1 gemale 1,5 Feb 10 12 :30 ttyo4

Let us now analyze the different parts of the output shown above :

-rwxr-x-x 15 bobby engineer 422 Jan1210 :38 purchase.c

1 2 3 4 5 6 7 8

From the leftmost column :

1st column denotes file type. It can be :

- for ordinary file

d for directory file

c for character device file. Here, it is an ordinary file.

2nd column denotes file permissions for the owner, group and others respectively from left. The permissions can be :

r read permission

w write permission

x execute permission

- no permission

Therefore, here rwxr-x—x means the owner has read, write and execute (rwx) permissions, the group has read and execute (r-x) permissions and all others have only execute (x) permission.

3rd column here shows, that there are two links to this file.

4th column here shows, that bobby is the owner of this file.

5th column here shows that engineer is the group name, the owner belongs to.

NOTES

6th column here shows that file size in bytes is 422.

7th column here shows that Jan 12 10 :38 is the date and time, the file was last modified.

8th column here shows that the filename is purchase.c

The cp Command

The cp command creates a duplicate copy of a file.

Syntax : cp file1 file2

Options None

The cp command of UNIX copies one file to another or one or more files to a directory. Here, the file1 is copied as file2. If file2 already exists, it is overwritten by the new file. The file names specified may be full path names or just the file name (current working directory, will then be assumed).

Examples :

(*i*) $ cp mkt.c new_mkt.c

Here the file 'mkt.c' which is present in the current directory will be copied as 'new_mkt.c' in the same current directory.

(*ii*) $ cp *.c /usr/mkt

The above command will copy all the files ending with the letter '.c' to the directory called mkt present under usr directory.

The mv Command

This command moves or renames files.

Syntax : mv file1 file2

Options : None

The mv command moves a file from one directory to the another directory. Here 'file1 refers to the source filename and 'file2' refers to the destination filename. Moving a file to another within the same directory is equivalent to renaming the file. Otherwise also, mv doesn't really move the file, it just renames it and changes directory entries.

Examples :

(*i*) $ mv *.c /usr/mkt

This command will move all the files ending with the letter '.c' to the directory called 'mkt'.

(*ii*) $ mv mkt.c new_mkt.c

Here, mv will rename the file 'mkt.c' present in the current working directory as 'new_mkt.c' in the same working directory.

The ln Command

The 'ln' command adds one or more links to a file.

Syntax : ln file1 file2

NOTES

The ln command here establishes a link to an existing file. File name 'file1' specifies the file that has to be linked and file name 'file2' specifies the directory into which the link has to be established. If the 'file2' is in the same directory as file1 then the file seems to carry names, but physically there is only one copy. If you use the ls –li command, you will find that the link count has been incremented by one and that both the files have the same inode number, as they refer to the same data blocks in the disk. Any changes that are made to one file will be reflected in the other. And if 'file2' specifies a different directory, the file will be physically present at one place but will appear as if it is present in other directory, thereby allowing different users to access the file. It saves a lot of disk space because the file is not duplicated.

But you should note that you should have a write permission to the directory under which the link is to be created.

Examples :

(*i*) $ ln /usr/mkt/mkt.c /usr/mkt1/new_mkt.c

This will create a link for file mkt.c in 'mkt' directory to 'mkt1' directory by the name 'new-mkt.c'.

(*ii*) $ ln myfile.prg newfile.prg

The above command links the file 'myfile.prg' as 'new_file.prg' in the same directory. You can see these files by giving the ls command.

The rm Command

The 'rm' command removes files or directories.

Syntax : rm [options] file(s)

This command is used for removing either a single file or a group of files. When you remove a file, you are actually removing a link. The space occupied by the file on the disk is freed, only when you remove the last link to a file.

Options :

i – confirms on each file before deleting it.

f – removes only those files which do not have write permission.

r – deletes the directory and its contents along with all the sub-directories and their contents.

Examples :

(*i*) $ rm –ir /usr/mkt/new_mkt.c

The above command will remove all the files and sub-directories (and their contents) of the mkt directory and the mkt directory itself.

(*ii*) $ rm /usr/mkt/*.c

This will remove all the c program files (.c) from the mkt directory.

The cat Command

It displays the contents of a file onto the screen.

Syntax : cat file....

The cat writes the contents of one of more files, onto the screen in the sequence specified. If you do not specify an input file, cat reads its data from the standard input file, generally the keyboard.

Examples :

(*i*) $ cat /usr/mkt/new_mkt.c

This command will display the contents of c program file 'new_mkt.c' onto the screen.

(*ii*) $ cat

The above command will get the input from the keyboard as illustrated below :

My first UNIX session

I am enjoying learning UNIX.

Directory manipulation commands

These commands help you to see a directory listing, create a new directory, change a directory and remove directories from the disk. Let us discuss them in detail :

The pwd Command

The pwd command is used for printing the complete pathname of you current working directory, *i.e.,* the directory you are presently in.

Syntax : pwd

Options : None

Example : $ pwd

Output : /usr/project1/mkt

The mkdir Command

This command helps you to create new directories.

Syntax : mkdir dir1 [dir2.....]

The mkdir command allows you to create new directories. You can specify more than one directory name in a single command but they should carry different names. The directory names that you specify can be an absolute or relative pathname. The new directories created will be empty except for containing the dot entries, which are used by the UNIX system.

Options : None

Examples :

(*i*) $ mkdir lesson

This will create a directory called 'lesson' in you current working directory.

(*ii*) $ mkdir /usr/plan

This will create a directory called 'plan' inside the 'usr' directory.

The cd Command

The cd helps you to the change from one directory to another.

Syntax : cd [directory name]

The cd command changes from your current directory to the newly specified directory. cd without an argument takes you to HOME directory. You can use pwd to check whether you have arrived at the current directory or not.

Options : None

Examples :

(*i*) $ cd lesson

The above command will take you to the 'lesson' directory.

(*ii*) $cd /usr/plan

This takes you to the plan directory.

(*iii*) $cd

This takes you to your Home directory.

On issuing the pwd, you can see your HOME directory.

$ pwd

/usr/mkt

The rmdir Command

The rmdir removes directories from the disk.

Syntax : rmdir dir1 [dir2..]

This command removes one or more directories from the directory system. As stated in the mkdir command also, you can specify more than one directory

name but separated by commas. In order to remove a directory, you should have the write permission with you. The directory that you are trying to remove should not contain any files and directories under it. The directory to be removed should not be the current directory.

Examples :

(*i*) $ rmdir lesson

This will remove the 'lesson' directory, present in your current working directory.

(*ii*) $ rmdir/usr/plan

This will remove the plan directory, present inside the usr directory.

File ownership and permission manipulation commands

These commands allow you to change access permissions, owner and group of files. Let us explore these commands in more detail :

The chmod Command

The command chmod will change access permission to a file.

Syntax : chmod [for whom] operation permission file name.

A file is assigned default permissions when it is created so that it is accessed by the owner, group and others. The owner can change these permissions at his discretion using these commands. But only a superuser can change these permissions for any file on the system. In the syntax, 'For whom' denotes the user type, and can be :

u user or the file owner

g the group to which the file owner belongs

o other users, not part of the group

a all users

'Operation' denotes the options to be done, and can be :

+ add permission

- remove permission

= assign permission

' Permission' can be :

r read permission

w write permission

x execute permission

The filename(s) can be : the files on which you want to carry out this command.

Examples :

NOTES

Now let us take an example.

(*i*) First see the file permissions using the ls –1 command for mkt.c as shown below :

$ ls -l mkt.c

Output :

-rwx- -x- - 2 root other 1428 May 15 07 :34 mkt.c

i.e., user has rwx, group has x and all others also have x permission.

(*ii*) Now use the chmod command as illustrated below :

$ chmod u-x g+w o+r mkt.c

The above command remove execute (x) permission for user, give write (w) permission to group and give read (r) permission to all others.

(*iii*) Then again use the ls -l command to verify, if the permissions have been or not set.

$ ls -l mkt.c

Output :

-rw- - wxr-x 2 root other 1428 May 15 07 :34 mkt.c

Alternatively, you could have also used the following commands to do the same work :

$chmod u=rw g=wx o=rx mkt.c

If we use

$ chmod a=rwx mkt.c

$-ls -l mkt.c

Output :

-rwxrwxrwx 2 root other 1428 May 15 07 :34 file.c

In the above command, a=rwx assigns read, write and execute permission to all users.

The chown Command

The chown command changes the owner of the specified file(s).

Syntax : chown new-owner filename

The chown command changes the owner of the specified file (s). This command requires you to be in the superuser mode. The new owner can be the user

ID of the new owner or the new owner's user number. You can also specify the owner by his name. But the new owner should have an entry in the /etc/passwd file. The filename is the name(s) of the file(s), whose owner is to be changed.

Options : None

Examples :

(*i*) $ chown bobby sales.c

The above command now makes bobby the owner of sales.c file

(*ii*) $ ls -l sales.c

Output :

-rwxr-x- - x 1 bobby engineer 1826 May 15 17 :56 sales.c

The chgrp Command

The purpose of chgrp command is to change the group of a file.

Syntax : chgrp group filename.

Only the superuser can use this command. This command changes the group ownership of a file. Here group denotes the new group-ID and filename denotes the file who's group-ID is desired to be changed.

Example :

$ chgrp shefali sales.c

This changes the group-ID of the file 'sales.c' to the group called 'shefali'.

FILTERS

Filters are the programs (or commands) that reads from standard input file, process (or filter) it and write the output to the standard output file. Most UNIX commands are filters as they use standard input and output. For examples, cat is a filter command that joins the contents of the two files into one long stream of data as illustrated below :

$ cat student1 Student2 > class

The above command creates a temporary file 'class' by combining two files 'student1 and 'student2'.

Importance of Filters

The importance of filter commands is that these commands can be combined. The data from one filter can be passed to another filter. For instance, by

using cat filter, you can create a temporary file and by using the nl filter, you can adds the line numbers and store the output in another temporary file as. So, the data of filter 'cat' passes to the filter 'nl' as illustrated below :

nl < class > school

NOTES

Filter Commands

We are discussing below the important filter commands.

The tee Command

The tee is a useful filter command. It not only reads the standard input and sends the output onto the standard output, but also redirect a copy of what it has read, into the file of your choice. It thus forms the concept of T used by the plumbers. Therefore, if you have a piping problems, using number of filters, you can extract some of the data from the middle using the tee.

Examples :

(*i*) You can send the contents of file 'sonia.c' to both printers and another *file 'gandhi.c' by following command :*

$ cat sonia.c | tee gandhi.c | lpr

The above command is equivalent to following two commands :

$ cat soina c > gandhi

$ lpr gandhi

(*ii*) If you wish to append a file using tee, just use it as options shown below :

$ 1s –1| tee –a gandhi c | 1pr

The pg Command

This command displays the contents of a file (S) one page at a time. The format of the command is as follows :

pg [filename....]

Where, the file name is the name (S) of the file (S), you wish to view. You can specify either the absolute pathname or the relative pathname. If you do not specify any filename, it takes its inputs from the standard input file. This command is generally used at the end of the pipeline, because its output is not useful in midway.

After displaying one screen, it displays ' :' at the bottom of your screen, and waits for your response. If you press <RETURN> key it shows the next screen and so on. The various options used with pg commands are listed in Table 2.1.

Table 2.1. Options used with pg command

Option	*Function*
: n <Enter>	Skips the n number of pages before displaying the next page *e.g.*,
: 4 <Enter>	Skip 4 pages before displaying another page.
:-3 <Enter>	Skip back 3 pages before displaying another page
: n <Enter>	Scrolls n number of line forward or backward, *e.g.*,
: 50 < Enter>	Scrolls forward by 50 lines and
: -25 <Enter>	Scrolls backward by 25 lines
: p Program <Enter>	Cause pg to search forward for the first occurrence of the string Program or program
: n [gG]oa <Enter>	Causes pg to search forward for occurrence of the string Goa or goa.
:. Enter>	Causes the current line to be redisplayed.
: $ <Enter>	Causes the last screen of the file to be displayed
: q <Enter>	Quits the pg program, out to the shell.
: h <Enter>	Displays the list of pg commands
: n <Enter>	Takes you to the starting of the nth file, if you specify more than one file to the pg command
: p <Enter>	Takes you to the beginning of the previous file.

The more Command

This command displays the contents of a file (S) one screen (or page) at a time. The format of the command is as follows :

more [filename.....]

This command is similar to the pg command, but with very limited options. At the end of one screen display it displays following message :

—more— (20%) at the bottom of the file.

Here 20% means of that 20% of the file contents are yet to be listed, and that it waits for your response. Pressing spacebar will display the next screen, pressing <Enter> will cause one line at a time to be displayed and typing 'q' will quit the more program. 'More' command displays the contents of one file after the other if you specify more than one filename.

NOTES

The head Command

This command displays the first n lines of a file. The format of the command is as follows :

head [-n] [filename.....]

where n is the number of lines. If you don't specify a value for n the default value is 10 lines. Head is a filter program and can be used anywhere in the pipeline.

Examples :

(*i*) $ head manoj. c

This command will display the first 10 (default) lines of 'manoj. c'

(*ii*) $ head –4 manoj. c

This command will display the first 4 lines of 'manoj. c'

(*iii*) $ cat sanjay. C | head | wc –1

Here since head is without a value of n, you will see first 10 lines by the we –1 command.

(*iv*) $ head –4 < bill > billrepo 2>> & 1

This will takes it input from the file 'bill' and output the first 4 lines in the file 'billrepo'. The error messages if any will also be stored in the output file 'billrepo'.

The tail Command

This command displays the last n lines of a file. The format of the commands as follows :

tail [+/-n] [filename]

Where n, denotes the number of lines. If you do not specify any value for n by default it displays the last 10 lines. The filename is the name of the file you wish to view. But unlike head command, you can specify only one filename here. If you do not specify any filename, it takes its input from the standard input file.

Examples :

(*i*) $ tail manoj. c

This will display the last 10 lines of the file 'manoj.c'

(*ii*) $ tail –3 manoj. c

This will display the last 3 lines of the file 'manoj.c'

(*iii*) $ tail + 30 bill. c

This will display all lines starting from the 30th line to the end of the file.

NOTES

(*iv*) $ cat shallu. c | tail –5 | wc –c

This displays the number of characters in the last 5 lines of the file shallu c.

(*v*) $ tail +30 ritu. c | head –2

This displays lines starting 30th through 50, from the file 'ritu.c'.

The wc Command

This command counts characters, words, and lines in the input file. The format of the command's as follows :

wc [options] [filename.....]

If you don't specify a file name, we take its input from the standard input file and write it to the standard output. You can specify more than one filename at a given time. It will then list the count statistics for each of the file specified, one after the another. If you don't specify an option, it reports all the three counts : Characters, words, and lines.

(*i*) $ wc sarika

This command counts characters, words and lines in the file 'sarika'

(*ii*) $ wc –cw sarika

This command counts characters and words in the file 'sarika'

(*iii*) $ 1s | wc – 1

This command counts the number of lines in your current directory.

(*iv*) $ who –u | wc –1

This command counts the number of users currently logged on the system.

The cut Command

The cut is a filter program and can be used anywhere in a pipeline. This command is a bit different, from the earlier command, in the sense that it allows some field (S) or columns (S) of the file to be manipulated. It cuts the fields specified from a file (S) separated or delimited by a character. The default delimiter is the tab character. If you do not specify a filename, cut will take its input from the standard input file. You can also specify more than one file name. It will first cut from the first file, then the second file and so on. This command cuts specified characters or field (S) from a given file (S). The format of the command is as follows :

cut option [file name....]

The various options used with this command are :

-d delimiter, separating the fields

-c character to cut

-f fields to cut

It is compulsory to specify either the c or the f option to the cut command, without which it will give an error.

NOTES

Example :

Consider the file 'excel' with following data :

Roll no	Name	Class
101	Shalu	BCA
102	Jose	BCA
105	Vijaya	PGDCA
104	Devender	PGDCA
103	Tanuja	BCA

(*i*) $ cut –d" " –f1 excel

The above command will display –

101

102

105

104

103

Here the first field (-f1) was cut, and that the delimiter is a space (" ").

(*ii*) $ cut –f2,3 –d" " excel

The above command will cut 2nd and 3rd field from file 'excel'. The output will be as follows :

Shalu	BCA
Jose	BCA
Vijaya	PGDCA

(*iii*) $ cut excel | cut –d" " –f2 | grep "sh*"

The above command will show all those student names (2nd field) which begin with 'sh'.

(*iv*) $ cut c4, 5 stud_mas

This will cut the 4th and 5th (character) of the file excel.

The paste Command

This command concatenates data from files line by line. The formal of the command is as follows :

paste [-d <character>] filename....

NOTES

where "d" defines the delimiter character, when joining files. Tab is the default delimiter.

Example :

```
$ cut –f1 –d" " < excel > file1
$ cut –f2 –d" " < excel > file2
$ paste –d" " file11 file2
```

Output :

01	Shalu	BCA
02	Jose	BCA
03	Vijaya	PGDCA

The cmp Command

This command compares two files and reports the occurrence of the first difference between the two files. The format of the command is as follows :

```
cmp file11 file2
```

cmp compares the two file on a line wise line basis. The moment it encounters a difference, it comes out of the program, back to the shell prompt, with a message as illustrated below :

```
File2    file1   differ : char 31 line 5
```

i.e., the first difference occurs on 31st character from starting of file1 and it is on 5th line (of file1). If there is no difference between the two files, then cmp just takes you back to the shell prompt.

The diff Command

This command compares two files and reports of their differences. The format of the command is as follows :

```
diff [-e] file1 file2
```

The diff compares the two files on line wise line basis and even if the two lines differ by one character, the whole line is reported.

The sort Command

This command sorts the file on the key file (S) and options specified, Sorting is based on the ASCII collating sequence. The sort is a filter program and it sorts lines form an input file based on the key field (S) and option (S), you specify. The various options used with sort command are listed in Table 2.2.

Table 2.2. Options of sort command

Option	*Function*
-t	Specifies the field separator or the filed delimiter. Default is the tab character.
-n	Indicates numeric sort.
-i	Reverses the sort.
-f	Ignore case while sorting
-u	Unique sort-*i.e.*, only one copy of lines with duplicate keys goes to the output.
+(n)	skips first n fields and sort on (n+1)th field.
-n	stop considering after nth field
-m	It merges the specified files. Input files must be already sorted according to the same specification
-o	Use the file named in the following arguments as the output file.

NOTES

Examples :

(*i*) The command used to sort on first field in the ascending order is given below :

$ sort excel

The above command will sort on the first field *i.e.*, and would result :

01	Shalu	BCA
02	Jose	BCA
03	Tanuja	BCA
04	Devender	PGDCA
05	Vijaya	PGDCA

(*ii*) To sort the file excel on the student name, with space as the delimiter for fields. The output would be as follows :

$ sort –t" " +1 excel

04	Devender	PGDCA
02	Jose	BCA
01	Shalu	BCA
03	Tanuja	BCA
05	Vijaya	PGDCA

NOTES

(iii) $ cat excel | sort –t" " +1 | pg

This will show the sorted output listing page by page. If you specify more than one filename, the sort will sort all the lines of all the input files together. So, sort command can merge the files also.

The grep Command

This command searches the input file (s) for a matching pattern. The grep is an acronym for "globally find regular expression and print". It is very efficient command for searching a pattern of character strings in one or more files. You can use both regular expressions containing ordinary character and metacharacters, in the search pattern. If you do not specify an input file, it takes its input from the standard input file. grep can be used any where in the pipeline.

The format of the command is as follows :

grep [options] search-pattern [filename....]

Options :

-c Reports only the count of matching lines.

-i Reports only the names of files containing lines, with the search pattern.

v Displays those line, not having the search pattern.

n Precede each line by the line number in the input file.

Examples :

(i) $ grep std *.c

The above command will display all lines containing words such as for stdin, stdout, stderr, etc.

(ii) If you want to search the exact match of the "std" string then enclose it within quotes, as shown below :

$ grep "std" *.c

(iii) The following command will display all lines beginning with an uppercase letter.

$ grep " ^ [A-Z]. *

(iv) The following command will display all lines which begin with a blank.

$ grep "^.*" manoj.c

The find Command

This command is one of the most powerful commands of UNIX, with a very complex line structure. Sometimes, you know the name of a file, but you do not know where it is in the directory system. It could be very time consuming,

manually, to search every directory. The find can be used to find a file in the directory system. The format of the fine command is as follows :

find pathname ..... option.....

The find command searches through the directory hierarchy and looks for all those files which match the options specified. It searches every directory path starting with the directory specified in "pathname". find will report the filenames, with full path names, for each file that meets all the conditions specified by the options. The various options used with find command are listed in Table 2.3.

NOTES

Table 2.3. Options used with find command

Option	*Function*
-atime n	The file must have been accessed in the last n days where 'n' is an integer.
-time n	The file must have been changed in the last n days where 'n' is an integer.
-exec command	This executes a UNIX command for each file which meet all the conditions specified in the find command options.
-group name	This selects all files belonging to the group specified.

NOTE : Each option, as usual is preceded by "-". Some options will be followed by an argument. A "+" sign before a number ("n") in an option means "more than". A "-" sign means "less than" For example : the option -atime + 10 means "all files that have not been accessed for 10 days". If you specify a "-" as the pathname, the search starts with your current working directory.

Examples :

(*i*) $ find | -name 'shalu.c' –print

This finds a file called shalu.c, starting the search at the root directory.

(*ii*) $ find | -name 'a*' – print

This finds all those file which begin with the letter 'a'.

(*iii*) $ find | usr | project1 – newer '|etc| manoj –print.

This find all those files which are newer than |etc| manoj. The search will start with |usr|project1 directory.

(*iv*) $ find | -name 'dev' – type d –print

This prints all directories named 'dev' in the system.

PIPES

In UNIX, you can combine certain commands into one compaind command by a technique called pipe or piping. Pipe is the mechanism that combines two adjacent programs or commands by connecting the standard output of one program (or command) to the standard input of the next program (or command). So, the output of one command becomes the input of the next command. Piping is done by using the symbol '|' called a pipe symbol. The Syntax of piping command is as follows :

$ command1 | command2 | command 3.....

Examples :

(*i*) You can pipe the output of who command to the command wc for counting the number of users logged onto your system by giving the following command :

$ who | wc – l

As wc – l caints the number of lines from the listing of who command, you will get the output in a number (such as 4 or 5) indicating the number of users.

(*ii*) To list the programs (*e.g.*, 'ashok' and 'shefali') give the following command :

$ cat ashok shefali | n1 | more

The cat command combines the two data files 'ashok' and 'shefali' , and n1 command reads from these files and adds line numbers. The more command displays the data screen use screen.

Importance of Pipes

Piping is one of the powerful features of UNIX. Using pipes, you can perform a complex task by combining several simple commands. Pipes save your time and disk space. If you use pipes, there is no need to create temporary files that may occupy much of your disk space. For instance, if you do not use pipes to display listing of more than one program (as illustrated in previous command), you will need to create temporary files as shown following commands :

(*i*) First, you will combine the two files ('ashok' and 'shefali') and temporary file 'authors' by giving following command :

cat ashok shefali > authors

(*ii*) Then, you will read the file 'authors', adds the line numbers and stores the output in second temporary file 'friends' by giving the following command :

n1 < authors > friends

(iii) Now, you will read the file 'friends' for displaying the data by follows by command :

more < friends

Thus, two temporary files 'authors' and 'friends' will use your disk space unnecessarily and you will require to delete these files by giving the following command :

$ rm authors friends

So, piping not only makes your task easier but also saves the disk space.

NOTES

Communicating With Others

Anyone may communicate with other users who are logged in at the same time. Using electronic mail, you can also reach other users on those systems who are not logged in, as well as users on other campus computers, and computers all over the world (including North America, most of Europe, Japan, Israel, South Africa, Australia, and many other places). Some of these systems allow for real-time communication as well (using 'talk', for instance).

Who's on ?

There are several commands for finding who else is logged in and what they're doing. These include :

```
users show a simple list of which usernames are logged in
w list which users are logged in, when they logged in, and
what
programs they're running
finger provide names of users currently on, when they
logged in, and
where they logged in from
```

Experiment with all of these commands to see which you find most useful.

finger

The 'finger' command has far more uses than simply displaying a list of people logged in. It's more commonly used to learn more about an individual user of the system :

```
> finger demo
Login name : demo In real life : NSIT Demo Account
Directory : /nfs/harper/h2/demo Shell : /usr/local/bin/
tcsh
Last login Tue Jan 26 10 :18 on ttyp7 from somewhere.uchi
New mail received Wed Jan 27 16 :51 :02 1993;
```

NOTES

```
unread since Tue Jan 26 17 :50 :41 1993
Project : Demos. What else.
Plan :
To figure out what a plan is.
```

Many people 'finger' friends just to see the "New mail received...unread since..." statistics.

You should be aware that the "Last login" field refers only to the Sun you happen to be on : if "demo" generally logs in on harper, and you finger demo on harper, the "Last login" and when-mail-was-read statistics won't make sense. You may even see the message "Never logged in" instead of "Last login...", if your target habitually uses a machine other than the one you're on. To get around this problem, finger people on the machines they use most often : "finger myfriend@harper", for example.

chfn;.project/.plan files

To change your "finger name"- -the "In real life :" field above- -use the `chfn' command.

If you want to have a "Project" and "Plan" of your own, create files in your home directory called ".project" (this should be one line long) and ".plan". Don't forget that the names should begin with a dot. Both files need to be publicly readable, and your home directory must be publicly executable.

Sending Messages to Other Terminals

write

The 'write' utility is the simplest form of communication on a Unix system. If "friendsname" is logged in to the same machine you're on (also on harper, for instance), you can type

```
> write friendsname
```

then a single-line message, which you can end by pressing the return key and typing a Control-D.

The person you've written to will see a message like this on their screen :

```
Message from example@harper on ttyel at 0 :01...
This is a simple 'write' message.
EOF
```

talk (and ntalk)

Using the talk utility, you can communicate interactively with someone else who is currently logged in, even on a different machine. Invoke 'talk' like this :

```
> talk destination-user optional-terminal
```

NOTES

where "destination-user" is the username of the person you're sending the message, and a terminal name is optional (if the person you want to communicate with is logged in on more than one terminal, you must specify the one you want; usually you can leave it out). To send a message to a friend logged into harper as "user," say :

```
> talk user@harper
```

After issuing this command line you will get the following responses :

```
[No connection yet]
[Waiting for your party to respond]
```

If the intended person does not answer immediately, 'talk' will display the messages :

```
[Waiting for your party to respond]
[Ringing your party again]
```

until your correspondent responds. Once you establish contact, you will be able to type your message at the same time as your correspondent, and both messages will be displayed on the screen simultaneously, your message in the top half of the screen and your correspondent's in the bottom half.

If the person you are trying to contact does not respond, you can exit the program by entering a Control-C.

You may also be the object of an intended communication. If someone tries to communicate with you, a message similar to the one below will appear on your screen.

```
Message from Talk_Daemon@harper.UChicago.edu. at 15 :59
... talk : connection requested byafriend@harper.UChicago.edu.
talk : respond with : talkafriend@harper.UChicago.edu.
```

To complete the communication with your friend, say

```
> talkafriend@harper
```

When your correspondent is ready to talk, 'talk' will notify you, and the communication can take place. To exit the program when you are finished talking, enter a Control-C, and you will receive the message

[Connection closing. Exiting]

then be returned to your Unix prompt.

The version of talk running on our systems is not compatible with everyone else's version of 'talk'. If 'talk' doesn't work for you with a friend on a different system, try 'ntalk' instead; while `ntalk' works with some systems that `talk' does not, the two commands function in exactly the same way.

'mesg n'

Occasionally, you'll see the notation "(messages off)" when you finger someone. Or you'll attempt to use 'talk' or 'write' and get this response :

```
[Your party is refusing messages]
```

This means the person has issued the command 'mesg n', which prevents messages to a terminal. If you find yourself being constantly interrupted by such messages, you may want to use the command yourself. (To turn permission to your terminal back on, say 'mesg y'.)

Messages to a terminal from unknown users can be rather disturbing. If you're a stranger to someone, it's generally considered polite to send electronic mail before using 'talk' or 'write' to contact them.

Electronic mail

If the intended recipients of your messages are not currently logged in, you must use electronic mail (email) to communicate with them. Using email software, you can send, receive and read personal mail.

You may also use email to communicate with most of the shared systems which are connected to the campus network, and hundreds of thousands of computers outside of the uchicago.edu domain. For general information on email, please see the section on electronic mail in the NSIT Resource Guide.

mail

The simplest way to send mail to someone with the username "someone" is to type :

```
> mail someone
```

You can mail to several different people at the same time by specifying several usernames, separated by spaces.

When the 'mail' utility starts up, you will be prompted for a subject line. Once you provide one (and press the return key), you can enter your message. When you have finished the message, you can exit mail in one of two ways : by issuing the end-of-file character (Control-D); or by typing a period ('.') as the first character of the last line. When you exit, you will be prompted for 'Cc :', at which point you can enter any usernames you forgot to include in the original command line, or your own username (for your very own personal "carbon copy").

You can also use 'mail' to read mail you have received. If you have received mail, you will be notified when you log in. You may then call up mail by entering

```
> mail
```

to read new incoming messages.

Reading old messages, stored in the file "mbox" in your home directory, is also easy :

```
> mail -f mbox
```

NOTES

Pine

If Elm is still too complex for you, Pine is next on the list. Pine stands for Pine is not Elm (trust me, this is considered prime mail humor), and it's somewhat similar to Elm_but different.

It uses the same one-message-per-line, scroll-through-them-and-use-hotkeys-to-act-on-them principles as Elm, but Pine makes things a little easier. The number of features is less overwhelming, and there's a concerted effort to keep the same keys performing the same functions from screen to screen. Several items (such as address books) you'll have to "suffer" through with Elm rate their own full-screen editors in Pine. Pine even comes with its own text editor, Pico, which can be used as a general text editor. For the faint of heart, it's certainly an improvement over emacs or vi.

Remote Mail Clients

The "Common Mail Programs" section has generally assumed that you will run your mail program on the computer that contains your Internet mail. In many cases, however, you will wish to do all your mail reading on your personal computer, both because you may be charged for all the time you are logged onto your mail account, and because the programs on Macs and PCs are much friendlier than those on many UNIX systems.

What you want is a program that will call the system that receives your mail (or that will connect to it by whatever means necessary), grab all your new mail, and disconnect. Then you can read your mail at your leisure and enter new messages. If there are any new messages, the program should call your mail system and give it the new messages for delivery. As you have probably guessed, these programs exist and are known as mail clients.

The big difference between this approach and the "read your mail on your Internet computer" approach is that your mailbox is kept on your personal computer instead of on the Internet computer.

Obviously, there has to be a way for your mail client to talk to your Internet computer and transfer messages. There are several standards for this.

SMTP—Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (SMTP), or some variation of it (such as Extended SMTP) is used by computers on the Internet which handle mail to transfer messages from one machine to another. It's a one-way protocol—the SMTP client contacts the SMTP server and gives it a mail message.

Most mail client programs support SMTP for sending outgoing mail, simply because it's very easy to implement. Few mail clients support SMTP for incoming mail, because normally your mail computer can't contact your personal computer at will to give it mail. It's possible if your personal computer happens to be permanently networked to the mail computer via EtherNet, for instance, or if your mail computer knows how to use a modem to call your personal computer, but in most cases this isn't done.

POP3 (Post Office Protocol 3)

The standard protocol used by most mail clients to retrieve mail from a remote system is one of the post office protocols, either POP2 or usually its successor POP3. These protocols enable your mail client to grab new messages, delete messages, and do other things necessary for reading your incoming mail. POP only requires a rather "stupid" mail server in the sense that your mail client needs to have most of the intelligence needed for managing mail. It's a very simple protocol, and is offered by most mail clients.

POP3 is somewhat insecure in that your mail client needs to send your account name and password every time it calls. The more you do this, the greater the chance that someone with a network snooper might get both. (I'm not trying to scare you, but it's possible.) An extension known as APOP uses a secure algorithm known as MD5 to encrypt your password for each session.

Finally, note that standard POP3 has no way to send mail back to the mail server. There is an optional extension to POP3 known as XTND XMIT that allows this, but both the client and the server have to support it. Generally, a mail client uses SMTP to send messages and POP3 to retrieve them.

Desirable Features in Mail Clients

Here are some useful features to look for when shopping for a mail client :

- **Delete on retrieve.** The client should have the option to automatically delete mail on the server after it has been downloaded. If you only read mail using your client, you don't want a huge mail file building up on the server. On the other hand, if you only occasionally use your mail client you might want to leave your mail messages on the host so you can access them with your UNIX mail program.
- **Header only retrieve.** You can tell quite a bit about a message just by looking at the message header. If reconnecting to your server is easy, you might want to have your mail program download only the header. Then, if you want to see the actual text of the message, the program will download that. This can be very useful if some idiot mails you a humongous file—you can be spared the time it takes to download the whole thing to your computer.

NOTES

- **Name server support.** A machine name such as mailserv.bozo.edu is actually just a logical name for a computer that is truly identified by its IP number, something that looks like 130.029.13.12. Obviously, the machine name is easier to remember, and if anything happens to mailserv that requires the machine to move to a new IP address (such as a hardware upgrade), the administrators can map the name to the new IP address and you won't even notice. Those who are accessing the machine by number will have to find the new number and enter it. To turn the name into an IP number, though, your client needs to be smart enough to use a domain name server, which keeps track of what numbers go to what names.
- **POP3.** This is the standard way for a mail client to retrieve mail from the mail server. If your client doesn't support this, it darn well better have some way to retrieve mail that your mail server understands (for example, IMAP2 or PCMAIL).
- **Retrieve on start-up.** The client should enable you to immediately contact your mail server and retrieve all unread mail whenever you start it, because this will probably be your most common operation.
- **Separate SMTP server.** In some cases you will need to use a different machine to send mail (using SMTP) than you use to retrieve mail (using POP3). A good mail client should let you specify a different server for each.
- **SMTP.** This is the standard way for a mail client to give mail to the mail server. If your mail client doesn't understand SMTP, it should have some special protocol that your mail server understands to do the same thing (unless you don't want to send mail, of course). Some mail clients support SMTP connections as a way to receive messages, which can be useful if you expect your computer to be hooked up to the network all the time.
- **TCP/IP, SLIP, or PPP.** Your client should be able to access whatever network your mail host is on. Otherwise you'll just be talking to yourself. TCP/IP and TCP/SLIP are the most common network protocols mail programs are likely to need, and PPP is becoming more popular. If you have a SLIP or PPP driver that looks like TCP/IP to your mail program, all it needs is TCP/IP support.
- **Timed retrieval.** The client should be able to automatically connect to your mail server and check for new mail every so often, and beep if it finds new mail. If you're calling in using a modem, you might want to make this every few hours, or even once a day, but if you're directly networked with the server (perhaps via EtherNet), you might want to check every five minutes.
- **Other mail items.** A good mail client makes reading your mail as easy as possible. You shouldn't have to give up any of the features you enjoy under a UNIX mail program. These include a good text editor, header field filtering, an address book (aliases), and multiple mailboxes.

STUDENT ACTIVITY

1. What is meant by a multi-user and multi-tasking operating system ?

2. What are various Features of UNIX ?

3. What is function of a Kernel ?

4. Define Unix file structure.

SUMMARY

- The directories are files that give the file system a hierarchical structure. In a directory the data is put in a sequence of entries. Each such entry contains an inode number and the name of a file present in the directory.
- UNIX is such an operating system that can be run on a wide range of machines, from microcomputers to mainframes.
- UNIX has the facility to carry out more than one job at the same time. This feature of UNIX is called multitasking.
- Filters are the programs (or commands) that reads from standard input file, process (or filter) it and write the output to the standard output file.
- Piping is one of the powerful features of UNIX. Using pipes, you can perform a complex task by combining several simple commands.

SELF-ASSESSMENT QUESTIONS

1. What is the minimum space required of UNIX ?
2. Why are there so many various of UNIX in the market ?
3. Who is the developer of UNIX ?
4. In which year was first version released.
5. What makes UNIX portable and secure ?
6. Can UNIX be used as Network Operating System ?
7. How is UNIX different from other Operating systems like Dos, Windows, MAL ?
8. Which is the most important Part of UNIX Architecture ?
9. Give different layers of UNIX Architecture. Explain the intended purposes of each.
10. What are the various services provided by an Operating System ?
11. Where does Kernel Resides in Memory ?
12. How can the exceptions be resolved in UNIX ?
13. What is the role of processor execution levels ?
14. How are file organized in UNIX ?
15. What is a Path, in context of Files and Directories ?
16. How many types of File owner are there in Unix ?
17. How may files can a user make ?
18. What is the difference between a Directory and a file in UNIX ?
19. What are various steps involved in logging into UNIX ?

20. Can we get to know about last login in UNIX.?
21. Can we view the Password in UNIX.
22. Give the command to log off in UNIX.
23. Is unix case sensitive?
24. What message appears when wrong password is given ?
25. Can you change Passwords in UNIX ?
26. Give procedure to change Password in UNIX.
27. What Parameters should be kept in mind while choosing username and passwords ?
28. Can any other user change your password ?
29. What is Utility of finger command ?
30. How many options can be used with WHO command ?
31. What is the purpose of password command ?
32. How differently col command be used.
33. How many types of files there can be in UNIX ?
34. How can security of files can be maintained ?
35. What are different types of users in UNIX for files ?
36. What is function of ls command ? Give various options.
37. Differentiate between CP and MV Command.
38. Why is rm command used ?
39. Differentiate between chmod and chown Command.

NOTES

SECTION D

NOTES

UNIT 7 SHELL SCRIPT

★ LEARNING OBJECTIVES ★

- Shell as a Programming Language
- Shell Types
- Shell Metacharacters and Variables
- Variables
- Constructs Used in Shell Scripts
- Operators on Numeric Variables
- Operators on String Variables
- Operators on Files
- Logical Operators
- The Case...Esac Construct
- The For...Do...Done Construct
- While Loop
- Until Loop
- Break and Continue
- Editors
- Summary
- Self-Assessment Questions

The shell is UNIX system's Command interpreter. It is basically a program that interprets the command that the user keys in at the command prompt and performs various operations depending on what user types in. It is also used for interactive computing where user gets instant output. Shell also has the ability of redirecting the standard input, output and error files to other devices in place of standard devices. Different commands can also be combined using pipes. You can issue more than one command by using command terminator (;).

The Shell supports UNIX's multitasking feature using the background processing method, where more than one process can be started in the background. It also has the capability of filename expansion, using Metacharacters or wildcards.

NOTES

SHELL AS A PROGRAMMING LANGUAGE

Apart from performing the role of command interpreter, shell is also a programming language. It offers standard programming constructs like loops, conditional branching of control, defining and manipulating variables, file creation, etc.

A Shell Script

The shell script is similar to the batch files of DOS, where a sequence of frequently used UNIX commands are stored in a file and shell is made to read the file and execute the command. Such a file is called a shell script. For example, lets create and execute a shell script.

1. Open a file called Test1 using vi editor.
2. Enter the following command and save the file.

   who

   ls –l
3. To execute the script, enter the following commands.

   sh Test1

   Or

   chmod u +x Test1 (Change the mode of the file)

   $ Test1

sh is a command which creates a sub-shell generally called a child shell process. This child shell reads the Test1 file, executes the commands contained in the file and thereafter returns control to the original shell.

SHELL TYPES

The following types of shell are available in the UNIX system :

1. **Bourne Shell** – Bourne shell is the original command processor developed by At&T and named after its developer Stephen R.Bourne. It is the fastest and the widely used UNIX command processor and can be used on all UNIX systems. The executable filename is sh.
2. **C Shell** – C shell is another command processor developed by William Joy and others at the University of California at Berkeley. Its name has originated from the programming language 'C', as it resembles its syntax. The executable filename is csh.
3. **Korn Shell** – Korn shell is developed by David Korn which combines the best features of both the above shells. Its executable filename is ksh.

The echo Command

The echo command is used to display the messages on screen. In other words, it simply echoes back its argument on to the terminal (screen). For example, the output of the following command is 'Welcome to World of Computers'

```
$ echo Welcome to World of Computers
```

It is preferred to enclose the message that has to be echoed in double-quotes. Some of the above options used with echo are shown below :

Character	Function
\c	Keeps the cursor on the same line of the screen after displaying the argument.
\n	Displays an additional blank line after the argument.
\007	Displays the argument and then sounds the system bell on the next line.

SHELL METACHARACTERS AND VARIABLES

As discussed earlier, the shell offers certain unique features like filename expansion variables, etc. which makes it a powerful command language. The shell uses a set of symbols called metacharacters, to form a pattern to match various classes of filenames which can replace full filenames. These metacharacters are "*", " ?" and "[ ]" symbols.

In certain circumstances metacharacters are to be used literally, for example :

```
$ echo Display filenames of all * in current directory
```

On execution of the above statement when executed, the * is replaced by a list of the all filenames in the current directory. UNIX also provides certain neutralize metacharacters such as backslash (\), Single quotes (') and double quotes (") as listed in Table 7.1.

Table 7.1. Neutralise metacharacters of the shell

Escape Mechanism	*Effect*
\ (back slash)	Negate special properties of the single character following it.
' (pair of single quotes)	Negates special meaning of all characters enclosed within the quotes.
"" (pair of double quotes)	Negates special meaning of all enclosed characters except $',

NOTES

Other Shell Metacharacters

1. Word separators : space, tab, newline – These are special set of three characters used to separate one word from the next.
2. Command terminators – newline, semicolon and ampersand.

Newline is the simplest way of telling the shell that a command has ended. This is inserted on hitting the <Enter> key. For example :

```
$ pwd <Enter>
/user/temp1
```

Semicolon (;) can be used to terminate commands. It helps to put several commands on the same line. For example :

```
$ cal ;date
```

Ampersand (&) is another command terminator which causes the command preceding it to run in the background.

VARIABLES

Like other programming language, shell provides the user the ability to define variables and can assign values to them. A shell variable name must always begin with an upper-case or lower-case letter and may contain numbers and the underscore character. There are certain system variables which have special significance.

System Variables

System (environmental) variables are those which are available to each process as it begins execution as shell maintains its own set of variables.

Mentioned below are some of the standard environmental variables found in many systems.

(a) **Path** – It contains the search path string. The commands given by the user are searched in the directories specified in the path string. An error message is displayed on failure.

Example : /bin/usr/bin

(*b*) **Home** – It specifies the full path name for the user's login directory. The cd command without any argument will look for the contents of this variables and change the directory accordingly.

Example : /usr/user1

(*c*) **Term** – It holds the terminal specification. Being a UNIX system it can have different types of terminals. Typical entries found are vt100, vt200, ansi etc.

(*d*) **Logname** – It holds the user login name.

Example : user1

(*e*) **PS1** – It stores the primary prompt string, which is the dollar sign ($). To change the prompt, simply assign the new value to this variable.

Example : $ PS1="Trial" will change the prompt to 'Trial'

(*f*) **PS2** – It specifies the secondary prompt string, which is displayed for the continuation of commands into the next line. Usually a greater than symbol (>) is assigned to it.

(g) **Shell** – It stores the name of the shell (Bourne, Korn or C). In Bourne shell, the entry sh is found in this variable.

Creating a Variable

The format for creating a variable is :

<variable_name>=

There should be no spaces on either side of the assignment operator.

$ name ="Vivek"

To refer the contents of a variable the $ symbol is used.

$ echo $name

Vivek

Reading a Value into a Variable

Along with assigning a value to a variable, the shell also allows a user to enter a value into a variable during execution of a shell script by using the read command.

Examples :

Lets write a shell script which will accept the Roll No, Name, and Course Name from the user and display the same on the terminal screen.

1. Open a file called Test2 using vi editor.
2. Enter the following lines and save it.

```
echo "Enter your Roll_No : \c"
read r_no
echo "Enter you Name :\c"
read name
echo "Enter your Course_Name :\c"
read c_name
echo "Roll_No : $r_no, Name : $name, Course : $c_name"
```

3. sh Test2 <Enter>

Simple Arithmetic on Shell Variables

The command expr can be used to evaluate arithmetical computations on the shell variables having integer values.

NOTES

```
$ expr 4+10
14
$ expr 8 – 2
6
$ expr 3 \* 4
12
$ expr 45 /5
8
```

The arguments of the expr command must be separated by a blank space from one another. The special symbols must be preceded by a back slash (\) so that the shell does not expand them. The division of expr is an integer division, *i.e.*, it displays only the integer portion of the result.

The output of the expr command can be stored in a variable. Arithmetic computation of variables are also possible through the expr command.

Examples :

```
$ sum='expr 25 + 5'
$ echo $sum                    //    will display 30
$ x=12
$ y=80
$ echo sum is $x + $y
sum is 12 + 80                       will display the arguments.
$ echo sum is 'expr $x + $y'
sum is 92
$ sum='expr $x + $y'                 will store 92 in the variable sum
```

Local and Global Variables

By default the shell variables are 'local' to the shell that creates them. The values of these shell variables are not available to the newly created child process.

export command is provided by UNIX, which can be used to declare a variable as global so that these variables are available to the child process.

Below mentioned example explains the concept of local and global variables.

NOTES

```
$ City = Jaipur
$ echo "${City}"
Jaipur
$ sh                    Creates a new shell
$ echo "${City}"        No response
$ City =Agra            Assigning a new value 'Agra' to city
$ echo "${City}
Agra
$ ctrl d                Returns to parent shell
$ echo "$" {City}       Returns 'Jaipur'. Parent is unaware of 'Agra'
$ sh                    Creates a child shell
$ echo "${City}         value 'Agra' destroyed
$ ctrl d                Return to parent
```

At this stage users can use export command with variable used in the shell so that it can be used by the other programs.

```
$ City = Jaipur
$ export City
echo "${City}"
Jaipur
$ sh                    Creates a new child shell
$ echo "${City}"
Jaipur                  Child shell has the variable 'City'
$
$ City = Agra           Assign a new value to City
$ echo "${City}"
Agra
$ ctrl d                Return to parent
$ echo "${City}"        Parent shell retains its original value 'Jaipur'
```

The last two commands display that the variables can be exported or passed on to subshells, but reverse is not true. Export command causes a copy of the variables and values to be passed onto a child shell.

CONSTRUCTS USED IN SHELL SCRIPTS

NOTES

Like other programming languages, the UNIX shell also offers certain constructs for looping and decision making constructs which can be used in shell scripts.

If...then...else

```
If <condition>
then command(s)
else commands(s)
fi                          <indicates the end of the if construct>
```

The if construct is usually used in conjunction with the 'test' construct. The test command is used to test for the position of files and values of variables. The result of the test command is always 'true' or 'false'.

OPERATORS ON NUMERIC VARIABLES

In shell, the following operators on numeric variables are used :

-eq : equals to
-ne : not equals to
-gt : greater than
-lt : less than
-ge : greater than or equal to
-le : less than or equal to

Examples :

```
$x=10; y=25
$test $x –eq $y      testing for equality of the variables 'x' and 'y'
$echo $ ?            checking the exit status
1                    false, as value of 'x' is not equal to the value of 'y'
$
```

NOTES

OPERATORS ON STRING VARIABLES

In shell, following operators on string variables are used :

= : equality of strings

!= : not equal

-z : zero length (string contains zero characters : null string)

-n : string length is no zero

Examples

$ emp_name="Pankaj"

$ test –z $emp_name — will return the exit status 1 as the string name is not null

$ test –n $emp_name — will return 0 as the string is not null

$ test –z "$emp_address" — will return 0 as the variable has not been defined

$ test $name ="Pankaj"

will return 1 as the value of name is not equal to "Pankaj"

OPERATORS ON FILES

In shell, following operators on files are used :

-f : the file exists and is an ordinary file

-s : the file exists and the file size is non zero

-d : directory exists

-r : file exists and has read permission

-w : file exists and has write permission

-x : file exists and has execute permission

Examples :

$ if test –f "test.doc" — will check the existence of the file test.doc, if it exists and it is an ordinary file.

then echo "It's a Ordinary file"

else echo "Its not a Ordinary file"

fi

returns 0 else 1

$ test –r "test.doc"	will check for the read permission for the file test.doc
$ test –d "$HOME"	will check for the existence of the user's home directory

NOTES

LOGICAL OPERATORS

The logical AND, logical OR and logical NOT operators are used for combining more than one condition.

-a : logical AND

-o : logical OR

! : logical NOT

Examples :

$ test –r "test.doc" –a –w	"test.doc" will check both the read and write permission for the file `test.doc' and returns either 0 or 1 depending upon the result.
$ echo $?<Enter>	will show the exist status of the above command.

THE CASE...ESAC CONSTRUCT

The shell script uses this construct to perform a specific set of instructions depending on the value of a variable and can be used in place of the if construct.

The syntax is as follows :

```
case variablename in
pattern) command;;
pattern) command;;
*) default;;
esac
```

The case statement allows a shell script to choose from a list of alternatives. This statement compares the value of the variable with the patterns from top to bottom and performs the commands associated with the first pattern that matches. The patterns are written using the shell's pattern matching rules. Each action is terminated by the double semicolon ';;'. The word 'esac' marks the end of case construct.

In case, nothing matches the pattern then the command following the "*)" label is executed, it is used as the last choice. If it were provided as the first choice then the commands following it will be executed always and the case statement will ignore all the following patterns.

The case...esac construct is commonly used for setting up menus, where the course of action is to be decided from a set of alternatives.

Examples :

```
case ${opt} in
(1) ls
(2) who
(3) date
(4) cal
esac
```

THE FOR ... DO ... DONE CONSTRUCT

The for construct is the only shell control flow statement that is commonly typed at the prompt rather than putting it in a script file. The commands to be executed are specified between the words do.. done.

The syntax for the for statement is as follows :

```
for <variable> in <list>
do
<commands>
done
```

Examples :

(*i*) A for loop construct to display names one file per line is given below :

```
$ for i in Jaya Aruna Rahul
> do
> echo $i
> done
$
```

(*ii*) Shell script using for loop is given below :

```
For i in 1 2 3 4 5 6 7 8 9 0
```

```
do
echo $i
done
```

The above loop executes for 10 times and displays the numbers 1 2 3 4 5 6 7 8 9 0

NOTES

WHILE LOOP

This loop uses the exit status from a command to control the execution of the commands(s) in the body of the loop. The loop is executed repeatedly until the set of commands condition returns a non-zero status.

The syntax of the while loop is as follows :

```
while <command>
do
<commands>
# loop body executed as long as command returns true (a zero)
done
```

Examples :

The shell program for printing the first 10 positive numbers using the while loop is given below :

```
# using while loop
ctr=1                    #assign a value 1 to a variable ctr
while [$ctr –le 10]
do
echo $ctr
num='expr $ctr+ 1'       # increment num by 1
done
#end of script
```

UNTIL LOOP

This loop checks the exit status of the command, and executes the command(s) enclosed within the do and done statement, until the condition command returns true (a zero status). The syntax for the until loop is :

NOTES

```
until <command>
do
<command(s)>
# loop body executed as long as command returns false.
done
```

Example :

Printing the first 10 positive numbers using until loop.

```
val =1
until [ $val –gt 10]
do
echo $val
num='expr $val + 1'
done
```

BREAK AND CONTINUE

At times, you may require to skip the remaining commands of a loop and return to the beginning of the loop. Sometimes, a need may arise to terminate and exit the loop. The shell provides the commands break and continue for these purposes.

The continue command will skip the remaining statements or commands in a loop and return to the beginning of the loop. Whereas the break command will terminate the loop and exit the loop.

Example :

Let's consider the previous example.

```
while true
do
echo "Enter your option"
echo "Press 5 to exit"
read opt
case ${opt} in
(1)    ls;;
(2)    who;;
(3)    date;;
```

```
(4)     cal;;
(5)     break;;
(*)     echo "Invalid Option"
        esac
        done
        $
```

Whenever user presses 5, the break command is executed and while is terminated.

NOTES

Parameter Handling in Shell Scripts

The shell interprets commands in UNIX. Whenever a command is entered and the <enter> key is pressed, the shell puts word on the command line into special variables as follows :

$0 is the variable used to put command name.

$1 is the variable used to put the first argument.

$2 is the variable used to put the second argument.

And so on.

The UNIX shell creates variable up to $9. The variable $1 through $9 are also called positional parameters of the command line. Depending upon the number of arguments specified in the command, the shell will assign values to some or all of these variables.

Examples :

cat test.doc

$0 contains the command name i.e. cat

$1 contains the first argument i.e. test.doc

$2 does not contain anything (since there are no more arguments)

Apart from the variables $0 to $9, the shell also assigns values to the following variables when a command is entered :

$* contains the entire string of arguments.

$# contains the number of arguments specified in the command.

Examples :

cat testfil1 testfil2

$* it will contain the string of arguments "testfil1 testfil2"

$# it will contain the value of 2 since the number of arguments are 2.

1. Write a shell script which will take a filename as input and copy it.

2. Write a shell script which will accept the employee name and his basic as input and print his name, basic and HRA. HRA is based on the following information :

   If basic >2000 HRA = 550 else HRA = 350

3. Write a shell script that will take two filenames as input and remove one, if both are equal.
4. Write a shell script which will display the following menu :

   (*a*) Display who all are logged in.

   (*b*) Display long listing of files including hidden files page by page.

   (*c*) Display today's data and time.

   (*d*) Exit

   Depending upon the choice of the user, the appropriate actions should be performed. For choice 'a' to 'c' appropriate messages should be displayed before the desired output.

5. What are the different types of shell available in the UNIX system ?
6. What are shell metacharacters ? Explain by giving appropriate examples.
7. State and explain the system environmental variables of UNIX.
8. What are the different numeric, file and logical operators offered by UNIX.

Important Shell Script Considerations

From what we have seen so far, we can come to the conclusion that Shell is a very powerful tool which can be used by the programmer. We may face difficulties while executing Shell scripts. Here are some guidelines as to how we can effectively debug and execute our Shell script.

One of the important tools for debugging the script is the -v and -x options. We can trace the execution of the script step by step, looking at the substitutions made in the program.

While creating a complex Shell script, we should understand the basic problem and start off with a simple version. We can add extra features one by one. This way we will be able to identify the places where there might be errors.

We can try creating temporary files so that if we have a bug in our Shell script, we can look at the contents of that file and can trace the point where we have gone wrong.

The next way is to use echo statements strategically. For example, let us assume that our Shell script checks for a pattern and if it exists, then display the contents of the whole file. Here we can have an echo statement

as soon as the pattern is found and if it is not found also we can have an echo statement.

We can use the nonzero exit status for abnormally terminating programs. Thus we will know if our Shell script execution was successful or not.

Error checking at all points in a Shell script is very essential. For example, let us assume that we have a script which takes the name of the file as input, then checks if the file is readable and if it exists. Here we can have error statements- when the file does not exist and when it is not readable.

All error messages should be directed to the Standard Error. This way if there is any bug in the script, it is immediately displayed on the screen inviting us to rectify it.

Persistence is the final tool that is needed by us. We should try to debug the Shell script using all the above suggested methods. We must not give up very easily. Sometimes an error may be very obvious but we might have missed it.

Last but not the least, we must be able to work comfortably with the editors. Otherwise editing and creating of the script becomes a strenuous task.

EDITORS

We have discussed about various types of UNIX files in previous block. Text files are ordinary files that contain ASCII characters, thus are also called ASCII files. An ASCII (American Standard Code for Information and Interchange) is a standard format to represent all alphabets, numbers and other special symbols and is used to communicate data between different types of computers. How do we create an ASCII or a text file ? In UNIX, a text file is created and modified by a program, called Text Editor or simply Editor. You may have used the common DOS editors, EDLIN and EDIT. Similarly, UNIX operating system also comes with several editors. In this unit, we will discuss the features of the common editors along with their important commands.

Types of Editors

In general, we can classify editors into two types - Line Editors and Screen Editors.

(*a*) **Line Editors :** The early UNIX editors that edit processes one line at a time are called Line Editors. So with a line editor, you are required to give many commands to display or edit a set of lines. The common examples of UNIX line editors are ed and ex.

NOTES

(*i*) ed– ed was the first line editor of UNIX which is still used sometimes, though it is not popular. ed was popular on those days when most UNIX commands consisted of only two or three letters. It has became outdated now due to the use of screen editors that provide much more features.

(*ii*) ex– ex is more powerful and comprehensive than ed line editor. Some ex line-oriented commands are also used in few screen editors (such as vi) which we are discussing below.

(*b*) **Screen Editors :** Editors that make use of the whole screen for editing or processing more than one line at a time, are called screen editors. With screen editors, you can display and edit many lines by giving a single command. The common examples of screen editors are vi and emacs.

(*i*) vi –vi stands for 'Visual editor' vi is the standard full-screen UNIX tool and is the only editor available on SCO UNIX. As most of ed's and ex's features are also available in vi, you can straight away learn vi without learning ed or ex.

(*ii*) emacs –emacs is another popular screen editor of UNIX. It is distributed free of cost from Free Software Foundation, Cambridge. Although most vendors distribute emacs with UNIX system, emacs is not a part of UNIX.

In next part of this unit, we are discussing the commands of vi editor in detail. In the later part, we will summarise the important commands of emacs editor.

vi Editor

After comparative study of various types of editors, let's begin to learn the commands of vi editor. As vi editor is difficult to learn due too many commands, we advice you to keep on trying these commands while learning at your terminal.

Starting vi

There are many ways to load editor in the computer memory but the general syntax to start vi is –

$ vi [-option (s)] [Filename(s)]

where $ is the UNIX prompt

vi is the name of editor

option(s) is/are the letter(s) representing one or more options

File(s) is/are the name of text file(s) you want to open or create

[ ] means these entries are optional and may be skipped

NOTES

Let's see a few ways to load vi.

(*a*) Loading vi without a text file – If you want to load vi without specifying the name of text file to be opened or created, give the following command :

$ vi <Enter>

You will see now a blank screen as shown in Figure 7.2.

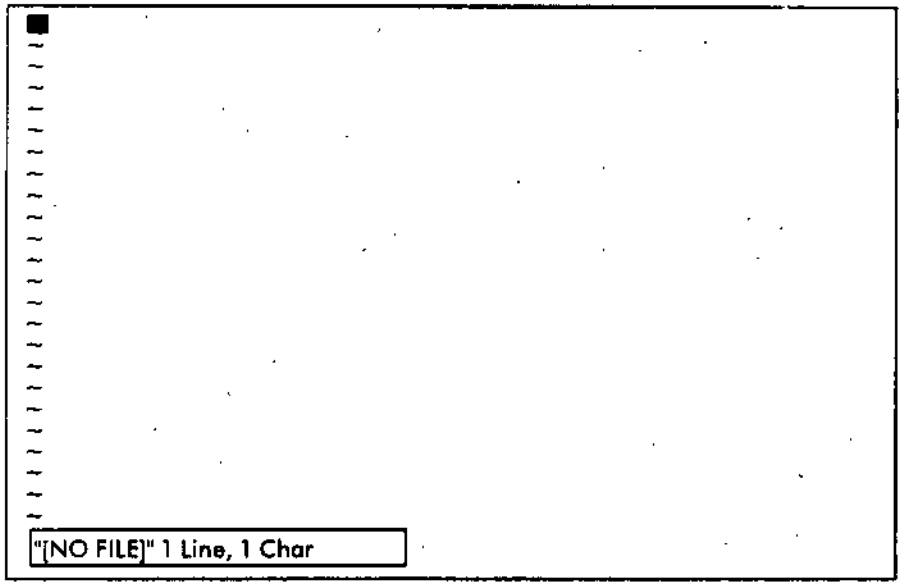


Figure 7.1. *The Blank Screen of vi Editor.*

The blank screen is actually not completely blank but is filled with a series of tilde(~) symbols running from upper left corner to down the screen. These tilde symbols show that the page is blank. At the bottom of screen is the status line showing that no file is opened and the cursor is on 1st line and 1st character position.

(*b*) Loading vi with text files – You can also specify the name of one or more text files which you want to create or open at the time of loading vi, by giving the following command :

$ vi mohit <Enter>

The above command opens the working copy of file 'mohit' in a memory area, called the editing buffer. If there would be no file with name 'mohit' in the current directory, a blank file with name 'mohit' will be stored in the editing buffer. Now, if you enter a new text or modify the existing text, that will be done in the editing buffer. So, your original file will remain unchanged unless you save it.

(*c*) Loading vi with text file and options – You can also specify three options while opening the text file as explained below.

(*i*) Sometimes the file may be crashed if you could not save the open file due to power failure or other interruption during editing of the file. You can recover the crashed file (say 'mohit') with option - r as shown below :

$ vi - r mohit <Enter>

(*ii*) To open a file for reading only and not for editing, give the option - r as shown below :

$ vi - R mohit <Enter>

Modes of Working in vi

You can work in vi in following two modes :

(*a*) **Command Mode :** When vi is started, it is in command mode which means that all your keystrokes are interpreted as commands. Most vi commands consist of one or two characters and do not require the <Enter> key. If you start entering text in command mode you will hear beep sounds indicating that you are making mistakes. Therefore, you must switch to the Insert mode for entering text in vi editor.

(*b*) **Insert Mode :** This mode is used to enter text in the editing buffer of the vi. Now, how will you switch to this mode from the command code ? You can do this by pressing i in the command more. Besides the lowercase i, you can also press some other keys for text entry and changing to insert mode as listed in the Table 7.2.

Table 7.2. Commands with their Functions

Commands	*Functions*
I	Inserts characters before the cursor.
I	Inserts characters at the beginning of the current line.
a	Appends characters after the current character.
A	Appends characters at the end of the current line.
c	Changes the following text
C	Changes the rest of the text on the line.
o	Inserts new line immediately before current line.
O	Inserts new line immediately before current line.
r	Replaces characters with a single character.
R	Replaces characters with characters until <ESC> key is pressed.
<Enter>	Inserts new line immediately following current character.

NOTES

Editing Text in vi

After coming to the insert mode, you can type your text. Type 4-5 lines of any text in the vi text editor screen by pressing <Enter> key after end of each line. When you want to end the text entry, press <ESC> key for returning to command mode.

Saving the Text

During entering of the text, always remember to save your file otherwise if power goes off, you will choose all the currently edited text. For saving the file for the first time, give the following :

keystroke commands at the command mode :

: w sweety

The text file will be saved with name 'sweety 'and you can still continue editing the text. If you don't want to save the file and want to quit vi give the following command :

:q!

There are many options for saving or abandoning a file in vi, which are listed in Table 7.3.

Table 7.3. Various options for saving/abandoning file in vi editor

Commands	*Functions*
: W(Filename)	Saves the text file with the new name, (Filename) and resumes editing.
: w	Saves the file with existing name and resumes editing.
:q!	Quits vi editor without saving the file.
:Q	Quits vi editor only when file is saved.
:x	Saves the file and quits vi editor.
:wq	Same as :x
Zz	Same as :x

NOTE: In vi, you can also use ex commands. When a colon (:) is followed by the command it indicates that it is an ex command. For example, : w and :q are ex command. You can also switch to ex line editor by typing at the vi command prompt. Similarly, you can switch back to vi from ex, by typing vi at the ex command prompt. As ex command prompt is displayed as colon (:), you need not have to type : again before typing vi.

Moving the Cursor During Editing

The cursor, you know is a blinking underscore character that marks a position on the screen, where user can enter or modify the text. In vi, the line on which cursor is positioned is called the current line. In vi, you can move cursor in both command and insert modes. Some useful cursor commands of is editor are summarised in Table 7.4.

Table 7.4. Some useful cursor commands of vi

Commands	*Functions*
(left arrow)	Moves the cursor to the beginning of current line.
(Right arrow)	Moves the cursor to the end of Current line.
(Up arrow)	Move the cursor to left character on same line.
(Down arrow)	Move the cursor to right character on same line.
O	Same as left arrow key.
$	Same as right arrow key.
_	Moves cursor to the beginning of the previous line.
<Ctrl>M	Moves cursor to the beginning of the next line.
b	Moves cursor to the beginning of the previous word.
e	Moves cursor to the end of the next word
w	Moves cursor to the beginning of the next word.
G	Moves cursor to the last line of the file.
nG	Moves cursor to the beginning of line n (*e.g.*, 5G moves to 5th line)
n|	Moves cursor to the beginning of column n (*e.g.*, 5moves to 5th column on same line)
H	Moves cursor to top line of the screen.
M	Moves cursor to middle of the screen.
L	Moves cursor to bottom of the screen
B	Same as b, but ignores punctuation in text.
E	Same as e, but ignores punctuation in text.
w	Same as w, but ignores punctuation in text.

Moving the Screen

Beside the cursor, you can also move the screen in vi. You can scroll the screen upward or downward by the commands listed in Table 7.5.

NOTES

Table 7.5. Screen movement command of vi

Commands	*Functions*
<Ctrl> U	Scrolls half of the screen upwards.
<Ctrl> D	Scrolls half of the screen downwards
<Ctrl> B	Scrolls the full screen backward (Up)
<Ctrl> F	Scrolls the full screen forward (down)

Modifying the Text

Modifying the text in vi is a difficult process because you cannot overwrite the characters. If you want to change a character of the word, then either you first delete that character and write new one or use a command to replace that character. For example, if your text contains a word 'Unix' and you want to write u in capital letter, then you cannot straight away position the cursor to 'u' and type 'U'. This is not allowed in vi editor. For changing 'unix' word with 'Unix', you must follow one of the methods given below :

Method I : Deleting and changing the character

Step 1 : Move the cursor to character 'u' of the word 'unix'.

Step 2 : Press x in the command mode. This will delete the character 'U'

Step 3 : Press i for switching to the insert mode.

Step 4 : Enter U at the cursor position to *i.e.,* before the word 'nix'.

Method II : Replacing the character

Step 1 : Same as step 1 of method I

Step 2 : Press r in the command mode for replacing the character.

Step 3 : Enter U immediately after pressing r.

Method III : Changing the case of letter

Step 1 : Same as step 1 of method I

Step 2 : Press tilde (~) character in the command mode. This will change lowercase 'u' to uppercase 'U'. The tilde (~) can also be used to change a uppercase letter to lowercase.

Any one of the above method will change 'unix' word to 'Unix'. So, you must have realised that editing is not so easy in vi as in a word processor. Various useful commands for deleting and undeleting the text are listed in Table 7.6.

Table 7.6. Important commands for deleting and undeleting the text

Commands	*Functions*
x	Deletes a character at the cursor position
d	Deletes the next character
D	Deletes characters to the end of the line
dd	Deletes the current line
: D	Same as dd
: x	Same as x
:X	deletes the previous character
: D$	Same as D
: U	Undoes the last deletion
p	Puts the last deleted text after the cursor
P	Puts the last deleted text before the cursor
u	Undoes the last change made to the text
U	Restores the line to its previous condition

Cutting the text

While editing a text file, we often require to move a block of text from one position to another within a file. In order to move the text, first we cut the text, then place the cursor at the position where the text has to be placed and finally we paste it. There are many commands to cut (or yank) the text in vi editor, which are called Yanking commands and some of them are listed in Table 7.7.

NOTES

Table 7.7. Important commands for yanking (cutting) the text

Commands	*Functions*
y	Cuts the characters at cursor position
yy	Cuts the current line
Y	Same as yy
yn	Cuts the n number of characters from cursor position
nyy	Cuts the n number of lines from current line
y$	Cuts from cursor position to end of line
yw	Cuts the word at cursor position

Searching a Text Pattern

At times, you are required to search a particular word or group of words in the text file. For example, in order to search a word 'love' in the text file, issue the following command at the command prompt :

/love

The above command will position the cursor at the first occurrence of the word 'love' from the current line. Suppose, you want to find the next occurrence of the word 'love', then simply press / at the command prompt. If there is no more same pattern, you will see the following message :

pattern not found

You can also search backward by giving the following command.

?love

Similarly for continue searching, you can press only ?. Some of the important options with same pattern for searching a text pattern are listed in Table 7.8.

Table 7.8. Options for searching a text-pattern

Commands	*Functions*
/pattern	Search forward a string 'pattern' in the text file
/	Search forward for the previous pattern
?pattern	Search backward a string 'pattern' in the text file
?	Search backward for the previous pattern
n	Repeat last / or ? Command and search forward
N	Repeat last / or ? command and search backward

Repeating vi Commands

You can execute vi commands repeatedly by adding a number to the command. This number indicates the number of times a command to be repeated. You can use this method to repeat commands with any of the text-manipulation commands (such as deletion, cursor movements, cutting the text etc.) discussed in this unit.

NOTE : You can execute

Examples

(*i*) To delete five characters from the cursor position, give the following command :

: X5

(*ii*) To delete three lines from the current line, give the following command :

: D5

(*iii*) To move the cursor to the beginning of next three words, give the following command :

W3

emacs EDITOR

Besides vi editor, emacs is also a popular text editor. Although the earlier versions of emacs were difficult to learn, the latest one is easier due to a user-friendly interface. This version of emacs supports a mouse and provides pull-down menus. You can also issue commands from the keyboard by using <Ctrl> and <Alt> keys.

Starting emacs Editor

To start emacs, enter the following command :

$ emacs

You will see a startup screen followed by almost a blank screen with a status line at the top/bottom. As emacs work in only single mode, you can straight away keep on entering the text. You are also required to press <Enter> key after each line as you do in Vi editor. Type 3-4 lines of text and save the file as described below.

Saving the file

In emacs, you can save the file with one of the following ways :

(*a*) If you want to resume editing after saving the file, press <Ctrl> X followed by <Ctrl> S key.

(*b*) If you want to quit emacs after saving the file, press <Ctrl> X followed by <Ctrl> C key.

In either of the above command, emacs will ask you to enter the name of the file, if you will be saving the file for the first time.

NOTES

Important Commands

Though we are not discussing emacs in detail, we are summarising the frequently used commands of emacs in Table 7.9.

Table 7.9. Some frequently used commands of emacs

Commands	*Functions*
Alt <	Moves the cursor to the beginning of file
Alt >	Moves the cursor to the end of file
Ctrl V	Moves the cursor to the next screen
Alt V	Moves the cursor to the previous screen
Ctrl D	Deletes a character at the cursor position
Del	Deletes all characters from beginning of line to the cursor position
Ctrl K	Deletes all characters from the cursors position to the end of line.
Ctrl @	To mark the beginning/end of a block for copying and moving the text
Ctrl w	To delete the marked block of text
Ctrl Y	To paste the marked block of text

Restricted and Special Versions of Editors

Each of the editors has some restricted and special versions that offers a limited range of features specially for beginners as listed in Table 7.10.

Table 7.10. Restricted and special version of editors

Version	*Description*
red	Restricted version of ed
edit	Restricted version of ex
vedit	Restricted version of vi
view	Special version of vi for read only mode

Input/Output Redirection Pipes and I/O Error Detection

UNIX is known for having many powerful features of to control input and output. In UNIX system, every program and UNIX commands are connected to three files – the standard input, the standard output and the standard error.

Standard Input

The basic source from which UNIX programs read their input is called the standard input. By default, the standard input is assigned to the special file representing the keyboard which is the actual source of the input data.

Standard Output

The basic target to which UNIX programs write their output is called the standard output. By default, the standard output is assigned to the special file representing the screen of your terminal which is the usual target of the output.

Standard Error

The output target to which UNIX programs write their error messages is called the standard error. By default, the standard error is also assigned to special file representing the screen of your terminal.

Redirection

Although most of the time, standard input, standard output and standard error are accepted to the user, you can also change them. You can change the input source or the output target. For instance, sometimes you may require to get input from the file instead of the keyboard or you may want output on printer instead of the monitor. In such circumstances, you need to redirect the standard input, standard output or standard error. Redirection is a technique in UNIX to reassign the standard input, standard output or standard error to a specified file.

NOTES

Redirecting the Standard Input

You can redirect the standard input for a command by writing the command in following way :

- First type the command;
- Put < (less than) symbol; and
- Then write the name of the input file.

Examples :

(*i*) You can take the input from the file 'account', instead of the standard input (keyboard) by the following command :

```
$ cat < account
```

(*ii*) To edit the file 'komal' by taking the input from file ex.script and not from keyboard give the following command :.

```
ex komal <ex.script>
```

Redirecting the Standard Output

You can also redirect the standard output for a command by writing the command in following way.

- First type the command.
- Put > (greater than) symbol and
- Then write the name of the output file.

Examples :

(*i*) To redirect the contents of the file 'preeti.c' to the file 'vandana.c' instead of the screen, give the following command :

```
$ cat preeti.c > vandana.c
```

(*ii*) To redirect the listing of ls –1 command to a file, 'sachin' instead of the screen, give the following command :

```
$ ls –1 > sachin
```

(*iii*) You can redirect the listing of ls –1 command on other terminal (say tty 02), give the following command :

```
$ ls -1 > /dev/tty02
```

Redirecting the output in append mode

If you redirect the output to a file and if the file already exists, it is over written by new data. So, in that case, you can redirect the output to already existing file in append mode by using >> instead of >.

Examples

(*i*) To redirect the output of the ls –1, command to the already existing file 'sachin', give the following command :

NOTES

$ ls –1 >> sachin

If the file 'sachin' does not exist, the shell will create this file otherwise new output will be appended at the end of this file.

(*ii*) *Using >> symbol, you can accumulate data into a file by running your program again and again.* To accumulate the reports on a file 'invlist' by running the program 'invoice' repeatedly, give the following command :

$ invoice >> invlist

Redirecting the standard error

Error messages are normally displayed on the screen, but you can also redirect them to a file or printer by using 2> as explained in following example :

(*i*) To execute a program 'invoice' by sending the output to file 'invlist' and the error messages to another file 'errors', give the following command :

$ invoice > invlist 2> errors

(*ii*) If you want to redirect both the standard output and the standard error to a file 'invrepo', use 2>&1 symbol as shown in following command :

$ invoice > invrepo 2>&1

Redirecting the error in append mode

As you do in standard output, you can use >> symbol for appending the standard error to a file as illustrated in following commands :

$ invoice >> invlist 2>> errors

$ *invoice << invrepo 2>>&1*

Redirecting both input and output

You can redirect the input and output at the same time. For example, to obtain input to cat command from the file 'infile' and to redirect its output to another file 'outfile', give the following command :

$ cat <infile> outfile

Introducing the ex Editor

ex is not really another editor. *vi* is the visual mode of the more general, underlying line editor, *ex*. Some *ex* commands can be useful to you while you are working in *vi*, for they can save you a lot of editing time. Most of these *commands can be used without ever leaving vi.*

You already know how to think of files as a sequence of numbered lines. *ex* simply gives you editing commands with greater mobility and scope. With

ex you can move easily between files and transfer text from one file to another in a variety of ways. You can quickly edit blocks of text larger than a single screen. And with global replacement you can make substitutions throughout a file for a given pattern.

NOTES

- Move around a file by using line numbers.
- Use *ex* commands to copy, move, and delete blocks of text.
- Save files and parts of files.
- Work with multiple files (reading in text or commands, traveling between files).

ex Commands

Long before *vi* or any other screen editor was invented, people communicated with computers on printing terminals, rather than on today's CRTs. Line numbers were a way to quickly identify a part of a file to be worked on, and line editors evolved to edit those files. A programmer or other computer user would typically print out a line (or lines) on the printing terminal, give the editing commands to change just that line, then reprint to check the edited line.

People rarely edit files on printing terminals any more, but some *ex* line editor commands are still useful to users of the more sophisticated visual editor built on top of *ex*. Although it is simpler to make most edits with *vi*, the line orientation of *ex* gives it an advantage when you want to make large-scale changes to more than one part of a file.

Before you start off simply memorizing *ex* commands (or worse, ignoring them), let's first take some of the mystery out of line editors. Seeing how *ex* works when it is invoked directly will help make sense of the sometimes obscure command syntax.

Open a file that is familiar to you and try a few *ex* commands. Just as you can invoke the *vi* editor on a file, you can invoke the *ex* line editor on a file. If you invoke *ex*, you will see a message about the total number of lines in the file, and a colon command prompt.

For example :

```
$ ex practice
"practice" 6 lines, 320 characters
:
```

You won't see any lines in the file unless you give an *ex* command that causes one or more lines to be displayed.

ex commands consist of a line address (which can simply be a line number) plus a command; they are finished with a carriage return. One of the most

NOTES

basic commands is p for print (to the screen). So, for example, if you type 1p at the prompt, you will see the first line of the file :

```
:1p
With a screen editor you can
:
```

In fact, you can leave off the p, because a line number by itself is equivalent to a print command for that line. To print more than one line, you can specify a range of line numbers (for example, 1,3 - two numbers separated by commas, with or without spaces in between). For example :

```
:1,3
With a screen editor you can
scroll the page, move the cursor,
delete lines, insert characters, and more,
```

A command without a line number is assumed to affect the current line. So, for example, the substitute command (s), which allows you to substitute one word for another, could be entered like this :

```
:1
With a screen editor you can
:s/screen/line/
With a line editor you can
```

Notice that the changed line is reprinted after the command is issued. You could also make the same change like this :

```
:1s/screen/line/
With a line editor you can
```

Even though you will be invoking *ex* commands from *vi* and will not be using them directly, it is worthwhile to spend a few minutes in *ex* itself. You will get a feel for how you need to tell the editor which line (or lines) to work on, as well as which command to execute.

After you have given a few *ex* commands on your *practice* file, you should invoke *vi* on that same file, so that you can see it in the more familiar visual mode. The command :vi will get you from *ex* to *vi*.

To invoke an *ex* command from *vi*, you must type the special bottom line character : (colon). Then type the command and press [RETURN] to execute it. So, for example, in the *ex* editor you move to a line simply by typing the number of the line at the colon prompt. To move to line 6 of a file using this command from within *vi*, enter :

```
:6
```

Press [RETURN].

Following the exercise, we will be discussing *ex* commands only as they are executed from *vi*.

Exercise : The *ex* Editor

NOTES

At the UNIX prompt, invoke ex editor on a practice file	`ex practice`
A message appears	`"practice" 6 lines 320 characters`
Go to and print (display) first line	`:1`
Print (display) lines 1 through 3	`:1,3`
Substitute screen for line on line 1	`:1s/screen/line`
Invoke vi editor on file	`:vi`
Go to first line	`:1`

Editing with ex

Many *ex* commands that perform normal editing operations have an equivalent in *vi* that does the job more simply. Obviously, you will use dw or dd to delete a single word or line rather than using the delete command in *ex*. However, when you want to make changes that affect numerous lines, you will find the *ex* commands more useful. They allow you to modify large blocks of text with a single command.

These *ex* commands are listed below, along with abbreviations for those commands. Remember that in *vi* each *ex* command must be preceded with a colon. You can use the full command name or the abbreviation, whichever is easier to remember.

`delete`	`d`	Delete lines.
`move`	`m`	Move lines.
`copy`	`co`	Copy lines.
	`t`	Copy lines (a synonym for co).

You can separate the different elements of an *ex* command with spaces, if you find the command easier to read that way. For example, you can separate line addresses, patterns, and commands in this way. You cannot, however, use a space as a separator inside a pattern or at the end of a substitute command.

Line Addresses

For each *ex* editing command, you have to tell *ex* which line number(s) to edit. And for the *ex* move and copy commands, you also need to tell *ex* where to move or copy the text to.

You can specify line addresses in several ways :

- With explicit line numbers.

NOTES

- With symbols that help you to specify line numbers relative to your current position in the file.
- With search patterns as *addresses* that identify the lines to be affected.

Let's look at some examples.

Defining a Range of Lines

You can use line numbers to define explicitly a line or range of lines. Addresses that use explicit numbers are called *absolute* line addresses. For example :

`:3,18d`	Delete lines 3 through 18.
`:160,224m23`	Move lines 160 through 244 to follow line 23. (Like `delete` and `put` in *vi*.)
`:23,29co100`	Copy lines 23 through 29 and put after line 100. (Like `yank` and `put` in *vi*.)

To make editing with line numbers easier, you can also display all line numbers on the left of the screen. The command :

```
:set number
```

or its abbreviation :

```
:set nu
```

displays line numbers. The file *practice* then appears :

```
1 With a screen editor
2 you can scroll the page,
3 move the cursor, delete lines,
4 insert characters and more
```

The displayed line numbers are not saved when you write a file, and they do not print if you print the file. Line numbers are displayed either until you quit the *vi* session or until you disable the *set* option :

```
:set nonumber
```

or :

```
:set nonu
```

To temporarily display the line numbers for a set of lines, you can use the # sign. For example :

```
:1,10#
```

would display the line numbers from line one to line ten.

You can also use the [CTRL-G] command to display the current line number. You can thus identify the line numbers corresponding to the start and end of a block of text by moving to the start of the block, typing [CTRL-G] then moving to the end of the block and typing [CTRL-G] again.

Yet another way to identify line numbers is with the *ex* = command :

```
:=
```

Print the total number of lines.

```
:.=
```

Print the line number of the current line.

```
:/pattern/=
```

Print the line number of the first line that matches *pattern*.

NOTES

Line Addressing Symbols

You can also use symbols for line addresses. A dot (.) stands for the current line; $ stands for the last line of the file. % stands for every line in the file; it's the same as the combination 1,$. These symbols can also be combined with absolute line addresses. For example :

```
:.,$d
```

Delete from current line to end of file.

```
:20,.m$
```

Move from line 20 through the current line to the end of the file.

```
:%d
```

Delete all the lines in a file.

```
:%t$
```

Copy all lines and place them at the end of the file (making a consecutive duplicate).

In addition to an absolute line address, you can specify an address relative to the current line. The symbols + and - work like arithmetic operators. When placed before a number, these symbols add or subtract the value that follow. For example :

```
:.,.+20d
```

Delete from current line through the next 20 lines.

```
:226,$m.-2
```

Move lines 226 through the end of the file to two lines above the current line.

```
:.,+20#
```

Display line numbers from the current line to 20 lines further on in the file.

In fact, you don't need to type the dot (.) when you use + or -, because the current line is the assumed starting position.

Without a number following them, + and - are equivalent to +1 and -1, respectively. [1] Similarly, ++ and - - each extend the range by an additional line, and so on. The + and - can also be used with search patterns, as shown in the next section.

NOTES

[1] In a relative address, you shouldn't separate the plus or minus symbol from the number that follows it. For example, +10 means "10 lines following," but + 10 means "11 lines following (1 + 10)," which is probably not what you mean.

The number 0 stands for the top of the file (imaginary line 0). 0 is equivalent to 1-, and both allow you to move or copy lines to the very start of a file, before the first line of existing text. For example :

```
:-,+t0
```

Copy three lines (the line above the cursor through the line below the cursor) and put them at the top of the file.

Search Patterns

Another way that *ex* can address lines is by using search patterns. For example :

```
:/pattern/d
```

Delete the next line containing *pattern*.

```
:/pattern/+d
```

Delete the line *below* the next line containing *pattern*. (You could also use +1 instead of + alone.)

```
:/pattern1/,/pattern2/d
```

Delete from the first line containing *pattern1* through the first line containing *pattern2*.

```
:.,/pattern/m23
```

Take text from current line (.) through the first line containing *pattern* and put after line 23.

Note that patterns are delimited by a slash both *before* and *after*.

If you make deletions by pattern with *vi* and *ex*, there is a difference in the way the two editors operate. Suppose you have in your file *practice* the lines :

```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters and more, while seeing results
of your edits as you make them.
```

Keystrokes	Results
`d/while`	`With a screen editor you can scroll the page, move the cursor, while seeing results of your edits as you make them.`

The *vi* delete to *pattern* command deletes from the cursor up to the word *while* but leaves the remainder of both lines.

NOTES

```
:../while/d     With a screen editor you can scroll the
                of your edits as you make them.
```

The *ex* command deletes the entire range of addressed lines; in this case both the current line and the line containing the pattern. All lines are deleted in their entirety.

Redefining the Current Line Position

Sometimes, using a relative line address in a command can give you unexpected results. For example, suppose the cursor is on line 1, and you want to print line 100 plus the five lines below it. If you type :

```
:100,+5 p
```

you'll get an error message saying, "First address exceeds second." The reason the command fails is that the second address is calculated relative to the current cursor position (line 1), so your command is really saying this :

```
:100,6 p
```

What you need is some way to tell the command to think of line 100 as the "current line," even though the cursor is on line 1.

ex provides such a way. When you use a semicolon instead of a comma, the first line address is recalculated as the current line. For example, the command :

```
:100;+5 p
```

prints the desired lines. The +5 is now calculated relative to line 100. A semicolon is useful with search patterns as well as absolute addresses. For example, to print the next line containing *pattern*, plus the 10 lines that follow it, enter the command :

```
:/pattern/;+10 p
```

Global Searches

You already know how to use / (slash) in *vi* to search for patterns of characters in your files. *ex* has a global command, g, that lets you search for a pattern and display all lines containing the pattern when it finds them. The command :g! does the opposite of :g. Use :g! (or its synonym :v) to search for all lines that do *not* contain *pattern*.

You can use the global command on all lines in the file, or you can use line addresses to limit a global search to specified lines or to a range of lines.

```
:g/pattern
```

Finds (moves to) the last occurrence of *pattern* in the file.

```
:g/pattern/p
```

Finds and displays all lines in the file containing *pattern*.

NOTES

`:g!/pattern/nu`

Finds and displays all lines in the file that don't contain *pattern*; also displays line number for each line found.

`:60,124g/pattern/p`

Finds and displays any lines between lines 60 and 124 containing *pattern*.

Combining ex Commands

You don't always need to type a colon to begin a new *ex* command. In *ex*, the vertical bar (|) is a command separator, allowing you to combine multiple commands from the same *ex* prompt (in much the same way that a semicolon separates multiple commands at the UNIX shell prompt). When you use the |, keep track of the line addresses you specify. If one command affects the order of lines in the file, the next command does its work using the new line positions. For example :

```
:1,3d | s/thier/their/
```

Delete lines 1 through 3 (leaving you now on the top line of the file); then make a substitution on the current line (which was line 4 before you invoked the *ex* prompt).

`:1,5 m 10 | g/pattern/nu`

Move lines 1 through 5 after line 10, and then display all lines (with numbers) containing *pattern*.

Saving and Exiting Files

You have learned the *vi* command ZZ to quit and write (save) your file. But you will frequently want to exit a file using *ex* commands, because these commands give you greater control. We've already mentioned some of these commands in passing. Now let's take a more formal look.

`:w`

Writes (saves) the buffer to the file but does not exit. You can (and should) use :w throughout your editing session to protect your edits against system failure or a major editing error.

`:q`

Quits the file (and returns to the UNIX prompt).

`:wq`

Both writes and quits the file. Both writes and quits (exits) the file. It's the same as :wq.

vi protects existing files and your edits in the buffer. For example, if you want to write your buffer to an existing file, *vi* gives you a warning. Likewise, if you have invoked *vi* on a file, made edits, and want to quit *without* saving the edits, *vi* gives you an error message such as :

```
No write since last change.
```

These warnings can prevent costly mistakes, but sometimes you want to proceed with the command anyway. An exclamation point (!) after your command overrides the warning :

NOTES

```
:w!
:q!
```

w! can also be used to save edits in a file that was opened in read-only mode with vi -R or view.

:q! is an essential editing command that allows you to quit without affecting the original file, regardless of any changes you made in this session. The contents of the buffer are discarded.

Renaming the Buffer

You can also use :w to save the entire buffer (the copy of the file you are editing) under a new filename.

Suppose you have a file *practice*, containing 600 lines. You open the file and make extensive edits. You want to quit but save *both* the old version of *practice* and your new edits for comparison. To save the edited buffer in a file called *practice.new*, give the command :

```
:w practice.new
```

Your old version, in the file *practice*, remains unchanged (provided that you didn't previously use :w). You can now quit the old version by typing :q.

Saving Part of a File

While editing, you will sometimes want to save just part of your file as a separate, new file. For example, you might have entered formatting codes and text that you want to use as a header for several files.

You can combine *ex* line addressing with the write command, w, to save part of a file. For example, if you are in the file *practice* and want to save part of practice as the file *newfile*, you could enter :

```
:230,$w newfile
```

Saves from line 230 to end of file in *newfile*.

```
:.,600w newfile
```

Saves from the current line to line 600 in *newfile*.

Appending to a Saved File

You can use the UNIX redirect and append operator (>>) with w to append all or part of the contents of the buffer to an existing file. For example, if you entered :

```
:1,10w newfile
```

then :

```
:340,$w >>newfile
```

newfile would contain lines 1–10 and from line 340 to the end of the buffer.

NOTES

Copying a File into Another File

Sometimes you want to copy text or data already entered on the system into the file you are editing. In *vi* you can read in the contents of another file with the *ex* command :

```
:read filename
```

or its abbreviation :

```
:r filename
```

This command inserts the contents of *filename* starting on the line after the cursor position in the file. If you want to specify a line other than the one the cursor's on, simply type the line number (or other line address) you want before the read or r command.

Let's suppose you are editing the file *practice* and want to read in a file called *data* from another directory called */usr/tim*. Position the cursor one line above the line where you want the new data inserted, and enter :

```
:r /usr/tim/data
```

The entire contents of */usr/tim/data* are read into *practice*, beginning below the line with the cursor.

To read in the same file and place it after line 185, you would enter :

```
:185r /usr/tim/data
```

Here are other ways to read in a file :

```
:$r /usr/tim/data
```

Place the read-in file at the end of the current file.

```
:0r /usr/tim/data
```

Place the read-in file at the very beginning of the current file.

```
:/pattern/r /usr/tim/data
```

Place the read-in file in the current file, after the line containing *pattern*.

Editing Multiple Files

ex commands enable you to switch between multiple files. The advantage to editing multiple files is speed. When you are sharing the system with other users, it takes time to exit and reenter *vi* for each file you want to edit. Staying in the same editing session and traveling between files is not only faster for access.

Invoking vi on Multiple Files one

When you first invoke *vi*, you can name more than one file to edit, and then use *ex* commands to travel between the files.

invokes *file1* first. After you have finished editing the first file, the *ex* command :w writes (saves) *file1* and :n calls in the next file (*file2*).

Suppose you want to edit two files, *practice* and *note*.

NOTES

Keystrokes	Results
`vi practice`	`With a screen editor you can scroll`
`note`	`the page, move the cursor, delete lines,`
	`insert characters, and more, while seeing`
	Open the two files *practice* and *note*. The first-named file, *practice*, appears on your screen. Perform any edits.
`:w`	`"practice" 6 lines 328 characters`
	Save the edited file *practice* with the *ex* command w. Press RETURN.
`:n`	`Dear Mr.`
	`Henshaw :`
	`Thank you for the prompt...`
	Call in the next file, *note*, with the *ex* command n. Press RETURN. Perform any edits.
`:x`	`"note" 23 lines 1343 characters`
	Save the second file, *note,* and quit the editing session.

Calling In New Files

You don't have to call in multiple files at the beginning of your editing session. You can switch to another file at any time with the *ex* command :e. If you want to edit another file within *vi*, you first need to save your current file (:w), then give the command :

```
:e filename
```

Suppose you are editing the file *practice* and want to edit the file *letter*, then return to *practice*.

Keystrokes	Results
`:w`	`"practice" 6 lines 328 characters`
	Save *practice* with w and press RETURN. *practice* is saved and remains on the screen. You can now switch to another file, because your edits are saved.
`:e letter`	`"letter" 23 lines 1344 characters`
	Call in the file *letter* with e and press RETURN. Perform any edits.

vi "remembers" two filenames at a time as the current and alternate filenames. These can be referred to by the symbols % (current filename) and

NOTES

(alternate filename). # is particularly useful with :e, since it allows you to switch easily back and forth between two files. In the example given just above, you could return to the first file, *practice*, by typing the command :e#. You could also read the file *practice* into the current file by typing :r#.

If you have not first saved the current file, *vi* will not allow you to switch files with :e or :n unless you tell it imperatively to do so by adding an exclamation point after the command.

For example, if after making some edits to *letter*, you wanted to discard the edits and return to *practice*, you could type :e!#.

The command :

```
:e!
```

is also useful. It discards your edits and returns to the last saved version of the current file.

In contrast to the # symbol, % is useful mainly when writing out the contents of the current buffer to a new file. For example, a few pages earlier, in the section "Renaming the Buffer," we showed how to save a second version of the file *practice* with the command :

```
:w practice.new
```

Since % stands for the current filename, the previous line could also have been typed :

```
:w %.new
```

Edits Between Files

When you give a yank buffer a one-letter name, you have a convenient way to move text from one file to another. Named buffers are not cleared when a new file is loaded into the *vi* buffer with the :e command. Thus, by yanking or deleting text from one file (into multiple named buffers if necessary), calling in a new file with :e, and putting the named buffer into the new file, you can transfer material between files.

The following example illustrates how to transfer text from one file to another.

Keystrokes	Results
"f4yy	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of the edits as you make them
	Yank four lines into buffer f.
:w	"practice" 6 lines 238 characters
	Save the file.
:e letter	Dear Mr. Henshaw :

```
I thought that you would
be interested to know that :
Yours truly,
```

Enter the file *letter* with :e. Move cursor to where the copied text will be placed.

```
"fp     Dear Mr.
Henshaw :
I thought that you would
be interested to know that :
With a screen editor you can scroll
the page, move the cursor, delete lines,
insert characters, and more, while seeing
the results of the edits as you make them
Yours truly,
```

Place yanked text from named buffer f below the cursor.

Another way to move text from one file to another is to use the *ex* commands :ya (yank) and :pu (put). These commands work the same way as the equivalent *vi* commands y and p, but they are used with *ex*'s line addressing capability and named buffers. For example :

```
:160,224ya a
```

would yank (copy) lines 160 through 224 into buffer a. Next you would move with :e to the file where you want to put these lines. Place the cursor on the line where you want to put the yanked lines. Then type :

```
:pu a
```

to put the contents of buffer a after the current line.

1. Some of the commands from the early editors like ed can be used with later, more sophisticated editors. If you should forget how to use one of the screen editors to perform some task, you can use the ed command sequence.
2. String searches use the same constructs in the majority of the UNIX editors. In addition, global commands may be difficult to perform in some editors, but are easy in the early editors like ed and ex.
3. The syntax you learn with ed is also used in other UNIX tools like grep and diff. Learning ed will enable you to use these other tools with less difficulty.
4. After learning the tools available in ed, other powerful file editing tools like sed are easy and natural to use to edit whole files.
5. If you are using a hard copy terminal, or if your stty file becomes corrupted and you don't have access to a screen editor you can still perform edits using line editors like ed and ex.

NOTES

NOTES

6. It is often easier to direct someone to use a line editor like ed, rather than have them start a screen editor like vi. The ed or ex editor is often the choice of telephone tech support because it is less complex to walk users through its use.

History of ed

The early versions of the **ed** editor, also known as the "standard Unix editor," were written by Ken Thompson. It is descended from the QED editor which was written by Butler Lampson and Peter Deutsch at Berkeley, in the middle 1960s. Before he wrote **ed**, Thompson wrote a version of QED for the Multics project. Dennis Ritchie also worked on that early, Bell Labs version, of QED. However, Thompson wanted a simpler editor so he built the first version of **ed**. In its evolution, many other people have worked on **ed**, and it has become a more complex and more powerful editor. Nonetheless, **ed** has retained much of the original flavor and functionality given it by Thompson in the late '60s.

Line editors

One of the interesting things about both of the standard Unix editors, (**ed** and **ex/vi**) is that they are "line editors" rather than page editors or document editors. This sometimes takes a little getting used to, as most of us are more familiar with the modern, full document, or word processors. To understand **ed** we need to consider the time at which **ed** was developed, and technology available at the time.

Remember that **ed** was developed back in the late 1960's and the "state-of-the-art" input device back then was the teletype, typified by the ASR-33 which first saw the light of day in 1968. This absolutely classic, all-around I/O Device was able to print a blazing, (for its time), 10 characters or 80 bits per second (upper case and symbols only), generating input at the same speed from its keyboard. It also had a built in punched paper tape reader, and could punch paper tape for output and off-line storage. Some models of this unit could even start and stop the paper tape reader or the paper tape punch on command from a host computer which is why it was given the "ASR" (Automatic Send & Receive) designation in it's name.

With a very slow I/O device long involved command names are very inefficient. That is why most of the **ed** commands are only one or two characters long. In addition to being slow the teletype or ASR 33 was a hard copy device. That meant that the user could look back on the paper and see what they had typed, unlike a monitor where you can only see 24 lines. For that reason there was no need to "refresh the screen" because all that would do would be print a lot of lines, pushing the paper up at 10 characters per second which would take a lot of time. The **ed** editor was designed to work on a single line of the input file at a time. Hence the appellation "line editor".

NOTES

As **ed** grew and developed over time, he acquired the ability to perform edits on a number of lines at the same time. He has also acquired new, additional, features that made him more powerful and more useful.

All the descendents of ed are forms of a line editor. A line editor is one which enables the user to work with a single line of text at a time. Once ed has at least one line of text in the working buffer, there is always a current line that is the default target for any edit.

When you open a file in ed, or when you start typing a new file, ed creates a working buffer, called the ***editing buffer***. This buffer holds the information in memory. Any changes you make will not be applied to the disk file until you execute a "write to disk" command. This means you can back out of changes easily, but it also means that you can lose a whole session's work if you forget to write the contents of the editing buffer back to disk before you quit ed. Fortunately, if you allow it, ed will remind you to save the contents of the editing buffer before you exit.

Starting ed

It is simple to invoke the ed editor. All you need to do is type either :

ed or **ed filename** at the system prompt to invoke the ed editor.

Figure 7.2 on the opposing page, shows how to invoke the ed editor without specifying a file name.

Figure 7.3 shows how to start ed and give it a file to edit. Note : ed will tell you how many characters or bytes are in an existing file as it reads them into the editing buffer.

Please note : For purposes of this text, the standard C shell prompt % is used to show the operating system prompt. When you see a %, that indicates a command to be entered on the command line.

In both examples, the first command **clear** simply clears the screen before ed is started. This makes the interaction with ed less cluttered and helps identify the ed command. I will eliminate the **clear** command from subsequent lines to reduce the clutter. However, I recommend you always issue the clear command before you start ed.

```
% clear <return>
% ed <return>
_  <— cursor
```

Figure 7.2

```
% clear <return>
% ed nifty_stuff <return>
207
_  <— cursor
```

Figure 7.3

The two modes of ed

The UNIX editors work in one of two modes :

(*a*) command mode

(*b*) text entry mode

Many new users find it somewhat confusing determining which mode ed is in (as well as switching back and forth between modes). If ed is in text entry mode, then all of the ed commands you enter will be taken as text input and inserted into the document you are editing. On the other hand, if you are in command mode, ed will try to interpret anything you type as if it were commands, (and possibility become very confused). One of the most common problems new users have with this is being in text entry mode and trying to use ed commands. Those commands will appear one after the other, but as ed is in text entry mode, they will not be executed. This can be both confusing and frustrating.

To help you remember which mode you are in, ed has a command that asks it to tell you when it is in command mode. Figure 7.4 shows the use of the P (or give me a special Prompt) command. Please note that like the majority of other UNIX utilities, ed is case sensitive. This command is the upper case P. After you enter a P, ed will present an asterisk * as a prompt when it is in command mode. When ed is in text entry mode there will be no prompt. If you want to turn this option back off, simply type another P command. (Commands that are turned on and off by successive entries of the same key are called "toggle" commands. Several of the ed commands toggle.)

```
%ed <return>
P
*_  <— This is the cursor.
```

Figure 7.4

Error messages in ed

In accordance with the terse nature of early UNIX editors, ed is very brief when identifying input errors. Figure 7.5 shows the error message generated by ed. As it is less than helpful, Figure 7.6 shows how to invoke the ed help message description tool. Although ed is not terribly verbose, it does help in some cases, especially when you already know what you have done wrong. In this case, the p command told ed to print the current line in the editing buffer to the screen. However, there is no line in the editing buffer available to print, so ed recognizes it as an error. To show that it has a problem with a command, ed prints a ?.

Finally, Figure 7.7 shows how to turn on the verbose message option for your current session with ed. This option will remain on for your whole editing session.

NOTE : The text lines shown in italics are comments on the session and not part of the actual editing session.

NOTES

```
%ed <return>
p <return>
? <— Here is ed's error message (real helpful isn't it)
_
```

Figure 7.5

```
%ed <return>
p <return>
?
h <return>
line out of range   <—now we know what is wrong...
_
```

Figure 7.6

```
%ed <return>
H <return>
p <return>
line out of range
_
```

Figure 7.7

Leaving the ed editor

When you are ready to end your editing session all you need to do is enter the q command (for quit) at the command prompt.

If you have not saved the editing buffer to disk since your last change, ed will prompt you to do so by issuing an error message. If you enter a second q ed will do as you asked, exit, and *not* save your work. If you make the double q your standard exit from ed you will, at some time, exit without having saved your work and have the chance to save the text in the editing buffer.

Figure 7.8 shows us adding some text to a file, and then trying to exit without saving. It shows how ed will provide an error message prompting us to save (write) the contents of the buffer to the file. In this example we exited without saving our work.

In this same light, there is a way to leave the ed editor without any chance of saving your work. If you enter the Q command, ed will exit and not check the status of the editing buffer to see if the most recent changes have been saved. This is the most dangerous way to exit from ed. (Remember, back in

NOTES

the beginning I told you you would have to allow ed to help you remember to save the contents of the editing buffer...this shows you how you can prevent ed from helping you...this is generally considered to be a Bad Thing by the forces for good in the community.)

In Figure 7.9, we also added some text, but then issued the absolute quit command, Q. Notice that the next thing we see is the command line prompt, showing that we have exited form ed without saving our work.

```
%ed <return>
H <return>
P <return>
*a <return>
This is a test <return>
. <return>
*q <return>
?
warning : expected 'w'
*q
%
```

Figure 7.8

```
%ed <return>
H <return>
P <return>
*a <return>
This is a test <return>
. <return>
*Q <return>
%
```

Figure 7.9

Figure 7.10 shows the preferred way to respond to the error on exit message. Save the contents of the editing buffer to a file, and then reissue the q command to ask ed to exit to the shell.

Had we written our data from the buffer to the disk first, and then issued a quit command, ed would have simply exited to the command line with no message at all. Figure 7.11 shows that process.

NOTES

```
%ed <return>
H <return>
P <return>
a* <return>
This is a test <return>
. <return>
*q <return>
?
warning : expected 'w'
*w my.nifty.file
*q <return>
%
```

Figure 7.10

```
%ed <return>
H <return>
P <return>
*a <return>
This is a test <return>
. <return>
*w another.nifty.file <return>
*q <return>
%
```

Figure 7.11

Displaying the lines in a buffer

Each time you access a line of the buffer, ed shows you the contents of that line. However, it is often handy to be able to display all or a subset of the lines in a file so you can see what the file looks like.

There are two ways to show a set of lines in ed, either with or without line numbers. Using the l, or list, command will show you the current line. You can also specify the address of a particular line and see that line. Finally, you can specify a range of lines and see all of those lines. Figure 7.12 shows all of these variants. (I now will assume that you realize that you must type a <return> after each command or input line and will no longer show those <return>s.)

Sometimes it is handy to have ed tell us the line numbers. For example, if we are writing a program or script and the interpreter tells us that there is an error in line 46, it might be handy to see not only line 46, but the lines that surround 46. As with the list command, the n, or number command by itself will display the current or active line preceded by its line number. You

can specify a single line number, or a range of line numbers and see that line or range. Figure 7.13 shows how these options work.

This is a good time to introduce you to a couple of short cuts in addressing. As you have seen in the two examples of showing buffer contents, you can specify a first line number address and a last line number address and see all of the lines between those two, including the named lines.

There are times when you may want to see all of the lines in the buffer. Let's pretend that there are 19 lines in the buffer. You could specify 1,19l to see all 19 lines. However, suppose you don't know how many lines there are in the buffer. There is a magic address, $, that says "the last line in the buffer". So rather than typing 1,19l we can simply type 1,$l saving one whole keystroke. By the same token, we could type 10,$l and see all the lines from 10 to the last one in the file.

If you want to see all of the lines in the file, there is a second, even shorter cut you can use. Rather than typing 1,$n, you can simply type ,n (comma n). In this case ed will assume you mean 1,$n. That is pretty handy if your file is relatively short and you want to manipulate all of the lines. You can use these shortcuts with any command that takes addresses, or line numbers, as part of the command. So, for example, if you wanted to delete all of the lines in a file, you could simply type ,d. Short, sweet, quick, efficient, all of the qualities we want in an editor.

```
%ed a.boring.file
H
P
*l
Now is the time
*1,3l
Now is the time
For all good folk
To come and learn UNIX!
*
```

Figure 7.12

```
%ed a.boring.file
H
P
*n
1       Now is the time
*1,3l
1       Now is the time
2       For all good folk
3       To come and learn UNIX!
*
```

Figure 7.13

NOTES

Moving about in a file with ed

The easiest way to move about in a file is to simply enter the line number you wish to "go to". Figure 7.14 shows how to enter 3 lines of text, and then transfer control to line 1 and then to line 3. Note the use of the n command to display the line number and the text of the line after it becomes the current line.

Figure 7.15 shows one of the ways to perform relative addressing in ed. The editor is started, text is entered into the editing buffer, and then the text entry (append) mode is ended by the period (.). At this point, the third line, (the most recent line entered) is the current line. Next the command -2 is entered. This command is the same as the .-2 command (. standing for *"here"* or *the current line*). Next the n command is used to show which line is now the current line. Please don't confuse the period used to end the append session with the period used to represent here or the current line. It is obvious to the most casual observer that the two symbols are very different. Yea, right, sure! ;-)

```
%ed a.boring.file
H
P
*a
Now is the time
For all good folk
To come and learn UNIX!

*1
Now is the time
*n
1       Now is the time
*3
To come and learn UNIX!
*n
3       To come and learn UNIX!
*
```

Figure 7.14

```
%ed
H
P
*a
Now is the time
For all good folk
To come and learn UNIX!

*-2
Now is the time
*n
1       Now is the time
*
```

Figure 7.15

Adding text to a file

Figure 7.16 shows how to insert text into a new file. (You have seen this example before, now we will discuss the actual mechanism). As you will remember, we cannot insert text into a new file because insert inserts text *before* the current (or named) line, and we have no lines yet. Therefore we must append our text to the first line of the file (line zero). After we have appended a line, we can then use the insert command to add additional text to the body of the file. The (Please note, from now on, the examples assume you have typed both the H and the P commands at some previous point in the editing session.)

Searching in a file

Figure 7.17 shows the two common ed search tools. The first is the *forward* search which starts at the current line and searches forward through the rest of the file to find the *first occurrence* of the string specified between the slashes. In this example, the search stops when it finds the first occurrence of the word "UNIX". The second search shown works upwards, or *backward* (toward the beginning of the file) from the current line. It stops when it finds the first occurrence of the string specified between the question marks. Please note : The editing buffer seems to be circular when searching with these tools. You can visualize a circular buffer as if it has its contents on the outside of a cylinder. When you reach the last line of the file, the next line "down" is the first line of the file. By the same token, if you move to the line "above" the first line of the file, you reach the last line of the file.

Figure 7.18 shows the global parameter. When the g (or global) command precedes the search pattern, each line in the file is checked for the occurrence of the pattern specified. In this case, the pattern is a lower case l (ell). Each line that contains this pattern will be reported (printed on the screen)

Please note that the search pattern is often a Regular Expression.

NOTES

```
%ed
H
P
*a
To come and learn UNIX!
.
*i
Now is the time
For all good folk
.
*
```

Figure 7.16

```
*/UNIX/
To come and learn UNIX!
* ?Now ?
Now is the time
*
```

Figure 7.17

```
*1 the number one, to go to the first line
Now is the time
*g/l/
For all good folk
To come and learn UNIX!
*
```

Figure 7.18

Deleting lines from a file

Figure 7.19 shows how to remove lines from a file. The format of the delete command is #, #d where the line number or numbers are optional. If there are no line numbers given, the delete command will delete the current line. The delete command can be preceded by a single line number to delete a single line. Remember, the undo command (u) will undo the most recent delete (if you remember to do it before you execute another undoable command.)

Finding (or changing) the name of the current file

In Figure 7.20 we see how to determine the name of the file associated with the buffer we are currently working with. (Remember, the contents of the file will not be altered until we issue a w write command.) The f command

by itself prints the name of the file associated with the current editing buffer. If we include the name of a file, then that file becomes associated with the current buffer. Remember, the contents of the editing buffer will not be saved in that file until you issue the write w command.

```
*,1
Now is the time
For all good folk
To come and learn UNIX!
*2,3d
*,n
1       Now is the time
*
```

Figure 7.19

```
ls
nifty_ed_file
ed nifty_ed_file
P
*,n
1       Now is the time
2       For all good folk
3       To come and learn UNIX!
*f
nifty_ed_file
*f new_file
*w
57
*!ls
new_file    nifty_ed_file
!
*
```

Figure 7.20

Combining two lines together

Figure 7.21 shows the join command j. This command will remove the carriage return, line feed (newline) at the end of the first line, joining the second line with the first. This command takes as parameters, two *contiguous* lines. (That means you can join lines numbered 4 and 5, but not lines numbered 4 and 11.) Please note in the example, that the two lines are joined, but as there were no spaces either at the end of the first line nor at the beginning of the second line, there is no space between the words "time" and "For". You would need to go back into this file and fix that mistake after the join command finished.

NOTES

Moving lines around in the buffer

This editor also gives you the opportunity to move lines from one place to another in the buffer. Figure 7.22 shows how to accomplish a simple move. Please note that the line(s) to be moved are moved to the line *following* the line listed as the target.

```
*,n
1     Now is the time
2     For all good folk
3     To come and learn UNIX!
*1,2j
*,n
1     Now is the timeFor all good folk
2     To come and learn UNIX!
*
```

Figure 7.21

```
*,n
1     Now is the time
2     For all good folk
3     To come and learn UNIX!
*3m1
*,n
1     Now is the time
2     To come and learn UNIX!
3     For all good folk
*
```

Figure 7.22

Copying lines in the buffer

While the move command (m) gives us the functional equivalent to the standard *cut and paste* function, the copy command t (stands for transfer, perhaps< ?>) allows you to copy and paste. The format of this command is #,#t# where the line number pair give ed the range of lines you wish to copy, and the last line number gives the location to copy to. As with move, the text will appear in the buffer *following* the line number listed. Figure 7.23 demonstrates a simple move.

Substitution in ed

This editor also allows specific search and replacement. The s command provides that capability. The format of the command is #,#s/target/new value/ n where n is a number indicating the occurrence on the line of the target to be replaced or substituted. If you do not provide a value for n, the substitute command will replace the first occurrence of the target string. Should you

NOTES

want ed to replace all the occurrences of the target string with the new value, use a "g" for global, in place of the n. Figure 7.24 shows this command and the results of a simple substitution.

Remember, substitution is your very best friend!

Other ed commands

There are many more editing commands, as listed in the "Table of Useful ed Commands" following the next set of examples. After you master the commands we have discussed here, you should play with, and learn, the rest of the ed commands. You will find that a complete grasp of the intricies of ed will help you with all of your other editing tasks. Besides, ed is a pretty cool tool!

```
*,n
1       Now is the time
2       For all good folk
3       To come and learn UNIX!
*2t3
*,n
1       Now is the time
2       For all good folk
3       To come and learn UNIX!
4       For all good folk
*
```

Figure 7.23

```
*,n
1       If you knew UNIX, like I knew UNIX
2       You would run away from UNIX ;-)
*1,2s/UNIX/Unix/2
*,n
1       If you knew UNIX, like I knew Unix
2       You would run away from UNIX ;-)
*2s/UNIX/Unix/
*,n
1       If you knew UNIX, like I knew Unix
2       You would run away from Unix ;-)
*
```

Figure 7.24

A table of useful ed commands

Command Action

! This command will act like a shell escape character and allow the subsequent key strokes to be passed to the UNIX shell and executed.

NOTES

In most cases control returns to ed after the execution of the command.

a — Allows you to leave command mode and append text *after* the current line.

c — The change command allows you to alter a range of text. The generic format of the command is [#,#]c where #,# is a range of lines you wish to replace with text you will type. This command first deletes the existing text over the range specified, and then puts you into insert mode to type new text to replace that which has been deleted.

d — This command allows you to delete one or more lines, the generic format of the command is [#,#]d where #,# is a range of lines to be deleted. The line after the last line deleted becomes the current line.

e — Reads the previously saved version of the current file into the editing buffer. If a file name is specified, then ed loads the most recent version of that file into the editing buffer, overwriting the current contents.

f — By itself, f displays the current filename, if you give f a file name, then you give a new name to the contents of the editing buffer. The syntax for this command is f or f *filename* to set a new filename.

g — Is the global option. If g precede a command, then the entire buffer is searched for all first occurrences of the pattern in each line, if the g follows the command, then the whole line is searched, and ed does not stop with the first occurrence. It is possible to attach a command-list to this command to be executed each time the pattern is matched. The normal syntax of this command is [#,#]g/RE/ command-list. The commands in the command list must be separated by protected new lines (\<enter>). If the g both precedes and follows a command, then the complete file is searched.

G — This command is the same as the g command, except it can have only 1 command following it rather than a command list.

h — Displays the most recent error message.

H — Displays, (or redisplays) the most recent error message and turns on the automatic, verbose error message display for the current editing session. This command is a toggle, if you enter a second H, it will turn verbose error display off. The default for this option is off.

i — This command allows you to insert text *before* the current line. If you precedes the command with a number, the text will be opened for insertion *before* the line specified.

A table of useful ed command (cont.)

NOTES

Command	Action
j	The j command allows you to remove the carriage return between any two lines, joining them into one line. The format of that command is : [#,#]j where # and # are two contiguous lines to be joined. If you do not give join two line numbers, the command will fail, but no error message will be given.
k	This command will set a marker in the text so you can find a particular line later. The format of this command is [#]k*x* where *x* is one of the letters from the range [a-z]. If no line number (#) is supplied, the current line is marked. To transfer control back to a line so marked, use the 'x command. (that is tick-mark x)
l	List either the entire contents of the buffer, or in the form [#,#]l, list the range of lines specified. In the second form, the last line listed becomes the current line.
m	The move command moves a specified range of text to a new location in the editing buffer. The correct form for the move command is : [#,#]m# where the #,# pair are the lines to be moved, and the # following the command shows where the lines will be moved. The line(s) will be moved to the line *following* the target line.
n	The generic form of the number command displays some range of numbers ([#,#]n) on the screen, preceding each line with the line number and the tab character. At the end of the command, the last line displayed becomes the current line.
p	This command prints either the current line or the lines specified by the number range that precedes the command ([#,#]p), then the lines listed are printed (shown on the screen) and the last line printed becomes the current line.
P	This toggle turns on or off the command mode prompt. (The default for this toggle is off.)
q	q lets you quit an ed session. If you have not saved the contents of the editing buffer, ed will give you the ? error message. If you issue another q command, ed will exit and not save the contents of the editing buffer.
Q	If you want to end an ed session without verifying that the contents of the editing buffer have been saved, use the Q command. This is a dangerous command.
r	The format of this command is : [#,#] r *filename*, where the contents of the file specified by filename is copied into the file at the location following the line(s) specified. If no filename is specified, the current file is used, and if there is no current file, an error message is generated.

Command	Action
s	The substitute command searches the current line for the specified text (stext). If ed finds the search text, it replaces it with the

NOTES

replacement text (rtext). The form of the command is [#,#]s/stext/rtext/n. If the line numbers are omitted, substitute works on the first occurrence it finds. This command can take an n suffix, if used, the nth occurrence of the stext will be replaced. The global options are frequently used with this command.

t — [#,#]t# is the format of the copy command (to remember it, you are copying from a range to a line number ?). This command copies the line(s) of text specified and inserts them *after* the line number specified as the target.

u — The undo command will help you recover from an error. You can undo the actions of the most recent a, c, d, i, j, m, r, s, t, u, v ,G, or V command. You must use the undo command before you issue another command in the list above, for the undo to work. If you leave the ed editor and return, you cannot undo a change. You cannot undo a change that has been written to disk. Please note, you can undo the results of an undo command as well!

v — This command is the opposite of the g command. Where g finds all occurrences of the pattern listed, v finds all lines that *do not match* the pattern. If a command-list is supplied, that command line is executed for each line that does not match the specified pattern. The proper syntax is [#,#]v/RE/command-list.

V — This command is the opposite of the G command. (See the discussion of v).

^V — Using this command allows you to enter non-displaying character ASCII code into the buffer. For example the string ^V^G imbeds the ASCII code for ^G (beep) into the text. Useful for some specialized ASCII (or "escape") sequences. (^ represents the CTRL or control key, so ^V means Control-V.)

w — If no file name is specified, this command writes the contents of the editing buffer to the disk file that was opened (or edited, or read) to initially load the editing buffer. This command will allow a subset of the editing buffer to be saved, that format is [#,#]w *filename*. If no line numbers are specified, the whole file is written. If a file name is specified as an argument, then that portion of the editing buffer specified is written to the disk and saved under the filename.

W — This command is similar to the w except that it *appends* to an existing file rather than writing over the contents.

/RE/ — This is used to search *forward* through the file for the first occurrence of the pattern enclosed in slashes. In this case, that is a regular expression.

?RE ? — This is used, like the slashes, but starts a search *backward* Please note, for purposes of these searches, the buffer is circular.

STUDENT ACTIVITY

1. What is a Shell Script ?

2. What is similarily between DOS Batch file UNIX Shell Script ?

3. What do you mean by redirection ?

4. What is standard output in Union Operating System ?

NOTES

SUMMARY

- Operating System makes interaction between user and hardware.
- UNIX structure consists of Kernel, shell and other tools and application.
- UNIX offers 64 systems tools.
- Exception is a condition where the system causes unexpected event.
- Unix has the concept of Interupt.
- File system in unix is arranged in weraschical order.
- Unix treats all Information a files.
- Unix file system consists of many directors which contain vital information used for working in the environment.
- I – node is a table associated with each file in UNIX.
- /bin directory contains executable files for UNIX commands.
- Unix is case sensitive.
- The banner command displays its argument exploded to a bigger size, into the standard Output.
- The cal command creates a calendar of the specific month for specified year.
- The password command allows the user to set or change the password.
- The who command lists the users that are currently logged into the system.
- The finger command with an argument gives you most information about the user.
- In UNIX , all utilities , applications and date are represented as files.
- The META CHARACTERS are A, ? and { }.
- The is command is used for listing information about files and directories.
- The cp command creates a duplicate copy of a file.
- *The mv command moves or renames files.*
- The In command adds one or more links to a file.
- The rm command removers files or directories.
- The cat command displays the content of a file into the screen.
- The pwd command is used for printing the complete pathname of your current working directory.
- The mkdir.command helps you to create a new directory.
- The cd command helps you to change from one directory to another.

- The rmdirr command removes directories from the disk.
- The chmod command changes the access permission of a file.
- The chown command changes the owner of the specified file(s).
- The chgrp command is used to change the group of a file.
- Unix proceed with various types of editors like ed, ex , vi, sed etc.
- vi can be invoked by typing in vi at the $ promp followed by filename.
- Ed editors is line editors – Edition can be done based on line numbers.
- The insertion of text in vi editor is done in the insert mode , append mode or open mode.
- There are commands to delete join two lines or undoing a command in vi.
- The sed editors does not show changes made to the file unlike the other editors.
- This used to replace, search , view and delete required number of lines from file.
- Pipes and filters help in performing multiple tasks in a single command line. A pipe takes output of a command as its input. A filter takes input from the standard Input, process it and than displays it on to the standard output.
- Sort filter is used to arrange the input in alphabetical order. This command comes with different options.
- The grep command is used to search for a specified expression. The options available with grep command are –V , -I, -N, -C. There are various ways to specify the regular expressions to be searched.
- The output and the input can be redirected to sources other than the standard Input and the standard output. This is called Redirection.
- Redirection may be Input redirection or Output redirection. Both Input and Output redirection is possible simultaneously.
- The unix system command is shell.
- Shell has the ability to redirect the standard input, output and error file.
- Shell script is used to write many commands and execute then at one go.
- Bourne shell is a command processor used on all Unix systems.
- C shell has executable file named csh.
- Korn shell has executable file named ksh.
- A process is an entity that represents the basic unit of work to be implemented in the system.
- The process table contains details of every process that is initialized.

NOTES

NOTES

- The ps command displays the status of all active process.
- The STIME stands for starting time of Process.
- The nice command executes a command with a different priority than others processess.
- A daemon is a process that executes in the background in order to be available all the times.
- Shell uses a set of symbols called meta character to form a pattern, which is used to match various classes of filenames replacing full names.
- There are different types of shall variables available. These are user defined , environment, local and global valuable.
- User defined variables are created by user environment variables are those which are predefined by the system, local variables are known only to the shell on which we are working , global variables are known to the subsequent shells too.
- Shell offers programming language constracts like if else, switch case, for loop and while loop.
- There are special features offered by the shell for writing shell script some of them are symbol, command substitution , positional parameters, escape mechanism, arithmetic and conditional operators, the shift and exit command.
- The nl command and the pr commands are useful for preparing documents for printing.
- UNIX provides the following utilities for printing the text.
  i. A filter that formats text for the printer.
  ii. A print spooling program.
  iii. A program that actually writes the spooled print output to the printer.
- Super User is a System Administrator.
- Booting is the process of starting up a computer.
- The password file contains a one line entry for every account.
- The for command allows you to take a backup of all or selected files in a directory, hierarchy on to external storage device or the hard disk itself.
- All error messages should be directed to the standard error.

SELF-ASSESSMENTS QUESTIONS

NOTES

Solved Exercise

I. True or False :

1. UNIX user the concept of time sharing.
2. An OS looks after the management of the devices of the system.
3. Unix doesnot offer security to files of a user.
4. The devices are also treated as files in Unix.
5. Upper case and lower case are treated same in UNIX.

II. Fill in the Blanks :

1. ..... forms the core of the UNIX OS.
2. ..... forms the interface between Kernel & the user.
3. Password cannot have ..... in it.
4. Ln is used to ..... to another file.
5. Ed is a ..... editor while vi is a ..... editor.

Answers

I. True or False :

1. True
2. True
3. False
4. True
5. False

II. Fill in the Blanks :

1. Kernel
2. Shell
3. special character
4. link
5. line, screen

Unsolved Exercise

I. True or False :

1. The executable filename of Bourne shell is Ksh.
2. The file system in UNIX forms an important post of the UNIX system.
3. Login name can be combination of letter and numbers.
4. Login name can be of upto 2 characters.
5. Date command displays current Date on the screen.
6. Filenames can be upto 14 character.
7. Chgrp can be used to change the group of file.
8. mv can be used to move or rename a file.
9. rm is used to rename a file.
10. cp is used to displays contents of a file.
11. When vi is started, it is in insert mode.

NOTES

12. Editing is as easy in vi as in any word processor.
13. You can execute vi commands repeatedly by adding a number to the command.
14. Emacs works in only one mode.
15. Emacs is a part of UNIX system.
16. The data from one filter can be passed to another filter.
17. The tee command displays the contents of the file one page at a time.
18. The wc command counts characters, words and lines in the input file.
19. The grep command cannot search the input file for a matching pattern.
20. The sort command by default sorts the file on the key field and options specified.
21. The shell is UNIX system's command interpreter.
22. Shell is not a programming language.
23. The Bourn shell is the fastest and the widely used UNIX command processor.
24. The \c character used with the Echo command keeps the cursor on the same line of the screen after displaying the argument.
25. Korn shell was developed by William Joy.
26. The for construct is used when we want menu driver shell scripts.
27. The while loop may become infinite under certain circumstances
28. The test command is used to test the equality or requality of the integers.
29. The until loop is executed as long as the control command execution become true.
30. The for loop perform the same operation on list of values.
31. The $ @ and the * are identical.
32. Under the Bourine Shell, the priority number must be proceeded by a '-' symbol.
33. You can run the sort command in both foreground and background.
34. You can use a signal number with a command for stopping all the running processes in background.
35. Process 0 is the mother of all processes.
36. Ps –d command shows the listing of only currently active processes.
37. The nl command cannot be used to display formatted output on the screen.
38. The pr command with option -r prints all files simultaneously in different columns.
39. The pr command with -pn option starts printing at page n where n indicates an integer.
40. The pr command is also used to control the increment of the line numbers.
41. The lp command with flag -Ppr sends the print job to the printer named ptr.

II. Fill in the Blanks :

1. Efficiency of the UNIX system ..... as the number after marks increases.
2. The library functions proceeded by UNIX are contained in the ..... Directory.
3. The/tmp directory contains ..... .
4. The files of the user are stored in the directory.
5. The ..... symbol tells the user that unix is ready to accept commands.
6. ..... command can be used to change the password.
7. ..... command is used to gets Calendar.
8. A group of files in unix can be referred with the help of ..... .
9. ..... command is used to display contents of file.
10. ..... command is used for changing owner of the object
11. ..... option in Is command is used to print rode number of each file in first column
12. The commands used to cut the text in vi are called ..... commands.
13. ..... is the restricted version of vi.
14. ..... command search forward s string 'pattern' in the vi text file.
15. Press ..... keys to save the file and quit emacs.
16. The basic source from which UNIX programs read their input is called ..... .
17. ..... messages are normally displayed on the screen but you can also ..... them to a file or printer.
18. ..... are the programs that read from standard input file, process it and write the output to the standard output file.
19. The ..... command displays the first 10 lines of a file.
20. The..... command concatenates data from files line by line.
21. The shell supports UNIX's multitasking feature using the ..... method.
22. Shell offers standard programming constructs like ....., ..., ..., etc.
23. A shell script is similar to the ..... files of DOS where a ..... of frequently used UNIX commands are stored in a file.
24. The executable filename of Korn shell is ..... .
25. The ..... command is used to display the messages on ..... .
26. The ..... is the name given to a data structure that store the details about all processes.
27. ..... command is used to view all the information about all the processes running on the entire system.
28. The ..... command executes a command with a different priority than other processes.

NOTES

29. The ..... command terminate a process.
30. In ..... operating system, the many processes run in background.
31. ..... and ..... commands are used for preparing documents for printing.
32. The pr command with option ..... prints document in double space.
33. The pr command with option ..... provides text as a header.
34. The lpr command with flag ..... do not print the header page.
35. ..... command prints the listing in two columns.

III. Match the following :

(*a*)	:W	(*i*)	Cuts (yanks) the current line.
(*b*)	$	(*ii*)	Deletes the current line
(*c*)	G	(*iii*)	Inserts characters before the cursor
(*d*)	<Ctrl>F	(*iv*)	Functions as a right arrow key
(*e*)	dd	(*v*)	Scrolls the full screen forward
(*f*)	yy	(*vi*)	Moves cursor to the last line of the file
(*g*)	:D3	(*vii*)	Saves the files with existing name and resumes editing
(*h*)	i	(*viii*)	Delete three lines from the current line
(*i*)	R	(*ix*)	Search backward for the previous pattern
(*j*)	?	(*x*)	Replace characters with characters until <Esc> key's pressed.

Detailed Questions

1. How many types of shell are present in UNIX environment ?
2. What is default shell for UNIX ?
3. What is executable file name for Bourne Shell ?
4. Why do we require a Back Slash ?
5. Differentiate between use of (1) single quote and (4) double quote.
6. What do you understand by a variables ?
7. What are two types of variables present in UNIX ?
8. How do you read a value of a variable ?
9. What is on if else fi construct represent ?
10. Why is fi required at end of a program ?
11. Why do we require numeric operators ?
12. How many types of numeric operators there in all ?
13. What is use of string variable operators ?
14. What do you understand by –n option ?
15. Differentiate between –z & -n option.
16. Write a shell script to find if a file exist in a directory or not.

NOTES

17. For what purpose – a option is used ?
18. Can options be used simultaneously, with each other ?
19. What is use of logical operators in Shell Script ?
20. Write a Shell script to check if a year is a leap year or not using one of loggical operators.
21. Write a Shell script to add, multiply, divide & subtract two numbers occupting from user.
22. Why is case.... esac construct used ?
23. How is for – do – done construct different from case – esac construct?
24. Give syntax for – do – done construct.
25. How is while loop different from of or do done construct?
26. How many times a loop is executed?
27. Differentiate between while and until loop.
28. How many times until loop executes ?
29. Give specification of password file.
30. Can the Password file be changed ?
31. What do you understand by Editor ?
32. How many types of Editors are present ?
33. What is need of an editor ?
34. Give steps to Start vi Editor.
35. What are different modes for working in vi ?
36. Give command to save text in a file.
37. What are various ways to modify text & How ? Give procedure.
38. How to delete a characters a line ?
39. How can a file be redirected to standard error file ?

LAB EXERCISES

1. Write a Shell script to check if the input string is a palindrome.
2. Write a Shell script to accept a number and a word as command line arguments and print the word the given number of times on each line. What happens if a number and a string is given as command line arguments ?
3. Write a Shell script to find the file with the maximum size in the current directory.
4. Write a Shell script to accept an input and check if the given input is a directory. If it is a directory, then display its contents and revoke the execute permission for group and others for all files starting with "b" in the directory.

5. Write a Shell script to accept two filenames and check if both exist. If the second filename exists, then the contents of the first filename should be appended to it. If the second filename does not exist then create a newfile with the contents of the first file.
6. Write a Shell script to accept a number in the command line and displays the sum up to that number. For example, if the number given is 20, then the sum of numbers 1, 2, 3, ..., 20 should be displayed. By default, the sum upto 50 should be displayed.
7. Write a Shell script to find the number of ordinary files and directory files in the current directory.
8. Write a Shell script to accept an alphabet from the user and list all the files starting with that letter starting from the current directory.
9. Write a Shell script to accept the name of the directory as command line argument and display the long listing of that directory. By default the HOME directory's contents should be displayed.
10. Write a Shell script which accepts a number and displays the list of odd numbers below that number.

Solutions to Lab Exercises

1.
```
# To check if the input string is a palindrome
# Usage $0 string
    If [ $tt -It I ]
    then
        echo "neage $0 string"
        echo "invalid argiunenta"
        exit
    fi
    cnt= 'echo $1 | wc -c'
    inp=$1
    while [ $cnt -gt 0 ]
    do
        var-'echo $1 | cut – c$cnt'
        cnt='expr $cnt - I'
        temp='echo $cenp$vftr'
    done
    if [ $inp - $temp ]
    then
        echo $inp is a palindrome
    else
        echo $inp is not a palindrome
    fi
```

```
echo "the input string is $inp"
echo "the reversed string is $temp"
```

2. # To take as input a number and a word and display the word that many number of times

```
# Usage $0 number word
    if [ $« -it l ]
    then
        echo "usage $0 number word"
        echo "invalid usage"
        exit
    fi
    cnt-1
    while [ $cnt -le $1 ]
    do
        echo "\n your word is $2"
        cnt='expr $cnt + 1'
        done
```

If we type in a string instead of a word in the command line, we must enclose the string within double quotes.

3. # To find the file with the maximum size in the current directory and display

```
max=0
for k in '1s-1 [ tr -s " " | cut - d" " –f 5'
do
    if [ $k -gt $max ]
    then
        max=°$k
    else
        max='$max
    fi
done
tem='1s -1 | grep $max | tr -s " " | cut -d" " –f 5, 9'
echo the file with largest size is : $tem
```

4. # to accept a filename as input and if it is a directory display its contents and revoke the execute permission for group and others.

```
# Usage $0 filename
if [ $» -It I ]
then
    echo "invalid usage : Usage $0 filename"
    exit
```

NOTES

```
fi
if [-d $1 ]
then
        echo "$1 is a directory"
        echo "displaying the existing permissions for files starting with "b"
        cd    $1
        is -1 b*
        chaod 766 b*
        echo "displaying the changed permiasione for files starting with "b"
        1s -1 b*
elif [-f $1-a-r $1 ]
then
        echo "$l Is an ordinary file. displaying its contents..."
        cat $1 | pg
else
        ecbo "$l does not exists"
fi
```

5. # to accept two filenames and check if both exists. If second file exists then append the contents of the first file to it.

```
# Usage $0 file 1 file2
if [ $#-it i ]
then
      echo "invalid usage : Usage $0 file1 file2"
      exit
fi
if [ -f $1 ]
then
      if [ -f $2 ]
      then
            echo "both $1 and $2 exist"
            echo "appending data from $1 to $2"
            cat   $2 » $1
            echo "displaying contents of $2... "
            cat $2
      else
            echo "$2 does not exists, creating a new file..."
            cp $1 $2
            echo "displaying contents of $2..."
```

```
        cat $2
    fi
else
    echo "$1 does not exists or $1 is not an ordinary file"
fi
```

NOTES

6. # to accept a number from the user and calculate sum upto that number. By default sumup to 10 is found out and displayed

```
case $# in
0) vari-10; ;
*) vari-$1;;
esac
while true
do
    var1='expr $vari + 1'
    var='expr $vari \* $varl / 2'
    echo "sum upto $vari numbers is : $var"
    exit
done
```

7. # to find the number of ordinary files and directory files

```
if [ $# - 1t 1 ]
then
    echo "invalid Usage. Usage $0 directoryname"
    exit
fi
if [ -f $1 ]
then
    echo "$l is an ordinary file"
    exit
fi
if [ -d $1 ]
then
    echo "$1 is a directory"
    cd $1
    var='1s -1 | grep -c "^d"'
    varl= '1s -1 | grep -c "^_"'
    echo "number of directory files in $1 directory : $var"
    echo " number of ordinary file in $1 directory : $varl"
fi
```

NOTES

8. # to accept an alphabet and find the names of the files starting with that alphabet from the current directory

```
# Usage $0 alphabet
if $»-it i I
than
        echo "invalid usage. Usage $0 alphabet"
              exit
fi
find. -name "$1*" -print > filel 2 >err
if [ -s filel ]
then
        echo "displaying those tiles starting with "$l" from the current directory..."
        cat filel
        else
        echo "there are no tiles starting with "$l"
        fi
```

9. # to accept a directoryname and display its contents
if input is not given then HOME directory's contents should be listed case

```
$# in
0)      dir=$HOME;;
*)      dir=$l;;
esac
it [ -t $dir ]
then
        echo "$l is an ordinary file"
        exit
fi
if [ -d $dir ]
then
        echo "displaying the contents of the directory $dir"
        cd $dir
        Is -1
        exit
else
        echo "$l does not exists"
fi
```

10. # to display the list of odd numbers below a given number

```
#usage $0 number
cnt=1
if [ $# - 1t 1 ]
```

```
then
        echo "invalid usage, usage $0 number"
        exit
else
        echo "displaying the odd numbers below $1 ..... "
        while [ $cnt - 1t $1 ]
        do
                echo $cnt
                cnt='expr   $cnt + 2'
        done
fi
```

NOTES