BCA(N)-102

School of Computer Science & IT
Uttarakhand Open University,
Haldwani

Operating System

Block-1

Unit-1

Introduction to Operating System

1.1 Introduction to Operating system

Operating System (or shortly OS) primarily provides services for running applications on a computer system.

Need for an OS:

The primary need for the OS arises from the fact that user needs to be provided with services and OS ought to facilitate the provisioning of these services. The central part of a computer system is a processing engine called CPU. A system should make it possible for a user's application to use the processing unit. A user application would need to store information. The OS makes memory available to an application when required. Similarly, user applications need use of input facility to communicate with the application. This is often in the form of a key board, or a mouse or even a joy stick (if the application is a game for instance).

Keyboard

Mouse

Joystick

Monitor

Printer

The output usually provided by a video monitor or a printer as some times the user may wish to generate an output in the form of a printed document. Output may be available in some other forms. For example it may be a video or an audio file.

Let us consider few applications.

- Document Design
- Accounting
- E-mail
- Image processing
- Games

We notice that each of the above application requires resources for

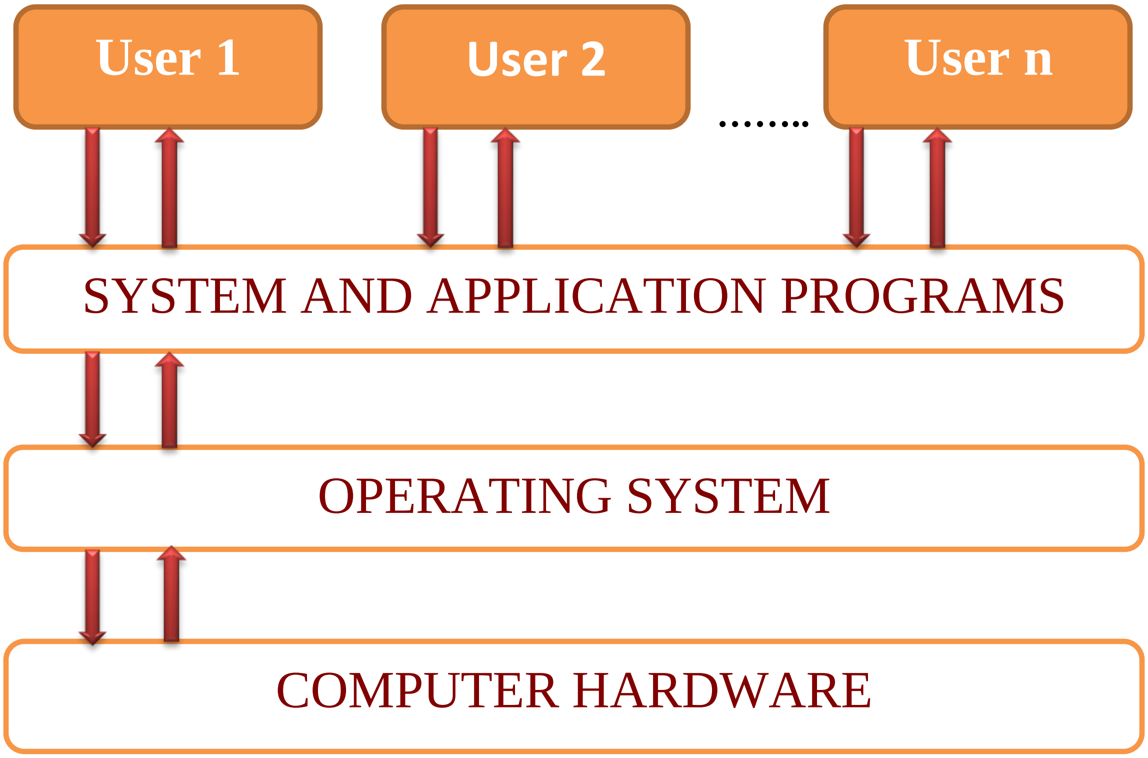
- Processing information
- Storage of Information
- Mechanism to inputting information
- Provision for outputting information
- These service facilities are provided by an operating system regardless of the nature of application.

The OS offers generic services to support all the above operations. These operations in turn facilitate the applications mentioned earlier. To that extent an OS operation is ***application neutral*** and ***service specific.***

1.2 User and System View:

From the user point of view the primary consideration is always the convenience. It should be easy to use an application. In launching an application, it helps to have an icon which gives a clue which application it is. We have seen some helpful clues for launching a browser, e-mail or even a document preparation application. In other words, the human computer interface which helps to identify an application and its launch is very useful.

This hides a lot of details of the more elementary instructions that help in selecting the application. Similarly, if we examine the programs that help us in using input devices like a key board – all the complex details of character reading program are hidden from the user. The same is true when we write a program. For instance, when we use a programming language like C, a printf command helps to generate the desired form of output. The following figure essentially depicts the basic schema of the use of OS from a user stand point. However, when it comes to the view point of a system, the OS needs to ensure that all the system users and applications get to use the facilities that they need.



Also, OS needs to ensure that system resources are utilized efficiently. For instance, there may be many service requests on a Web server. Each user request need to be serviced. Similarly, there may be many programs residing in the main memory. The system need to determine which programs are active and which need to await some form of input or output. Those that need to wait can be suspended temporarily from engaging the processor. This strategy alone enhances the processor throughput. In other words, it is important for an operating system to have a control policy and algorithm to allocate the system resources.

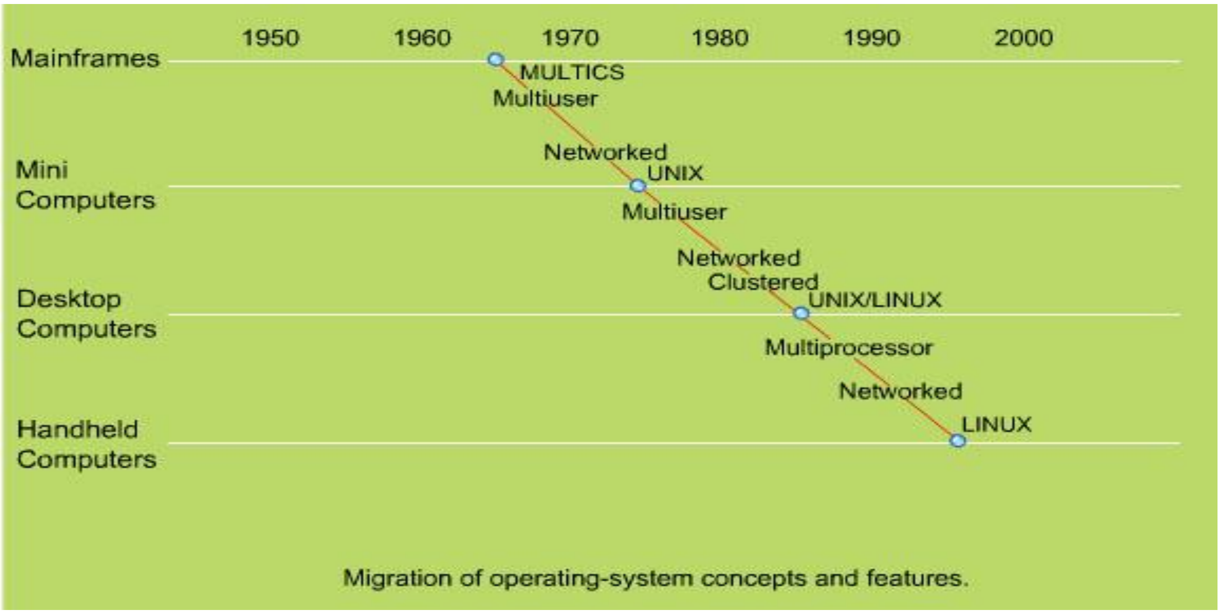
1.3 The Evolution:

It would be worthwhile to trace some developments that have happened in the last four to five decades. In the 1960s, the common form of computing facility was a mainframe computer system. The mainframe computer system would be normally housed in a computer center with a controlled environment which was usually an air conditioned area with a clean room like facility. The users used to bring in a deck of punched cards which encoded the list of program instructions.

- ➢ The mode of operation was as follows:
- ➢ User would prepare a program as a deck of punched cards.
- ➢ The header cards in the deck were the “job control” cards which would indicate which compiler was to be used (like Fortran / Cobol compilers).
- ➢ The deck of cards would be handed in to an operator who would collect such jobs from various users.
- ➢ The operators would invariably group the submitted jobs as Fortran jobs, Cobol jobs etc. In addition, these were classified as “long jobs” that required considerable processing time or short jobs which required a short and limited computational time.

Each set of jobs was considered as a batch and the processing would be done for a batch. Like for instance there may be a batch of short Fortran jobs. The output for each job would be separated and turned over to users in a collection area. This scenario clearly shows that there was no interactivity. Users had no direct control. Also, at any one time only one program would engage the processor. This meant that if there was any input or output in between processing then the processor would wait idling till such time that the

I/O is completed. This meant that processor would idling most of the time as processor speeds were orders of magnitude higher than the input or output or even memory units. Clearly, this led to poor utilization of the processor. The systems that utilized the CPU and memory better and with multiple users connected to the systems evolved over a period of time as shown in Table1.1.

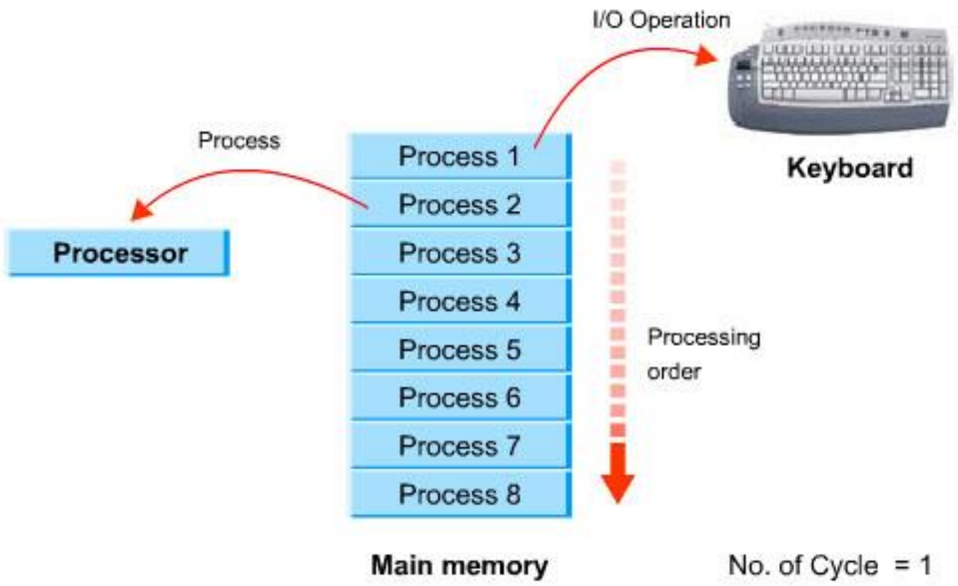


Migration of operating-system concepts and features.

At this time we would like to invoke Von - Neumann principle of stored program operation. For a program to be executed it ought to be stored in the memory. In the scheme of things discussed in the previous paragraph, we notice that at any time only one program was kept in the memory and executed. In the decade of 70s this basic mode of operation was altered and system designers contemplated having more than one program resident in the memory. This clearly meant that when one program is awaiting completion of an input or output, another program could, in fact, engage the CPU.

Late 60's and early 70's

Storing multiple executable (at the same time) in the main memory is called multiprogramming. With multiple executable residing in the main memory, the immediate consideration is: we now need a policy to allocate memory and processor time to the resident programs. It is obvious that by utilizing the processor for another process when a process is engaged in input or output the processor utilization and, therefore, its output are higher. Overall, the multiprogramming leads to higher throughput for this reason.



Multiprogramming

While multiprogramming did lead to enhanced throughput of a system, the systems still essentially operated in batch processing mode.

1980's

In late 70s and early part of the decade of 80s the system designers offered some interactivity with each user having a capability to access system. This is the period when the timeshared systems came on the scene.

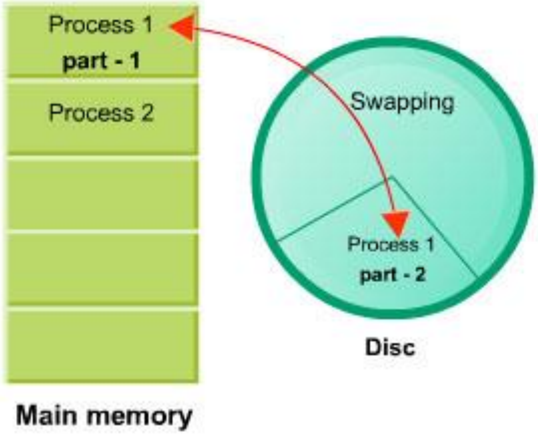
Basically, the idea is to give every user an illusion that all the system resources were available to him as his program executed. To strengthen this illusion a clever way was devised by which each user was allocated a slice of time to engage the processor. During the allocated time slice a users' program would be executed. Now imagine if the next turn for the same program comes quickly enough, the user would have an illusion that the system was continuously available to his task. This is what precisely time sharing systems attempted – giving each user a small time slice and returning back quickly enough so that he never feels lack of continuity. In fact, he carries an impression that the system is entirely available to him alone.

Timeshared systems clearly require several design considerations. These include the following: How many programs may reside in the main memory to allow, and also sustain timesharing? What should be the time slice allocated to process each program?

How would one protect a users' program and data from being overwritten by another users' program? Basically, the design trends that were clearly evident during the decade of 1970-80 were: Achieve as much overlapping as may be feasible between I/O and processing. Bulk storage on disks clearly witnessed a phenomenal growth. This also helped to implement the concept to offer an illusion of extended storage. The concept of “virtual storage” came into the vogue. The virtual storage essentially utilizes these disks to offer enhanced addressable space. The fact that only that part of a program that is currently active need be in the main memory also meant that multi-programming could support many more programs. In fact this could be further enhanced as follows:

1. Only required active parts of the programs could be swapped in from disks.
2. Suspended programs could be swapped out.

This means that a large number of users can access the system. This was to satisfy the notion that “computing” facility be brought to a user as opposed to the notion that the “user go to compute”. The fact that a facility is brought to a user gives the notion of a utility or a service in its true sense. In fact, the PC truly reflects the notion of “computing utility” - it is regarded now as a personal productivity tool.



Swapping of program parts main memory - disc, vice-versa

It was in early 1970s Bell Laboratory scientists came up with the now well-known OS:

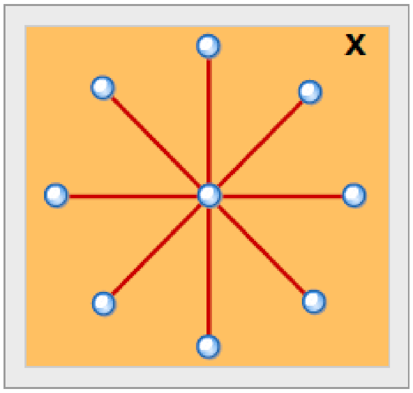
Unix. Also, as the microcomputers came on scene in 1980s a forerunner to current DOS was a system called CP/M. The decade of 1980s saw many advances with the promise of networked systems. One notable project amongst these was the project Athena at MIT in USA. The project forms the basis to several modern developments. The client-server paradigm was indeed a major fall out. The users could have a common server to the so called X-terminals.

The X windows also provided many widgets to support convenient human computer interfaces. Using X windows it is possible to create multiple windows. In fact each of the windows offers a virtual terminal. In other words it is possible to think about each of these windows as a front-end terminal connection. So it is possible to launch different applications from each of the windows. This is what you experience on modern day PC which also supports such an operating environment.

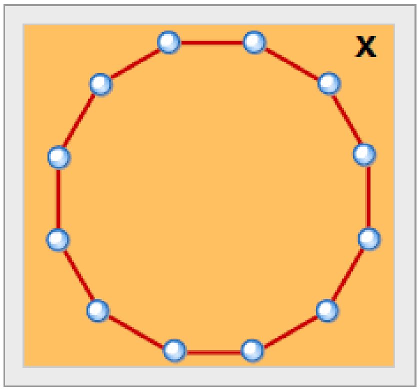
In our discussions, we shall discuss many of the above issues in greater detail as we move to later chapters. On the micro-computer front the development was aimed at relieving the processor from handling input output responsibilities. The I/O processing was primarily handled by two mechanisms: one was BIOS and the other was the graphics cards to drive the display. The processor now was relieved from regulating the I/O. This made it possible to utilize the processor more effectively for other processing tasks. With the advent of 1990s the computer Networking topologies like star ring and general graphs, as shown in the figure, were communication was pretty much the order of the day.

CP/M based computer

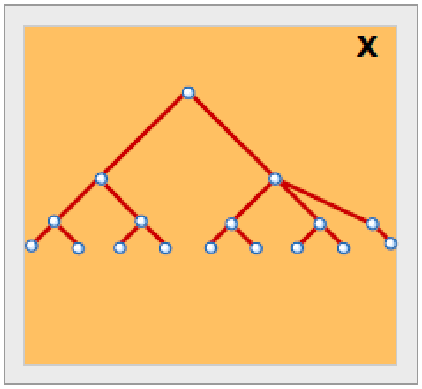
Being experimented with protocols for communication amongst computers evolved. In particular, the TCP/IP suites of network protocols were implemented. The growth in the networking area also resulted in giving users a capability to establish communication between computers. It was now possible to connect to a remote computer using a telnet protocol. It was also possible to get a file stored in a remote location using a file transfer (FTP) protocol. All such services are broadly called network services.



(a) Star

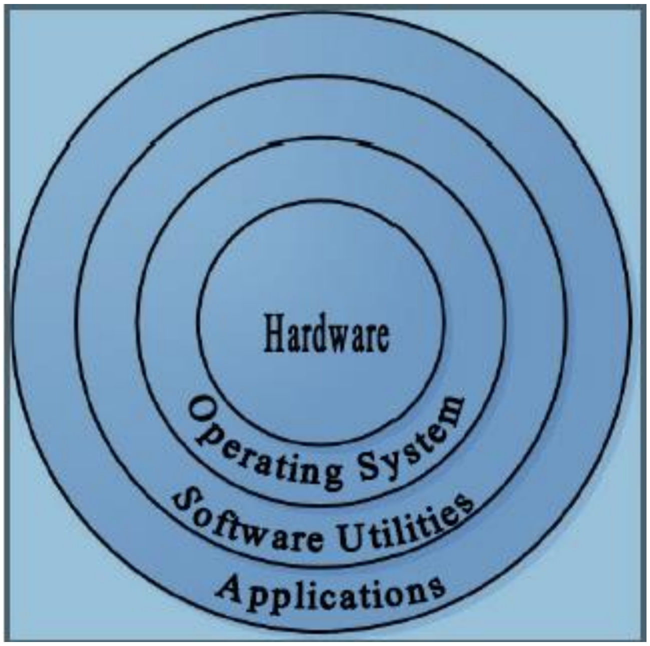


(b) Ring



(c) Tree

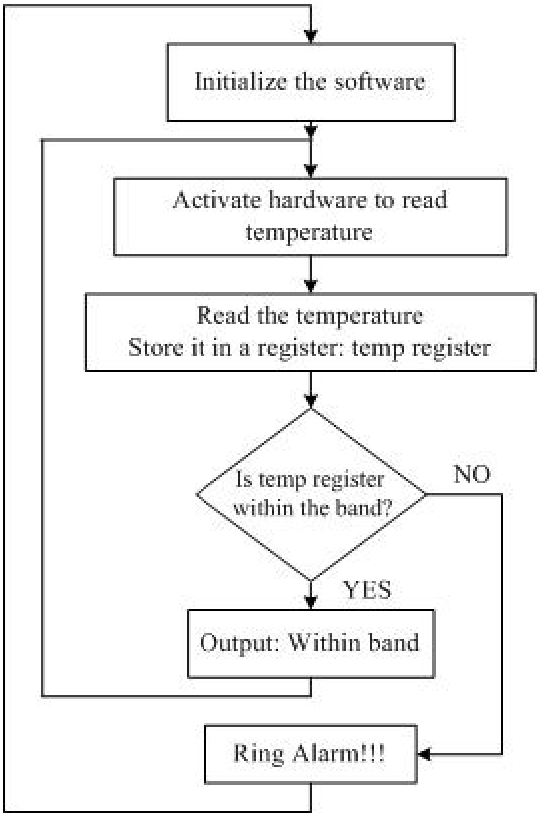
Let’s now briefly explore where the OS appears in the context of the software and application.



Let’s consider a scenario where we need to embed the computer system in an industrial application. This may be regulating the temperature of a vessel in a process control. In a typical process control scenario

- Monitoring – initializes and activates the hardware.
- Input – Reads the values from sensors and stores it in register.
- Decision – checks whether the readings are within the range.
- Output – responds to the situation.

- Scenario: A temperature monitoring chemical process.
- What we need: A supervisory program to raise an alarm when temperature goes beyond a certain band.
- The desired sequence of operational events: Measure input temperature, process the most recent measurement, perform an output task.


Initialize the software
Activate hardware to read temperature
Read the temperature
Store it in a register: temp register
Is temp register within the band?
NO
YES
Output: Within band
Ring Alarm!!!

The computer system may be employed in a variety of operational scenarios like a bank, airlines reservation system, university admissions and several others. In each of these we need to provide the resources for

- Processing
- User access to the system
- Storage and management of information
- Protection of information against accidental and intentional misuse
- Support for data processing activities

- Communication with I/O devices

- Management of all activities in a transparent manner.

Let’s now review What Does an OS Do?

- Power On Self Test (POST)

- Resource management

- Support for multi-user

- Error Handling

- Communication support over Network

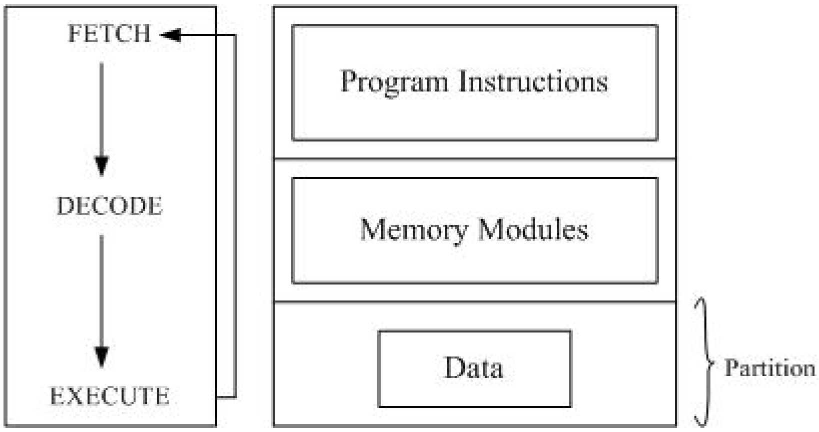
- (Optional) Deadline support so that safety critical application run and fail Gracefully

1.4 Operational View:

Let’s briefly look at the underlying principle of operation of a computer system. Current systems are based on The Von-Neumann principle. The principle states that a program is initially stored in memory and executed by fetching an instruction at a time.

The basic cycle of operation is

- ➢ Fetch an instruction (Fetch)
- ➢ Interpret the instruction (Decode)
- ➢ Execute the instruction (Execute)



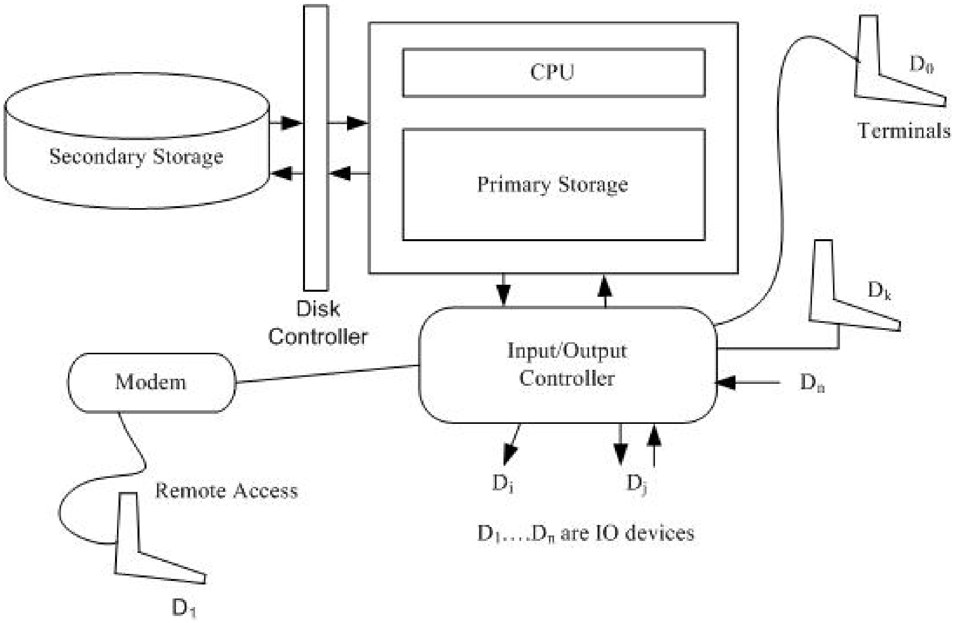
Operating Cycle Memory Map

Modern systems allow multiple users to use a computer system. Even on a stand-alone PC there may be multiple applications which are running simultaneously. For instance, we have a mail program receiving mails, a clock to display time while we may be engaged in browsing a word process.

In other words OS need to schedule the processor amongst all the application simultaneously without giving an impression that the processor time is being divided and scheduled per an application.

An Operational Overview:

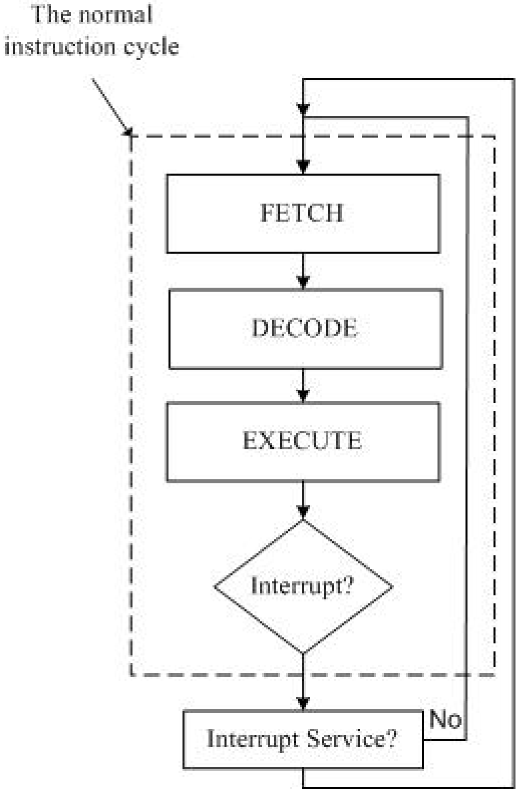
- Processor – schedule and allocate processor time.
- Memory – executes program and access data
- Input output devices
- Communication with devices
- Mutual exclusion – schedule the usage of shared device and fair access
- Shell of an OS
- Human computer interface (HCI/CHI)



A Modern Computer System

The peripheral devices communicate in a mode known as interrupt mode .Typically human input is considered important and often uses this mode. This is so because human desire to guide the operation. For instance, we use a mouse click or a key board input.

These require immediate attention by needing an appropriate interruption service.



1.5 Processes and Tools:

Processes:

Most OSs support a notion that a program in execution may be regarded as a process. Clearly, with multiple applications active at the same time there are many processes that are active. The OS needs to manage all these processes. Often applications may spawn processes that need to communicate with each other. Such inter process communication forms the primary basis of distributed computing.

With the coming together of communications network and computers it is possible to create processes on one machine and seek services from another machine. The machine seeking the services is called ***client machine*** and the machine offering services is called ***server machine***. When you use a web browser, your computer is the client and the machine which provides the content for browsing is the web server. All modern Oss support client-server operations for service provisioning.

Tools:

One important issue is: how does one use the services efficiently? In addition to the device handling utilities, most OSs provides many other general purpose utilities as a set of packaged tools. The tools help us to think in terms of higher level of operations. In the absence of tools and tool kits we would be required to write a fresh program each time.

As an example, consider a sort utility in UNIX. It supports identification of fields in a file with multi field structured data. Sort allows us to sort on a chosen field on a file. Imagine if we had to write a sort program every time we had a set of records making a file. Also look at the support we derive from the compression software tools like compression of files for long-term storage or transmission (Zip tools in windows environment). It would be stupendous task to write a compression program for every file transfer.

1.6 Trends in Computing:

The emergence of micro-computer, in particular Apple mackintosh, offered improvement in the manner in which human computer interfaces could be utilized for I/O and also for launching the applications. Apple was the first computer system to use mouse clicks to launch application.

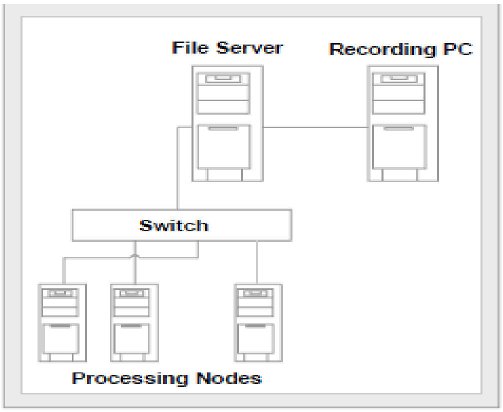
It also was the first environment to offer a "drag and drop" facility. In fact this set trends to offer icon driven convenient ways of launching applications.

Apple Mac Computer

In this section we shall examine other trends in computing. The application scenario can considerably influence the types of services that need to be managed by an operating system. One such example is the emergence of parallel computing.

1.7 Parallel Computing:

There are many problem solving situations like weather forecasting, image processing, statistical data analysis and simulation, pharmaceutical drug synthesis and others where using a single processor becomes a limiting factor For this class of problems one has to resort to parallel processing. In a parallel processing environment we may have a problem split in such a way that different parts of the same problem use a different processor with minimal communication. Parallel processing together with the network based services supports a mode of computing which is referred some times as distributed computing. In distributed computing environment it is also possible to support distribution of data, files and execution of processes.



Parallel Computing

In some other situation a computer system may be a part of a control system. For instance, the system may be utilized for regulating a utility – like electrical power supply. There is usually an elaborate instrumentation to keep track of the health of plant or a process. In these situations the instrumentation monitors physical quantities like pressure, temperature or speed and the process needs to respond in a real-time mode to be able to regulate the operation of the system.

1.8 Real - Time Systems:

Another category of real-time systems are those that are used to offer services like ATM (Automatic teller machine) or airlines reservation systems. The systems of the latter kind are often called transaction oriented systems. This is because the basic information processing involves a transaction.

As an example, consider processing a request like withdrawing money. It is a financial transaction, which requires updating accounts within an acceptable response time. Both the process control systems and transaction oriented systems are real time systems in which the response time to a request is critical. The main objective is to assure timeliness in response. The transaction oriented systems may even use the computer network extensively and often. The process control systems may be bus based or may have local area network as well. Both these systems invariably recognize an event and offer a response. In a transaction oriented system like ATM, it may be a card input. In a process control system an event may be detected when the temperature in a process vessel exceeds a certain limit. The operating system used in such real-time systems is often referred to as RTOS.

In addition, to the above kind of systems, we are now witnessing emergence of a class of systems that are embedded within an application.

For instance, it is not unusual to find up to four microprocessors in a car. These may be used for regulating fuel injection, cruise control or even operation of brakes. Similarly, there are microcontrollers embedded in washing machines. The presence of these systems

ATM

is totally transparent to the user. He does not experience the presence of the system while using the gadgets. This is because the operating environment only requires minimal control buttons and response panels. The embedded systems are designed completely with a different philosophy. These are not general purpose computing environments.

Instead, these have dedicated mode of operation. In these cases the operating system is not aimed at raising through put with general purpose utility back-ups. Instead these systems are designed with minimal and dedicated services. The minimal service provisioning is done using a minimal OS kernel often called micro-kernel.

A well known embedded system is a common mobile phone and a personal digital assistant (PDA). One important Indian contribution in this class of systems is a Simputer. A Simputer can be utilized as a PDA. Simputer can also be used in a dedicated application. Simputers have also been upgraded to support wireless connectivity.

Simputer

1.9 Wireless Systems:

Wireless systems in general allow access from anywhere any time. These are also called ubiquitous systems. The ubiquity comes from the fact that unlike a wired system, the medium of communication is air which can be utilized from anywhere anytime.

Finally, the other major trend which we are witnessing now is driven by Web – the World Wide Web. All modern systems are internet compliant and allow a user to connect to the rest of the world. Web offers commercial organizations to offer their merchandise on the web. Also, it gives the consumers an opportunity to seek services using the web. In our country now Web is quite extensively utilized for railway reservation and also for the air ticket booking. Web can also offer other services. For example, down loading music is common. Web can, and in due course of time will, offer services which are currently offered by operating systems. In fact, then we will have the paradigm where "network is the computer" as was proclaimed by the SUN CEO, Scott McNealy a few years ago.

Unit-2

1.1 File Systems and Management

In the previous module, we emphasized that a computer system processes and stores information. Usually, during processing computers need to frequently access primary memory for instructions and data. However, the primary memory can be used only for only temporary storage of information. This is so because the primary memory of a computer system is volatile. The volatility is evinced by the fact that when we switch off the power the information stored in the primary memory is lost. The secondary memory, on the other hand, is non-volatile. This means that once the user has finished his current activity on a computer and shut down his system, the information on disks (or any other form of secondary memory) is still available for a later access. The non-volatility of the memory enables the disks to store information indefinitely. Note that this information can also be made available online all the time. Users think of all such information as files. As a matter of fact, while working on a computer system a user is continually engaged in managing or using his files in one way or another. OS provides support for such management through a file system. File system is *the* software which empowers users and applications to organize and manage their files. The organization and management of files may involve access, updates and several other file operations. In this chapter our focus shall be on organization and management of files.

1.2 What Are Files?

Suppose we are developing an application program. A program, which we prepare, is a file. Later we may compile this program file and get an object code or an executable. The executable is also a file. In other words, the output from a compiler may be an object code file or an executable file. When we store images from a web page we get an image file. If we store some music in digital format it is an audio file. So, in almost every situation we are engaged in using a file. In addition, we saw in the previous module that files are central to our view of communication with IO devices. So let us now ask again:

What is a file?

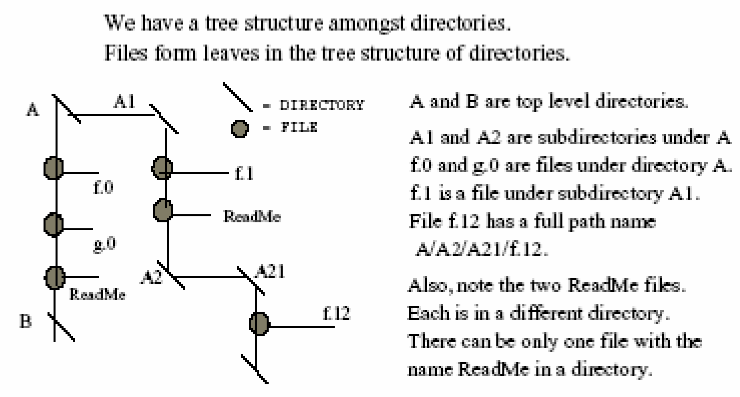
Irrespective of the content any organized information is a file. So be it a telephone numbers list or a program or an executable code or a web image or a data logged from an instrument we think of it always as a file. This formlessness and disassociation from content was emphasized first in

Unix. The formlessness essentially means that files are arbitrary bit (or byte) streams. Formlessness in Unix follows from the basic design principle: keep it simple. The main advantage to a user is flexibility in organizing files. In addition, it also makes it easy to design a file system. A file system is that software which allows users and applications to organize their files. The organization of information may involve access, updates and movement of information between devices. Later in this module we shall examine the user view of organizing files and the system view of managing the files of users and applications. We shall first look at the user view of files.

User's view of files: The very first need of a user is to be able to access some file he has stored in a non-volatile memory for an on-line access. Also, the file system should be able to locate the file sought by the user. This is achieved by associating an identification for a file i.e. a file must have a name. The name helps the user to identify the file. The file name also helps the file system to locate the file being sought by the user. Let us consider the organization of my files for the Compilers course and the Operating Systems course on the web. Clearly, all files in compilers course have a set of pages that are related. Also, the pages of the OS system course are related. It is, therefore, natural to think of organizing the files of individual courses together. In other words, we would like to see that a file system supports grouping of related files. In addition, we would like that all such groups be put together under some general category (like COURSES).

This is essentially like making one file folder for the compilers course pages and other one for the OS course pages. Both these folders could be placed within another folder, say COURSES. This is precisely how MAC OS defines its folders. In Unix, each such group, with related files in it, is called a directory. So the COURSES directory may have subdirectories OS and COMPILERS to get a hierarchical file organization. All modern OSs support such a hierarchical file organization. In Figure 2.1 we show a hierarchy of files. It must be noted that within a directory each file must have a distinct name. For instance, I tend to have ReadMe file in directories to give me the information on what is in each directory. At most there can be only one file with the name “ReadMe" in a directory. However, every subdirectory under this directory may also have its own ReadMe file. Unix emphasizes disassociation with content and form. So file names can be assigned any way.

Some systems, however, require specific name extensions to identify file type. MSDOS identifies executable files with a .COM or .EXE file name extension. Software systems like C or Pascal compilers expect file name extensions of .c or .p (or .pas) respectively. In



We have a tree structure amongst directories.
Files form leaves in the tree structure of directories.

A and B are top level directories.

A1 and A2 are subdirectories under A
f.0 and g.0 are files under directory A.
f.1 is a file under subdirectory A1.
File f.12 has a full path name
A/A2/A21/f.12.

Also, note the two ReadMe files.
Each is in a different directory.
There can be only one file with the
name ReadMe in a directory.

Figure 2.1: Directory and file organisation.

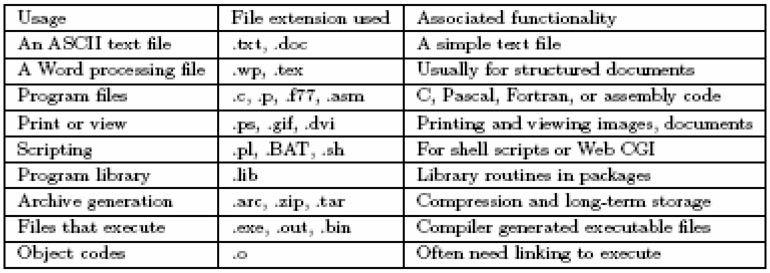
Section 1.2.1 and others we see some common considerations in associating a file name extension to define a file type.

1.2.1 File Types and Operations

Many OSs, particularly those used in personal computers, tend to use a file type information within a name. Even Unix software support systems use standard file extension names, even though Unix as an OS does not require this. Most PC-based Oss associate file types with specific applications that generate them. For instance, a database generating program will leave explicit information with a file descriptor that it has been generated by a certain database program. A file descriptor is kept within the file structure and is often used by the file system software to help OS provide file management services. MAC OS usually stores this information in its resource fork which is a part of its file descriptors.

This is done to let OS display the icons of the application environment in which this file was created. These icons are important for PC users. The icons offer the operational clues as well. In Windows, for instance, if a file has been created using *notepad* or *word* or has been stored from the browser, a corresponding give away icon appears. In fact, the OS assigns it a file type. If the icon has an Adobe sign on it and we double click on it the acrobat reader opens it right away. Of course, if we choose to open any of the files differently, the OS provides us that as a choice (often using the right button).

For a user the extension in the name of a file helps to identify the file type. When a user has a very large number of files, it is very helpful to know the type of a file from its name extensions. In Table 2.1, we have many commonly used file name extensions. PDP-11 machines, on which Unix was originally designed, used an octal 0407 as a magic number to identify its executable files. This number actually was a machine executable jump instruction which would simply set the program counter to fetch the first executable instruction in the file. Modern systems use many magic numbers to identify which application created or will execute a certain file.



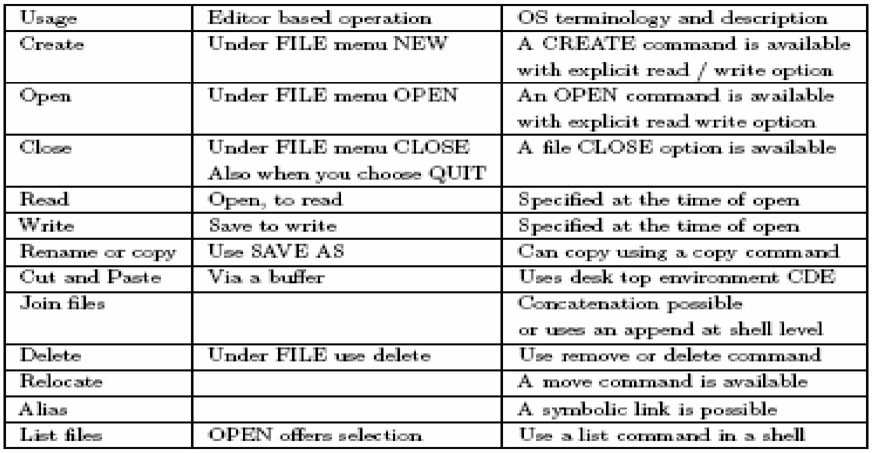
Usage	File extension used	Associated functionality
An ASCII text file	.txt, .doc	A simple text file
A Word processing file	.wp, .tex	Usually for structured documents
Program files	.c, .p, .f77, .asm	C, Pascal, Fortran, or assembly code
Print or view	.ps, .gif, .dvi	Printing and viewing images, documents
Scripting	.pl, .BAT, .sh	For shell scripts or Web CGI
Program library	.lib	Library routines in packages
Archive generation	.arc, .zip, .tar	Compression and long-term storage
Files that execute	.exe, .out, .bin	Compiler generated executable files
Object codes	.o	Often need linking to execute

Table 2.1: File extension and its context of use.

In addition to the file types, a file system must have many other pieces of information that are important. For instance, a file system must know at which location a file is placed in the disk, it should know its size, when was it created, i.e. date and time of creation. In addition, it should know who owns the files and who else may be permitted access to *read*, *write* or *execute*. We shall next dwell upon these operational issues.

File operations: As we observed earlier, a file is any organized information. So at that level of abstraction it should be possible for us to have some logical view of files, no matter how these may be stored. Note that the files are stored within the secondary storage. This is a physical view of a file. A file system (as a layer of software) provides a logical view of files to a user or to an application. Yet, at another level the file system offers the physical view to the OS. This means that the OS gets all the information it needs to physically locate, access, and do other file based operations whenever needed. Purely from an operational point of view, a user should be able to create a file. We will also assume that the creator owns the file. In that case he may wish to save or store this file. He should be able to read the contents of the file or even write into this file. Note that a user needs the write capability to update a file. He may wish to display or rename or append this file. He may even wish to make another copy or even delete this file. He may even wish to operate with two or more files. This may entail cut or copy from one file and paste information on the other.

Other management operations are like indicating who else has an authorization of an access to *read* or *write* or *execute* this file. In addition, a user should be able to move this file between his directories. For all of these operations the OS provides the services. These services may even be obtained from within an application like mail or a utility such as an editor. Unix provides a visual editor vi for ASCII file editing. It also provides another editor *sed* for stream editing. MAC OS and PCs provide a range of editors like SimpleText.



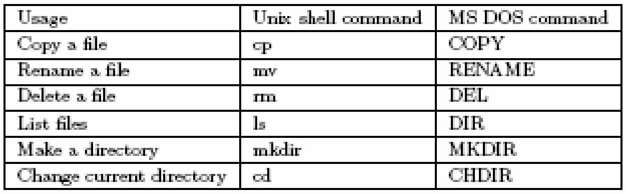
Usage	Editor based operation	OS terminology and description
Create	Under FILE menu NEW	A CREATE command is available with explicit read / write option
Open	Under FILE menu OPEN	An OPEN command is available with explicit read write option
Close	Under FILE menu CLOSE Also when you choose QUIT	A file CLOSE option is available
Read	Open, to read	Specified at the time of open
Write	Save to write	Specified at the time of open
Rename or copy	Use SAVE AS	Can copy using a copy command
Cut and Paste	Via a buffer	Uses desk top environment CDE
Join files		Concatenation possible or uses an append at shell level
Delete	Under FILE use delete	Use remove or delete command
Relocate		A move command is available
Alias		A symbolic link is possible
List files	OPEN offers selection	Use a list command in a shell

Table 2.2: File operations.

With multimedia capabilities now with PCs we have editors for audio and video files too. These often employ MIDI capabilities. MAC OS has Claris works (or Apple works) and MSDOS-based systems have Office 2000 suite of packaged applications which provide the needed file oriented services. See Table 2.2 for a summary of common file operations.

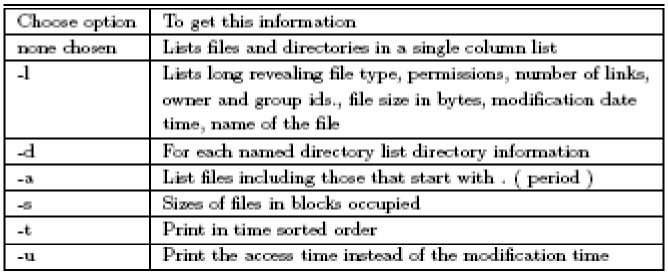
For illustration of many of the basic operations and introduction of shell commands we shall assume that we are dealing with ASCII text files. One may need information on file sizes. More particularly, one may wish to determine the number of lines, words or characters in a file. For such requirements, a shell may have a suite of word counting programs. When there are many files, one often needs longer file names. Often file names may bear a common stem to help us categorize them. For instance, I tend to use “prog” as a prefix to identify my program text files. A programmer derives considerable support through use of regular expressions within file names. Use of regular expressions enhances programmer productivity in checking or accessing file names. For instance, prog* will mean all files prefixed with stem prog, while my file? may mean all the files with prefix my file followed by at most one character within the current directory. Now that we have seen the file operations, we move on to services. Table 2.3 gives a brief description of the file-oriented services that are made available in a Unix OS. There are similar MS DOS commands. It is a very rewarding experience to try these commands and use regular expression operators like ? and * in conjunction with these commands.

Later we shall discuss some of these commands and other file-related issues in greater depth. Unix, as also the MS environment, allows users to manage the organization of their files. A command which helps viewing current status of files is the *ls* command in



Usage	Unix shell command	MS DOS command
Copy a file	cp	COPY
Rename a file	mv	RENAME
Delete a file	rm	DEL
List files	ls	DIR
Make a directory	mkdir	MKDIR
Change current directory	cd	CHDIR

Table 2.3: File oriented services.



Choose option	To get this information
none chosen	Lists files and directories in a single column list
-l	Lists long revealing file type, permissions, number of links, owner and group ids., file size in bytes, modification date time, name of the file
-d	For each named directory list directory information
-a	List files including those that start with . (period)
-s	Sizes of files in blocks occupied
-t	Print in time sorted order
-u	Print the access time instead of the modification time

Table 2.4: Unix *ls* command options.

Unix (or the *dir* command in MS environment). This command is very versatile. It helps immensely to know various facets and usage options available under the *ls* command. The *ls* command: Unix's *ls* command which lists files and subdirectories in a directory is very revealing. It has many options that offer a wealth of information. It also offers an insight in to what is going on with the files i.e. how the file system is updating the information about files in “inode” which is a short form for an index node in Unix. We shall learn more about inode in Section 2.4. In fact, it is very rewarding to study *ls* command in all its details. Table 2.4 summarizes some of the options and their effects.

Using regular expressions: Most operating systems allow use of regular expression operators in conjunction with the commands. This affords enormous flexibility in usage of a command. For instance, one may input a partial pattern and complete the rest by a * or a ? operator. This not only saves on typing but also helps you when you are searching a file after a long time gap and you do not remember the exact file names completely. Suppose a directory has files with names like Comp_page_1.gif, Comp_page_2.gif and Comp_page_1.ps and Comp_page_2.ps. Suppose you wish to list files for page_2. Use a partial name like *ls* C*p*2 or even *2* in *ls* command. We next illustrate the use of operator ?. For instance, use of *ls* my file? in *ls* command will list all files in the current directory with prefix my file followed by at most one character.

Besides these operators, there are command options that make a command structure very flexible. One useful option is to always use the *-i* option with the *rm* command in Unix. A *rm -i* my files* will interrogate a user for each file with prefix my file for a possible removal. This is very useful, as by itself *rm* my file* will remove all the files without any further prompts and this can be very dangerous. A powerful command option within the *rm* command is to use a *-r* option. This results in recursive removal, which means it removes all the files that are linked within a directory tree. It would remove files in the current, as well as, subdirectories all the way down. One should be careful in choosing the options, particularly for remove or delete commands, as information may be lost irretrievably.

It often happens that we may need to use a file in more than one context. For instance, we may need a file in two projects. If each project is in a separate directory then we have two possible solutions. One is to keep two copies, one in each directory or to create a symbolic link and keep one copy. If we keep two unrelated copies we have the problem of consistency because a change in one is not reflected in the other. The symbolic link helps to alleviate this problem. Unix provides the *ln* command to generate a link anywhere regardless of directory locations with the following structure and interpretation: *ln* fileName pseudonym.

Now fileName file has an alias in pseudonym too. Note that the two directories which share a file link should be in the same disk partition. Later, in the chapter on security, we shall observe how this simple facility may also become a security hazard.

1.2 File Access Rights

After defining a fairly wide range of possible operations on files we shall now look at the file system which supports all these services on behalf of the OS. In the preamble of this chapter we defined a file system as that software which allows users and applications to organize and manage their files. The organization of information may involve access, updates, and movement of information between devices. Our first major concern is access.

Access permissions: Typically a file may be accessed to read or write or execute. The usage is determined usually by the context in which the file is created. For instance, a city bus timetable file is created by a transport authority for the benefit of its customers. So this file may be accessed by all members of public. While they can access it for a read operation, they cannot write into it. An associated file may be available to the supervisor who assigns duties to drivers. He can, not only read but also write in to the files that assign drivers to bus routes. The management at the transport authority can read, write and even execute some files that generate the bus schedules. In other words, a file system must manage access by checking the access rights of users. In general, access is managed by keeping access rights information for each file in a file system.

Who can access files?: Unix recognizes three categories of users of files, e.g. user (usually the user who created it and owns it), the group, and others. The owner may be a person or a program (usually an application or a system-based utility). The notion of “group” comes from software engineering and denotes a team effort. The basic concept is that users in a group may share files for a common project. Group members often need to share files to support each other's activity. Others has the connotation of public usage as in the example above. Unix organizes access as a three bit information for each i.e. owner, group, and others. So the access rights are defined by 9 bits as *rwx rwx rwx* respectively for owner, group and others. The *rwx* can be defined as an octal number too. If all bits are set then we have a pattern 111 111 111 (or 777 in octal) which means the owner has read, write, and execute rights, and the group to which he belongs has also read, write and execute rights, and others have read, write and execute rights as well. A pattern of 111 110 100 (or 764 octal, also denoted as *rwx rw- r--*) means the owner has read, write, and execute permissions; the group has read and write permissions but no execute permission and others have only the read permission. Note that Unix grouppermissions are for all or none. Windows 2000 and NTFS permit a greater degree ofrefinement on a group of users. Linux allows individual users to make up groups.

1.4 File Access and Security Concerns

The owner of a file can alter the permissions using the *chmod* command in Unix. The commonly used format is *chmod* octalPattern fileName which results in assigning the permission interpreted from the octalPattern to the file named fileName. There are other alternatives to *chmod* command like *chmod* changePattern fileName where changePattern may be of the form *go-rw* to denote withdrawal of read write permission from group and others. Anyone can view all the currently applicable access rights using a *ls* command in Unix with *-l* option. This command lists all the files and subdirectories of the current directory with associated access permissions.

Security concerns: Access permissions are the most elementary and constitute a fairly effective form of security measure in a standalone single user system. In a system which may be connected in a network this can get very complex. We shall for now regard the access control as our first line of security. On a PC which is a single-user system there is no security as such as anyone with an access to the PC has access to all the files.

Windows 2000 and XP systems do permit access restriction amongst all the users of the system. These may have users with system administrator privileges. In Unix too, the super-user (root) has access to all the files of all the users. So there is a need for securing files for individual users. Some systems provide security by having a password for files. However, an enhanced level of security is provided by encryption of important files. Most systems provide some form of encryption facility. A user may use his own encryption key to encrypt his file. When someone else accesses an encrypted file he sees a garbled file which has no pattern or meaning. Unix provides a crypt command to encrypt files.

The format of the crypt command is:

```
crypt EncryptionKey < inputFileName > outputFileName
```

The EncryptionKey provides a symmetric key, so that you can use the same key to retrieve the old file (simply reverse the roles of inputFileName and outputFileName) In Section 1.5 we briefly mention about audit trails which are usually maintained in syslog files in Unix systems. In a chapter on security we shall discuss these issues in detail. So far we have dealt with the logical view of a file. Next, we shall address the issues involved in storage and management of files.

1.5 File Storage Management

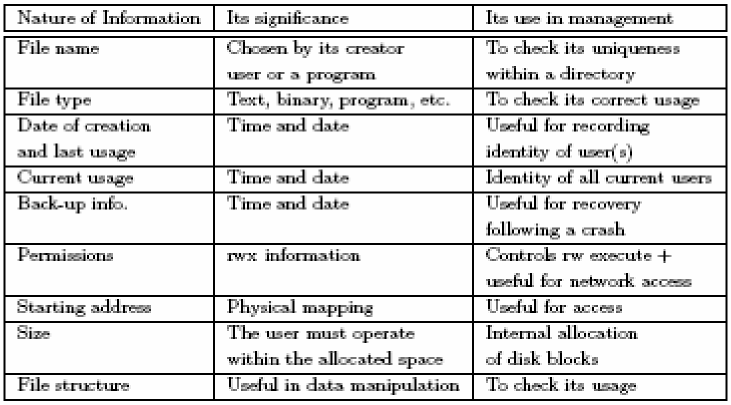
An operating system needs to maintain several pieces of information that can assist in management of files. For instance, it is important to record when the file was last used and by whom. Also, which are the current processes (recall a process is a program in execution) accessing a particular file. This helps in management of access. One of the important files from the system point of view is the audit trail which indicates who accessed when and did what. As mentioned earlier, these trails are maintained in syslog files under Unix. Audit trail is very useful in recovering from a system crash. It also is useful to detect un-authorized accesses to the system. There is an emerging area within the security community which looks up the audit trails for clues to determine the identity of an intruder.

In Table 2.5 we list the kind of information which may be needed to perform proper file management. While Unix emphasizes formlessness, it recognizes four basic file types internally. These are ordinary, directory, special, and named. Ordinary files are those that are created by users, programs or utilities. Directory is a file type that organizes files hierarchically, and the system views them differently from ordinary files. All IO communications are conducted as communications to and from special files. For the present we need not concern ourselves with named files. Unix maintains much of this information in a data structure called inode which is a short form for an index node. All file management operations in Unix are controlled and maintained by the information in the inode structure.

We shall now briefly study the structure of inode.

1.5.1 Inode in Unix

In Table 2.6 we describe typical inode contents. Typically, it offers all the information about access rights, file size, its date of creation, usage and modification. All this information is useful for the management in terms of allocation of physical space, securing information from malicious usage and providing services for legitimate user needs to support applications.



Nature of Information	Its significance	Its use in management
File name	Chosen by its creator user or a program	To check its uniqueness within a directory
File type	Text, binary, program, etc.	To check its correct usage
Date of creation and last usage	Time and date	Useful for recording identity of user(s)
Current usage	Time and date	Identity of all current users
Back-up info.	Time and date	Useful for recovery following a crash
Permissions	rwx information	Controls rw execute + useful for network access
Starting address	Physical mapping	Useful for access
Size	The user must operate within the allocated space	Internal allocation of disk blocks
File structure	Useful in data manipulation	To check its usage

Table 2.5: Information required for management of files.

Typically, a disk shall have inode tables which point to data blocks. In Figure 2.2 we show how a disk may have data and inode tables organized. We also show how a typical Unix-based system provides for a label on the disk.

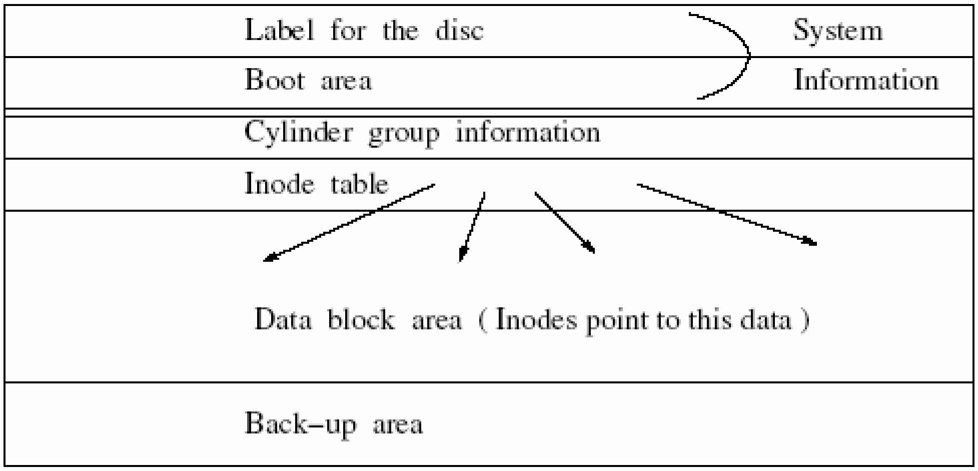
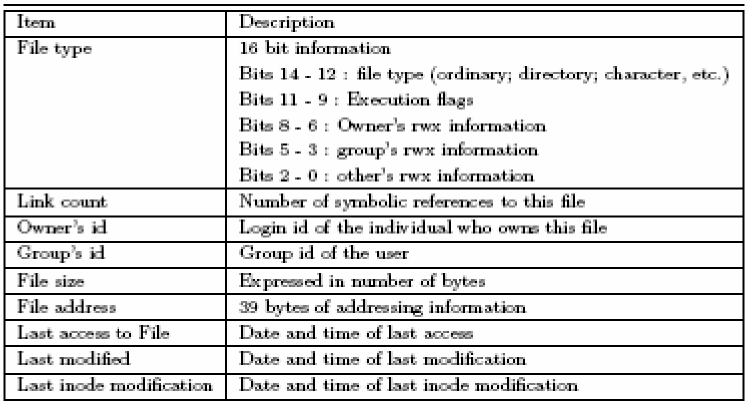

Label for the disc
Boot area
System Information
Cylinder group information
Inode table
Data block area (Inodes point to this data)
Back–up area

Figure 2.2: Organisation of inodes.



Item	Description
File type	16 bit information Bits 14 - 12 : file type (ordinary; directory; character, etc.) Bits 11 - 9 : Execution flags Bits 8 - 6 : Owner's rwx information Bits 5 - 3 : group's rwx information Bits 2 - 0 : other's rwx information
Link count	Number of symbolic references to this file
Owner's id	Login id of the individual who owns this file
Group's id	Group id of the user
File size	Expressed in number of bytes
File address	39 bytes of addressing information
Last access to File	Date and time of last access
Last modified	Date and time of last modification
Last inode modification	Date and time of last inode modification

Table 2.6: Inode structure in Unix.

1.5.2 File Control Blocks

In MS environment the counterpart of inode is FCB, which is a short form for File Control Block. The FCBs store file name, location of secondary storage, length of file in bytes, date and time of its creation, last access, etc. One clear advantage MS has over Unix is that it usually maintains file type by noting which application created it. It uses extension names like *doc, txt, dll*, etc. to identify how the file was created. Of course, notepad may be used to open any file (one can make sense out of it when it is a text file). Also, as we will see later (in Sections 1.7 and 1.8), MS environment uses a simple chain of clusters which is easy to manage files.

1.6 The Root File System

At this stage it would be worthwhile to think about the organization and management of files in the root file system. When an OS is installed initially, it creates a root file system. The OS not only ensures, but also specifies how the system and user files shall be distributed for space allocation on the disk storage. Almost always the root file system has a directory tree structure. This is just like the users file organization which we studied earlier in Figure 2.1. In OSs with Unix flavors the root of the root file system is a directory. The root is identified by the directory `/'. In MS environment it is identified by `n'. The root file system has several subdirectories. OS creates disk partitions to allocate files for specific usages. A certain disk partition may have system files and some others may have other user files or utilities. The system files are usually programs that are executable with .bin in Unix and .EXE extension in MS environment.

Under Unix the following convention is commonly employed.

- Subdirectory usr contain shareable binaries. These may be used both by users and the system. Usually these are used in read-only mode.
- Under subdirectories bin (found at any level of directory hierarchy) there are executables. For instance, the Unix commands are under /usr/bin. Clearly, these are shareable executables.
- Subdirectory sbin contains some binaries for system use. These files are used during boot time and on power-on.
- Subdirectories named lib anywhere usually contain libraries. A lib subdirectory may appear at many places. For example, as we explain a little later the graphics library which

supports the graphics user interface (GUI) uses the X11 graphics library, and there shall be a lib subdirectory under directory X11.

- Subdirectory etc contains the host related files. It usually has many subdirectories to store device, internet and configuration related information. Subdirectory hosts stores internet addresses of hosts machines which may access this host. Similarly, config subdirectory maintains system configuration information and inet subdirectory maintains internet configuration related information. Under subdirectory dev, we have all the IO device related utilities.
- Subdirectories mnt contain the device mount information (in Linux).
- Subdirectories tmp contain temporary files created during file operation. When you use an editor the OS maintains a file in this directory keeping track of all the edits. Clearly this is its temporary residence.
- Subdirectories var contain files which have variable data. For instance, mail and system log information keeps varying over time. It may also have subdirectories for spools. Spools are temporary storages. For instance, a file given away for printing may be spooled to be picked up by the printer. Even mails may be spooled temporarily.
- All X related file support is under a special directory X11. One finds all X11 library files under a lib directory within this directory.
- A user with name u name would find that his files are under /home/u name. This is also the home directory for the user u name.
- Subdirectories include contain the C header include files.
- A subdirectory marked as yp (a short form for yellow pages) has network information. Essentially, it provides a database support for network operations.

One major advantage of the root file system is that the system knows exactly where to look for some specific routines. For instance, when we give a command, the system looks for a path starting from the root directory and looks for an executable file with the command name specified (usually to find it under one of the bin directories). Users can customize their operational environment by providing a definition for an environment variable PATH which guides the sequence in which the OS searches for the commands. Unix, as also the MS environment, allows users to manage the organization of their files.

One of the commands which helps to view the current status of files is the *ls* command inUnix or the command dir in MS environment.

1.7 Block-based File Organization

Recall we observed in chapter 1 that disks are bulk data transfer devices (as opposed to character devices like a keyboard). So data transfer takes place from disks in blocks as large as 512 or 1024 bytes at a time. Any file which a user generates (or retrieves), therefore, moves in blocks. Each operating system has its own block management policy. We shall study the general principles underlying allocation policies. These policies map each linear byte stream into disk blocks. We consider a very simple case where we need to support a file system on one disk. Note a policy on storage management can heavily influence the performance of a file system (which in turn affects the throughput of an OS). File Storage allocation policy: Let us assume we know apriori the sizes of files before their creation. So this information can always be given to OS before a file is created. Consequently, the OS can simply make space available. In such a situation it is possible to follow a pre-allocation policy: find a suitable starting block so that the file can be accommodated in a contiguous sequence of disk blocks. A simple solution would be to allocate a sequence of contiguous blocks as shown in Figure 2.3.

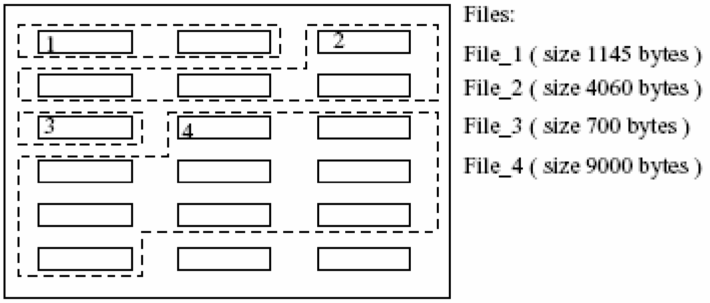

1
2
3
4
Files:
File_1 (size 1145 bytes)
File_2 (size 4060 bytes)
File_3 (size 700 bytes)
File_4 (size 9000 bytes)

Figure 2.3: Contiguous allocation.

The numbers 1, 2, 3 and 4 denote the starting blocks for the four files. One clear advantage of such a policy is that the retrieval of information is very fast. However, note that pre-allocation policy requires apriori knowledge. Also, it is a static policy. Often users' needs develop over time and files undergo changes. Therefore, we need a dynamic policy.

Chained list Allocation: There are two reasons why a dynamic block allocation policy is needed. The first is that in most cases it is not possible to know apriori the size of a file being created. The second is that there are some files that already exist and it is not easy to find contiguous regions. For instance, even though there may be enough space in the disk, yet it may not be possible to find a single large enough chunk to accommodate an incoming file. Also, users' needs evolve and a file during its lifetime undergoes changes. Contiguous blocks leave no room for such changes. That is because there may be already allocated files occupying the contiguous space.

In a dynamic situation, a list of free blocks is maintained. Allocation is made as the need arises. We may even allocate one block at a time from a free space list. The OS maintains a chain of free blocks and allocates next free block in the chain to an incoming file. This way the finally allocated files may be located at various positions on the disk. The obvious overhead is the maintenance of chained links. But then we now have a dynamically allocated disk space. An example is shown in Figure 2.4.

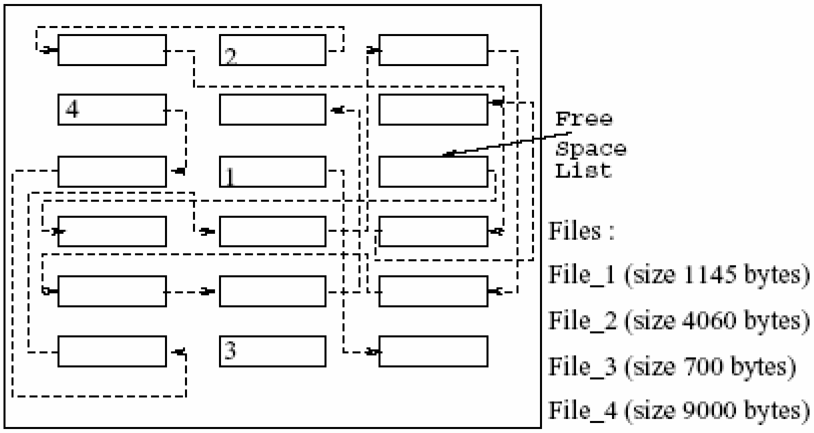


Figure 2.4: Chained allocation.

Chained list allocation does not require apriori size information. Also, it is a dynamic allocation method. However, it has one major disadvantage: random access to blocks is not possible.

Indexed allocation: In an indexed allocation we maintain an index table for each file in its very first block. Thus it is possible to obtain the address information for each of the blocks with only one level of indirection, i.e. from the index. This has the advantage that there is a direct access to every block of the file. This means we truly operate in the direct access mode at the block level.

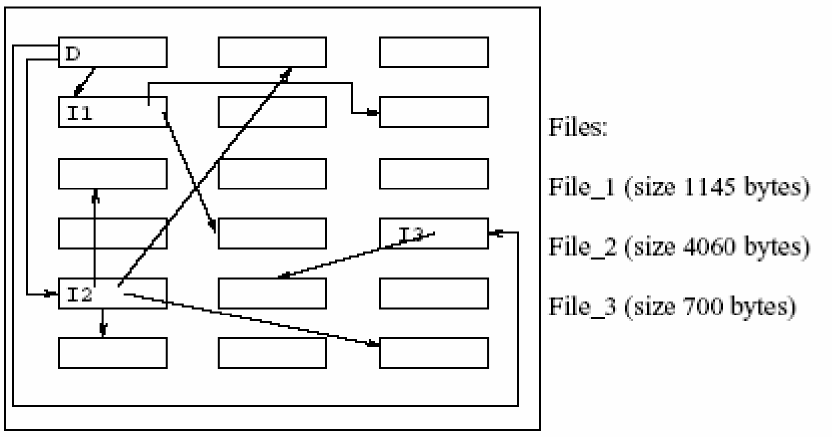


Figure 2.5: Indexed allocation.

In Figure 2.5 we see that File-2 occupies four blocks. Suppose we use a block I2 to store the starting addresses of these four blocks, then from this index we can access any of the four parts of this file. In a chained list arrangement we would have to traverse the links. In Figure 2.5 we have also shown D to denote the file's current directory. All files have their own index blocks. In terms of storage the overhead of storing the indices is more than the overhead of storing the links in the chained list arrangements. However, the speed of access compensates for the extra overhead.

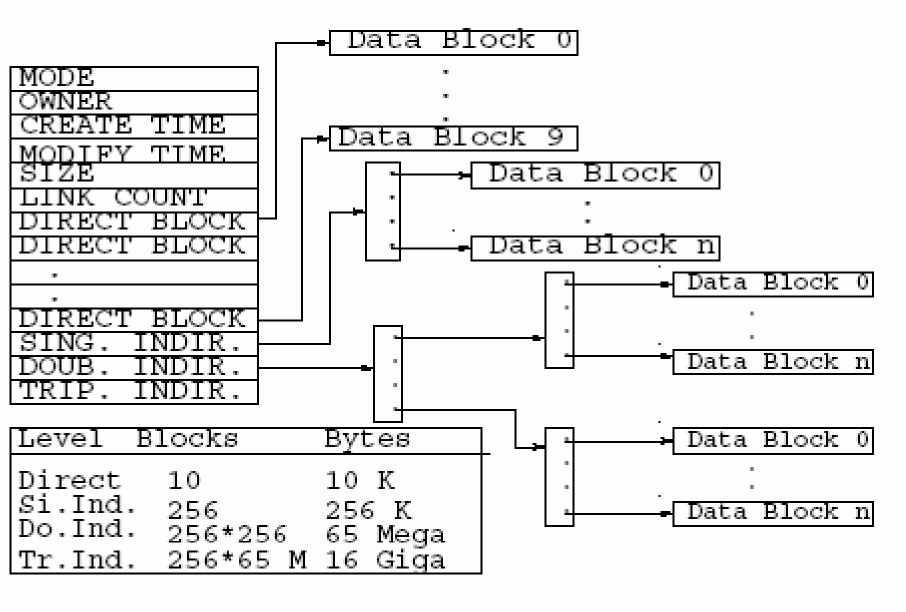
Internal and external Fragmentation: In mapping byte streams to blocks we assumed a block size of 1024 bytes. In our example, a file (File 1) of size 1145 bytes was allocated two blocks. The two blocks together have 2048 bytes capacity. We will fill the first block completely but the second block will be mostly empty. This is because only 121 bytes out of 1024 bytes are used. As the assignment of storage is by blocks in size of 1024 bytes the remaining bytes in the second block cannot be used. Such non-utilization of space caused internally (as it is within a file's space) is termed as internal fragmentation. We note that initially the whole disk is a free-space list of connected blocks. After a number of file insertions and deletion or modifications the free-space list becomes smaller in size. This can be explained as follows. For instance, suppose we have a file which was initially spread over 7 blocks. Now after a few edits the file needs only 4 blocks. This space of 3 blocks which got released is now not connected anywhere. It is not connected with the free storage list either. As a result, we end up with a hole of 3 blocks which is not connected anywhere. After many file edits and operations many such holes of various sizes get created. Suppose we now wish to insert a moderately large sized file thinking that adequate space should be still available. Then it may happen that the free space list has shrunk so much that enough space is not available. This may be because there are many unutilized holes in the disk. Such non-utilization, which is outside of file space, is regarded as external fragmentation. A file system, therefore, must periodic all perform an operation to rebuild free storage list by collecting all the unutilized holes and linking them back to free storage list. This process is called compaction. When you boot a system, often the compaction gets done automatically. This is usually a part of file system management check. Some run-time systems, like LISP and Java, support periodic automatic compaction. This is also referred to as run-time garbage collection.

1.8 Policies In Practice

MS DOS and OS2 (the PC-based systems) use a FAT (file allocation table) strategy. FAT is a table that has entries for files for each directory. The file name is used to get the starting address of the first block of a file. Each file block is chain linked to the next block till an EOF (end of file) is stored in some block. MS uses the notion of a cluster in place of blocks, i.e. the concept of cluster in MS is same as that of blocks in Unix. The cluster size is different for different sizes of disks. For instance, for a 256 MB disk the cluster may have a size of 4 KB and for a disk with size of 1 GB it may be 32 KB. The formula used for determining the cluster size in MS environment is disk-size/64K. FAT was created to keep track of all the file entries. To that extent it also has the information similar to the index node in Unix. Since MS environment uses chained allocation, FAT also maintains a list of “free" block chains. Earlier, the file names under MS DOS were restricted to eight characters and a three letter extension often indicating the file type like BAT or EXE, etc. Usually FAT is stored in the first few blocks of disk space.

An updated version of FAT, called FAT32, is used in Windows 98 and later systems. FAT32 additionally supports longer file names and file compression. File compression may be used to save on storage space for less often used files. Yet another version of the Windows is available under the Windows NT. This file system is called NTFS. Rather than having one FAT in the beginning of disk, the NTFS file system spreads file tables throughout the disks for efficient management. Like FAT32, it also supports long file names and file compression. Windows 2000 uses NTFS. Other characteristics worthy of note are the file access permissions supported by NTFS.

Unix always supported long file names and most Unix based systems such as Solaris and almost all Linux versions automatically compress the files that have not been used for long. Unix uses indexed allocation. Unix was designed to support truly large files. We next describe how large can be large files in Unix. Unix file sizes: Unix was designed to support large-scale program development with team effort. Within this framework, it supports group access to very large files at very high speeds. It also has a very flexible organization for files of various sizes. The information about files is stored in two parts. The first part has information about the mode of access, the symbolic links, owner and times of creation and last modification.



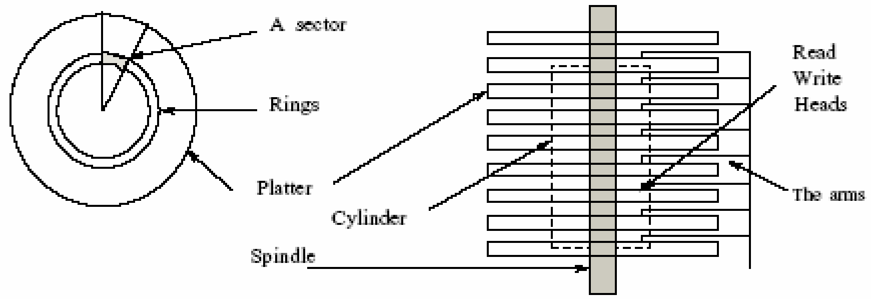
Level	Blocks	Bytes
Direct	10	10 K
Si.Ind.	256	256 K
Do.Ind.	256*256	65 Mega
Tr.Ind.	256*65 M	16 Giga

Figure 2.6: Storage allocation in Unix.

The second part is a 39 byte area within the inode structure. These 39 bytes are 13, 3 byte address pointers. Of the 39 bytes, first 10 point to the first 10 blocks of a file. If the files are longer then the other 3, 3 byte addresses are used for indirect indexing. So the 11th 3 byte address points to a block that has pointers to real data. In case the file is still larger then the 12th 3 byte address points to an index. This index in turn points to another index table which finally point to data. If the files are still larger then the 13th 3 byte address is used to support a triple indirect indexing. Obviously, Unix employs the indexed allocation. In Figure 2.6 we assume a data block size of 1024 bytes. We show the basic scheme and also show the size of files supported as the levels of indirection increase.

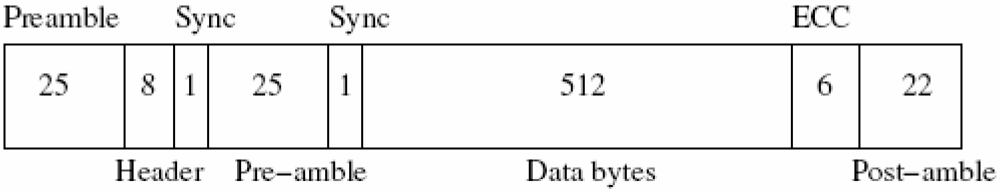
Physical Layout of Information on Media: In our discussions on file storage and management we have concentrated on logical storage of files. We, however, ignored one very important aspect. And that concerns the physical layout of information on the disk media. Of course, we shall revisit aspects of information map on physical medium later in the chapter on IO device management. For now, we let us examine Figures 2.7 and 2.8 to see how information is stored, read, and written in to a disk.

In Figure 2.7, tracks may be envisaged as rings on a disk platter. Each ring on a platter is capable of storing 1 bit along its width. These 1 bit wide rings are broken into sectors, which serve as blocks. In Section 2.6 we essentially referred to these as blocks.



Note that the rings on the disk platters on the spindle form a cylinder. Since all heads are on a particular ring at the same time, so it is easy to organise information on a cylinder. The information is stored in the sectors that can be identified on the rings. sectors are seperated from each other. All sectors can hold equal amount of information.

Figure 2.7: Information storage organisation on disks.



The numbers are in bytes

Figure 2.8: Information storage in sectors.

This break up into sectors is necessitated because of the physical nature of control required to let the system recognize, where within the tracks blocks begin in a disk. With disks moving at a very high speed, it is not possible to identify individual characters as they are laid out. Only the beginning of a block of information can be detected by hardware control to initiate a stream of bits for either input or output. The read-write heads on the tracks read or write a stream of data along the track in the identified sectors. With multiple disks mounted on a spindle as shown in Figure 2.7, it helps to think of a

Cylinder formed by tracks that are equidistant from the center. Just imagine a large number of tracks, one above the other, and you begin to see a cylinder. These cylinders can be given contiguous block sequence numbers to store information. In fact, this is desirable because then one can access these blocks in sequence without any additional head movement in a head per track disk. The question of our interest for now is: where is inode (or FAT block) located and how it helps to locate the physical file which is mapped on to sectors on tracks which form cylinders.

1.8.1 Disk Partitions

Disk-partitioning is an important notion. It allows a better management of disk space. The basic idea is rather simple. If you think of a disk as a large space then simply draw some boundaries to keep things in specific areas for specific purposes. In most cases the disk partitions are created at the time the disc is formatted. So a formatted disk has information about the partition size.

In Unix oriented systems, a physical partition of a disk houses a file system. Unix also allows creating a logical partition of disk space which may extend over multiple disk drives. In either case, every partition has its own file system management information. This information is about the files in that partition which populate the file system. Unix ensures that the partitions for the system kernel and the users files are located in different partitions (or file systems). Unix systems identify specific partitions to store the root file system, usually in root partition. The root partition may also co-locate other system functions with variable storage requirements which we discussed earlier in section 1.6. The user files may be in another file system, usually called home. Under Linux, a proc houses all the executable processes.

Under the Windows system too, a hard disk is partitioned. One interesting conceptual notion is to make each such partition that can be taken as a logical drive. In fact, one may have one drive and by partitioning, a user can make the OS offer a possibility to write into each partition as if it was writing in to a separate drive. There are many third-party tools for personal computer to help users to create partitions on their disks. Yet another use in the PC world is to house two operating system, one in each partition. For instance, using two partitions it is possible to have Linux on one and Windows on another partition in the disk. This gives enormous flexibility of operations. Typically, a 80 GB disk in modern machines may be utilized to house Windows XP and Linux with nearly 40 GB disk available for each.

Yet another associated concept in this context, is the way the disk partitions are mounted on a file system. Clearly, a disk partition, with all its contents, is essentially a set of organized information. It has its own directory structure. Hence, it is a tree by itself. This tree gets connected to some node in the overall tree structure of the file system and forks out. This is precisely what mounting means. The partition is regarded to be mounted in the file system. This basic concept is also carried to the file servers on a network. The network file system may have remote partitions which are mounted on it. It offers seamless file access as if all of the storage was on the local disk. In modern systems, the file servers are located on networks somewhere without the knowledge of the user. From a user's standpoint all that is important to note is that as a user, his files are a part of a large tree structure which is a file system.

1.8.2 Portable storage

There are external media like tapes, disks, and floppies. These storage devices can be physically ported. Most file systems recognize these as on-line files when these are mounted on an IO device like a tape drive or a floppy drive. Unix treats these as special files. PCs and MAC OS recognize these as external files and provide an icon when these are mounted.

In this chapter we have covered considerable ground. Files are the entities that users deal with all the time. Users create files, manage them and seek system support in their file management activity. The discussion here has been to help build up a conceptual basis and leaves much to be covered with respect to specific instructions. For specifics, one should consult manuals. In this very rapidly advancing field, while the concept does not change, the practice does and does at a phenomenal pace.

Unit-3

Processes and Process management

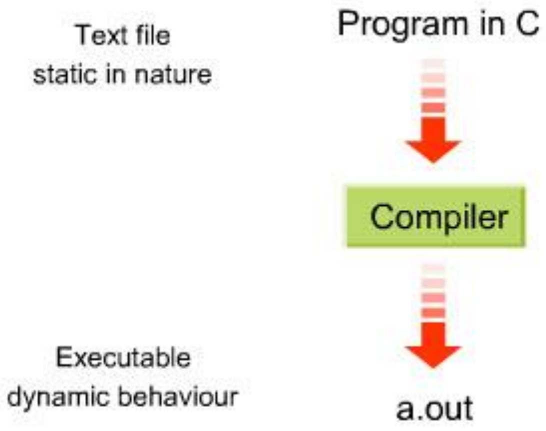
1.1 Introduction to Processes and Process management

a process is a ***program in executio****n*. In this module we shall explain how a process comes into existence and how processes are managed.

A process in execution needs resources like processing resource, memory and IO resources. Current machines allow several processes to share resources. In reality, one processor is shared amongst many processes. In the first module we indicated that the human computer interface provided by an OS involves supporting many concurrent processes like clock, icons and one or more windows.A system like a file server may even support processes from multiple users. And yet the owner of every process gets an illusion that the server (read processor) is available to their process without any interruption. This requires clever management and allocation of the processor as a resource. In this module we shall study the basic processor sharing mechanism amongst processes.

1.1.1 What is a Process

As we know a process is a *program in execution.* To understand the importance of this definition, let’s imagine that we have written a program called *my_prog.c* in C. On execution, this program may read in some data and output some data. Note that when a program is written and a file is prepared, it is still a script. It has no dynamics of its own i.e, it cannot cause any input processing or output to happen. Once we compile, and still later when we run this program, the intended operations take place. In other words, a program is a text script with no dynamic behavior. When a program is in execution, the script is acted upon. It can result in engaging a processor for some processing and it can also engage in I/O operations. It is for this reason a process is differentiated from program. While the program is a text script, a program in execution is a process.



In other words, To begin with let us define what is a "process" and in which way a process differs from a program. A process is an executable entity – it's a program in execution. When we compile a C language program we get an a.out file which is an executable file. When we seek to run this file – we see the program in execution. Every process has its instruction sequence. Clearly, therefore, at any point in time there is a current instruction in execution.

A program counter determines helps to identify the next instruction in the sequence. So process must have an inherent program counter. Referring back to the C language program – it's a text file. A program by itself is a passive entity and has no dynamic behavior of its own till we create the corresponding process. On the other hand, a process has a dynamic behavior and is an active entity.

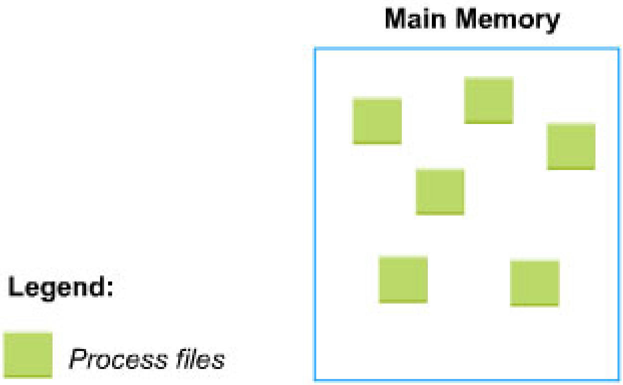
Processes get created, may have to be suspended awaiting an event like completing a certain I/O. A process terminates when the task it is defined for is completed. During the life time of a process it may seek memory dynamically. In fact, the *malloc* instruction in C precisely does that. In any case, from the stand point of OS a process should be memory resident and, therefore, needs to be stored in specific area within the main memory. Processes during their life time may also seek to use I/O devices.

For instance, an output may have to appear on a monitor or a printed output may be needed. In other words, process management requires not only making the processor available for execution but, in addition, allocates main memory, files and IO. The process management component then requires coordination with the main memory management, secondary memory management, as well as, files and I/O. We shall examine the memory management and I/O management issues briefly here. These topics will be taken up for more detailed study later.

1.2 Main Memory Management

As we observed earlier in the systems operate using Von-Neumann’s stored program concept. The basic idea is that an instruction sequence is required to be stored before it can be executed. Therefore, every executable file needs to be stored in the main memory.

In addition, we noted that modern systems support multi-programming. This means that more than one executable process may be stored in the main memory. If there are several programs residing in the memory, it is imperative that these be appropriately assigned specific areas.



Main memory management

The OS needs to select one amongst these to execute. Further these processes have their data areas associated with them and may even dynamically seek more data areas. In other words, the OS must cater to allocating and de-allocating memory to processes as well as to the data required by these processes. All processes need files for their operations and the OS must manage these as well.

1.2.1 Files and IO Management

On occasions processes need to operate on files. Typical file operations are:

1. Create: To create a file in the environment of operation
2. Open: To open an existing file from the environment of operation.
3. Read: To read data from an opened file.
4. Write: To write into an existing file or into a newly created file or it may be to modify or append or write into a newly created file.
5. Append: Like write except that writing results in appending to an existing file.
6. Modify or rewrite: Essentially like write – results in rewriting a file.

7. Delete: This may be to remove or disband a file from further use.

OS must support all of these and many other file operations. For instance, there are other file operations like which applications or users may be permitted to access the files. Files may be "owned" or "shared". There are file operations that permit a user to specify this. Also, files may be of different types. For instance, we have already seen that we may have executable and text files. In addition, there may be image files or audio files. Later in this course you will learn about various file types and the file operations on more details. For now it suffices to know that one major task OSs perform related to management of files.

1.2.2 Process Management

Multi-Programming and Time Sharing: To understand processes and management, we begin by considering a simple system with only one processor and one user running only one program, prog_1 shown in fig 3.1 (a).

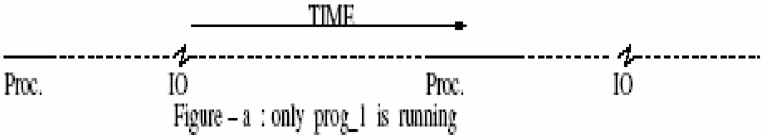


Figure 3.1(a): Multiple-program Processing

We also assume that IO and processing takes place serially. So when an IO is required the processor waits for IO to be completed .When we input from a keyboard we are operating at a speed nearly a million times slower than that of a processor. So the processor is idle most of time. Even the fastest IO devices are at least 1000 times slower than processors, i.e, a processor is heavily underutilized as shown in fig 3.1.

Recall that Von Neumann computing requires a program to reside in main memory to run. Clearly, having just one program would result in gross underutilization of the processor.

Let us assume that we now have two ready to run programs

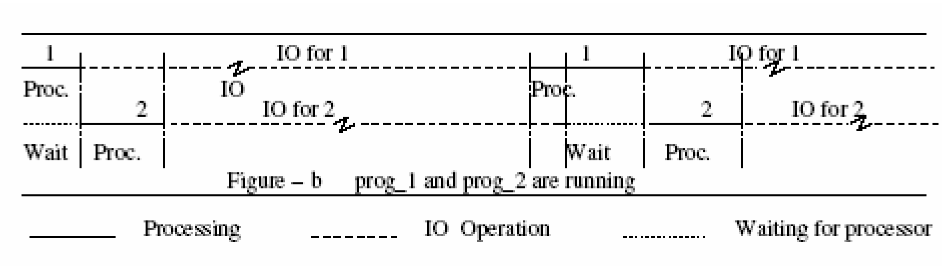


Figure 3.1(b): Multiple-program Processing

Consider two programs prog_1 and prog_2 resident in the main memory. When prog_1 may be seeking the IO we make the processor available to run prog_2 that is we may schedule the operation of prog_2.Because the processor is very much faster compared to all other devices, we will till end up with processor waiting for IO to be completed as shown in fig 3.1(b). In this case we have two programs resident in main memory. A multi-programming OS allows and manages several programs to be simultaneously resident in main memory.

1.3 Processor Utilization

Processor Utilization: A processor is a central and a key element of a computer system. This is so because all information processing gets done in a processor. So a computer's throughput depends upon the extent of utilization of its processor. The greater the utilization of the processor, larger is the amount of information processed.

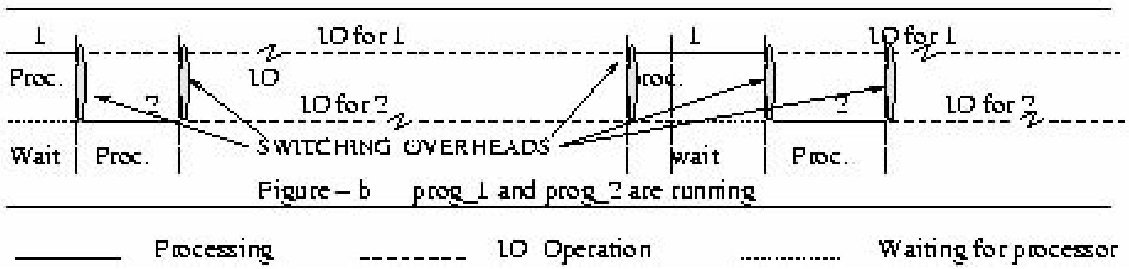


Figure – b prog_1 and prog_2 are running

In the light of the above let us briefly review this figure above. In a uni-programming system (figure a) we have one program engaging the processor. In such a situation the processor is idling for very long periods of time. This is so because IO and communication to devices (including memory) takes so much longer. In figure above we see that

during intervals when prog_1 is not engaging the processor we can utilize the processor to run another ready to run program. The processor now processes two programs without significantly sacrificing the time required to process prog_1. Note that we may have a small overhead in switching the context of use of a processor. However, multiprogramming results in improving the utilization of computer's resources. In this example, with multiple programs residing in the memory, we enhance the memory utilization also!!.

When we invest in a computer system we invest in all its components. So if any part of the system is idling, it is a waste of resource. Ideally, we can get the maximum through put from a system when all the system components are busy all the time. That then is the goal.

Multiprogramming support is essential to realize this goal because only those programs that are resident in the memory can engage devices within in a system.

1.3.1 Response Time

So far we have argued that use of multiprogramming increases utilization of processor and other elements within a computer system. So we should try to maximize the number of ready-to-run programs within a system. In fact, if these programs belong to different users then we have achieved sharing of the computer system resource amongst many users. Such a system is called a time sharing system.

We had observed that every time we switch the context of use of a processor we have some overhead. For instance, we must know where in the instruction sequence was the program suspended. We need to know the program counter (or the instruction address) to resume a suspended program. In addition, we may have some intermediate values stored in registers at the time of suspension. These values may be critical for subsequent instructions. All this information also must be stored safely somewhere at the time of suspension (i.e. before context of use is switched). When we resume a suspended program we must reload all this information (before we can actually resume). In essence, a switch in the context of use has its overhead. When the number of resident user programs competing for the same resources increases, the frequency of storage, reloads and wait periods also increase. If the over heads are high or the context switching is too frequent, the users will have to wait longer to get their programs executed. In other words, response time of the system will become longer. Generally, the response time of a system is defined as the time interval which spans the time from the last character input to the first character of output. It is important that when we design a time sharing system we keep the response time at some acceptable level. Otherwise the advantage of giving access to, and sharing, the resource would be lost. A system which we use to access book information in a library is a time-shared system. Clearly, the response time should be such that it should be acceptable, say a few seconds. A library system is also an online system .In an online system, devices (which can include instrumentation in a plant) are continuously monitored (observed) by the computer system . If in an online system the response time is also within some acceptable limits then we say it is a real-time system. For instance, the airlines or railway booking office usually has a real-time online reservation system.

A major area of research in OS is performance evaluation. In performance evaluation we study the percentage utilization of processor time, capacity utilization of memory, response time and of course, the throughput of the overall computer system.

1.4 Process States

Process States: In the previous example we have seen a few possibilities with regards to the operational scenarios. For instance, we say that a process is in run state (or mode) when it is engaging the processor. It is in wait state (or mode) when it is waiting for an IO to be completed. It may be even in wait mode when it is ready-to-run but the processor may not be free as it is currently engaged by some other process.

Each of such identifiable states descry be current operational conditions of a process. A study of process states helps to model the behavior for analytical studies.

For instance, in a simplistic model we may think of a five state model. The five states are: new-process, ready-to-run, running, waiting-on-IO and exit. The names are self-explanatory.

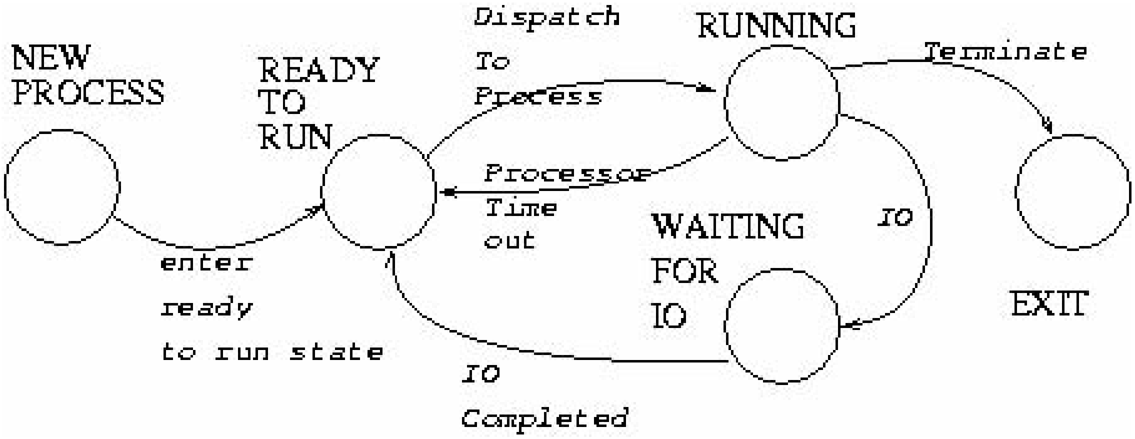


Figure 3.3

Figure 3.3: Modeling Process States

The new process is yet to be listed by an OS to be an active process that can be scheduled to execute. It enters the ready to run state when it is identified for future scheduling. Only then it may run. Once a processor is available then one of the ready to run processes may be chosen to run. It moves to the state “running”. During this run it may be timed out or may have to wait for an IO to be completed. If it moves to the state of waiting for IO then it moves to ready to run state when the IO is completed. When a process terminates its operation it moves to exit state. All of these transitions are expressed in the figure 3.3 above.

Process States: Management Issues

Process states: Management issues an important point to ponder is: what role does an OS play as processes migrate from one state to another? When a process is created the OS assigns it an id and also creates a data structure to record its progress. At some point in time OS makes this newly created process ready to run. This is a change in the state of this new process. With multiprogramming there are many ready to run processes in the main memory. The process data structure records state of a process as it evolves. A process marked as ready to run can be scheduled to run. The OS has a dispatcher module which chooses one of the ready to run processes and assigns it to the processor. The OS allocates a time slot to run this process. OS monitors the progress of every process during its life time. A process may, or may not, run for the entire duration of its allocated time slot. It may terminate well before the allocated time elapses or it may seek an IO. Sometimes a process may not be able to proceed till some event occurs. Such an event is detected as a synchronizing signal. Such a signal may even be received from some other process. When it waits for an IO to be completed, or some signal to arrive, the process is said to be blocked .OS must reallocate the processor now. OS marks this process as blocked for IO. OS must monitor all the IO activity to be able to detect completion of some IO or occurrence of some event. When that happens, the OS modifies the process data structure for one of the blocked processes and marks it ready to run. So, we see that OS maintains some data structures for management of processes. It modifies these data structures. In fact OS manages all the migrations between process states.

1.5 A Queuing Model

A Queuing Model: Data structures play an important role in management of processes. In general an OS may use more than one data structure in the management of processes. It may maintain a queue for all ready to run processes. It may maintain separate queues for blocked processes. It may even have a separate queue for each of the likely events (including completion of IO). This formulation shown in the figure 3.4 below is a very flexible model useful in modeling computer system operations. This type of model helps in the study and analysis of chosen OS policies.

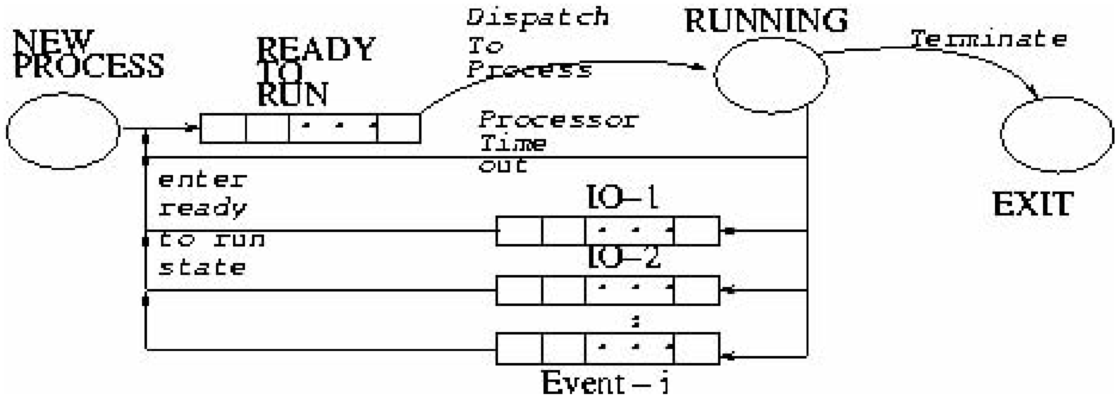


Figure 3.4: Queues-based Model

As an example, let us consider a first-come-first-served policy for ready-to-run queue. In such a case, processes enjoin the tail end of ready-to-run queue. Also, the processor is assigned to the process at the head of ready-to-run queue. Suppose we wish to compare this policy with another policy in which some processes have a higher priority over other processes. A comparison of these two policies can be used to study the following:

- The average and maximum delays experienced by the lowest priority process.

- Comparison of the best response times and throughputs in the two cases.

- Processor utilization in the two cases. And so on.

This kind of study can offer new insights. As an example, it is important to check what level of prioritization leads to a denial of service (also called starvation).The maxi mum delay for the lowest priority process increases as the range of priority difference increases. So at some threshold it may be unacceptably high. It may even become infinity. There may always be a higher priority process in the ready-to-run queue. As a result lower priority processes have no chance to run. That is starvation.

1.6 Scheduling

The OS maintains the data for processes in various queues. The OS keeps the process identifications in each queue. These queues advance based on some policy. These are usually referred to as scheduling policies.

To understand the nature of OS's scheduling policies, let us examine a few situations we experience in daily life. When we wish to buy a railway ticket at the ticket window, the queue is processed using an ``all customers are equal policy " i.e. first-come-first-served (FCFS). However, in a photocopy shop, customers with bulk copy requirements are often asked to wait. Some times their jobs are interrupted in favor of shorter jobs. The operators prefer to quickly service short job requests. This way they service a large number of customers quickly. The maximum waiting time for most of the customers is reduced considerably. This kind of scheduling is called shortest job first policy. In a university department, the secretary to the chairman of department always preempts any one's job to attend to the chairman's copy requests. Such a pre-emption is irrespective of the size of the job (or even its usefulness sometimes). The policy simply is priority based scheduling. The chairman has the highest priority. We also come across situations,

Typically in driving license offices and other bureaus, where applications are received till a certain time in the day (say 11:00 a.m.). All such applications are then taken as a batch. These are processed in the office and the outcome is announced for all at the same time (say 2:00 p.m.). Next batch of applications are received the following day and that batch is processed next. This kind of scheduling is termed batch processing.

In the context of processes we also need to understand preemptive and non-preemptive operations. Non-preemptive operations usually proceed towards completion uninterrupted. In a non-preemptive operation a process may suspend its operations temporarily or completely on its own. A process may suspend its operation for IO or terminate on completion. Note neither of these suspensions is forced upon it externally. On the other hand in a preemptive scheduling a suspension may be enforced by an OS. This may be to attend to an interrupt or because the process may have consumed its allocated time slot and OS must start execution of some other process. Note that each such policy affects the performance of the overall system in different ways.

1.7 Choosing a Policy

Depending upon the nature of operations the scheduling policy may differ. For instance, in a university set up, short job runs for student jobs may get a higher priority during assigned laboratory hours. In a financial institution processing of applications for investments may be processed in batches. In a design department projects nearing a deadline may have higher priority. So an OS policy may be chosen to suit situations with specific requirements. In fact, within a computer system we need a policy to schedule access to processor, memory, disc, IO and shared resource (like printers). For the present we shall examine processor scheduling policies only.

1.7.1 Policy Selection

A scheduling policy is often determined by a machine's configuration and usage. We consider processor scheduling in the following context:

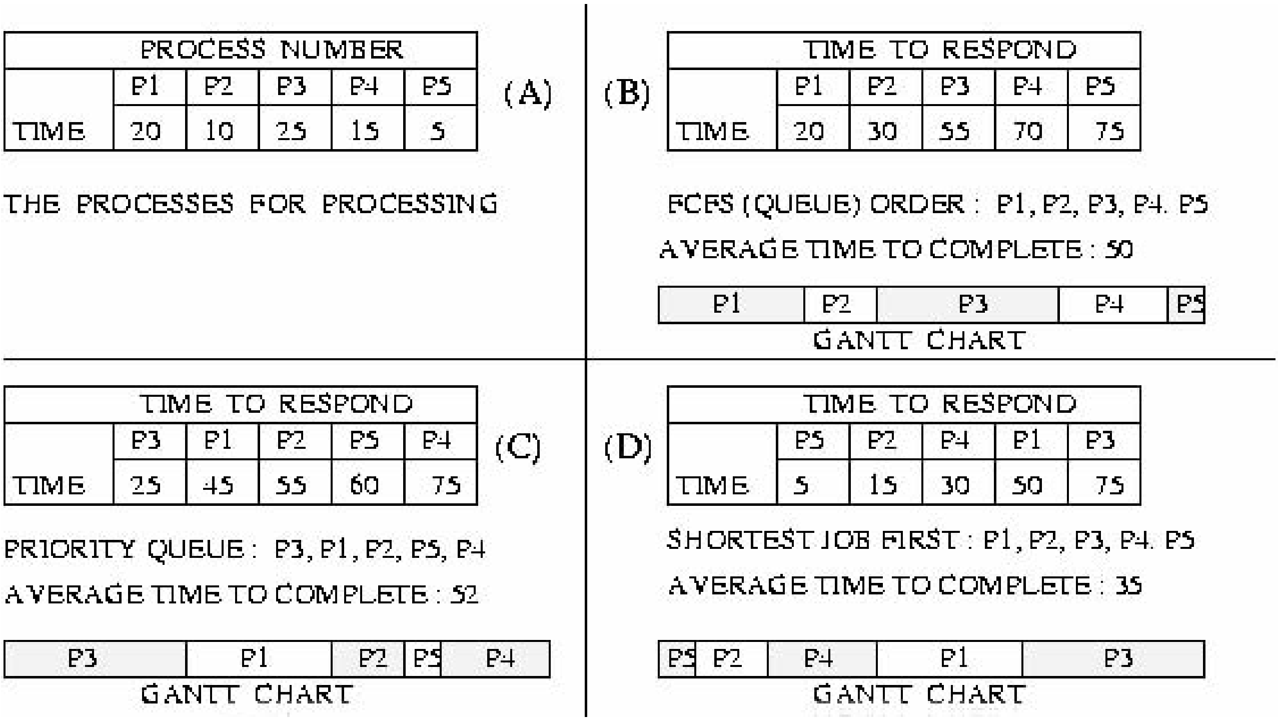
- We have only one processor in the system.
- We have a multiprogramming system i.e. there may be more than one ready-to run program resident in main memory.
- We study the effect (of the chosen scheduling policy) on the following:
    - The response time to users
    - The turnaround time (The time to complete a task).
    - The processor utilization.
    - The throughput of the system (Overall productivity of the system)
    - The fairness of allocation (includes starvation).
    - The effect on other resources.

A careful examination of the above criterion indicates that the measures for response time and turn around are user centered requirements. The processor utilization and throughput are system centered considerations. Last two affect both the users and system. It is quite possible that a scheduling policy satisfies users' needs but fails to utilize processor or gives a lower throughput. Some other policy may satisfy system centered requirements but may be poor from user's point of view. This is precisely what we will like to study. Though ideally we strive to satisfy both the user's and system's requirements, it may not be always possible to do so. Some compromises have to be made. To illustrate the effect of the choice of a policy, we evaluate each policy for exactly the same operational scenario. So, we set to choose a set of processes with some pre-assigned characteristics and evaluate each policy. We try to find out to what extent it meets a set criterion. This way we can compare these policies against each other.

1.7.2 Comparison of Policies

We begin by assuming that we have 5 processes p1 through p5 with processing time requirements as shown in the figure below at 3.5 (A).

- The jobs have run to completion.
- No new jobs arrive till these jobs are processed.
- Time required for each of the jobs is known apriori.
- During the run of jobs there is no suspension for IO operations.



(A)

	PROCESS NUMBER				
	P1	P2	P3	P4	P5
TIME	20	10	25	15	5

THE PROCESSES FOR PROCESSING

(B)

	TIME TO RESPOND				
	P1	P2	P3	P4	P5
TIME	20	30	55	70	75

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5

AVERAGE TIME TO COMPLETE : 50

(C)

	TIME TO RESPOND				
	P3	P1	P2	P5	P4
TIME	25	45	55	60	75

PRIORITY QUEUE : P3, P1, P2, P5, P4

AVERAGE TIME TO COMPLETE : 52

(D)

	TIME TO RESPOND				
	P5	P2	P4	P1	P3
TIME	5	15	30	50	75

SHORTEST JOB FIRST : P1, P2, P3, P4. P5

AVERAGE TIME TO COMPLETE : 35

Figure 3.5: Comparison of three non-preemptive scheduling policies

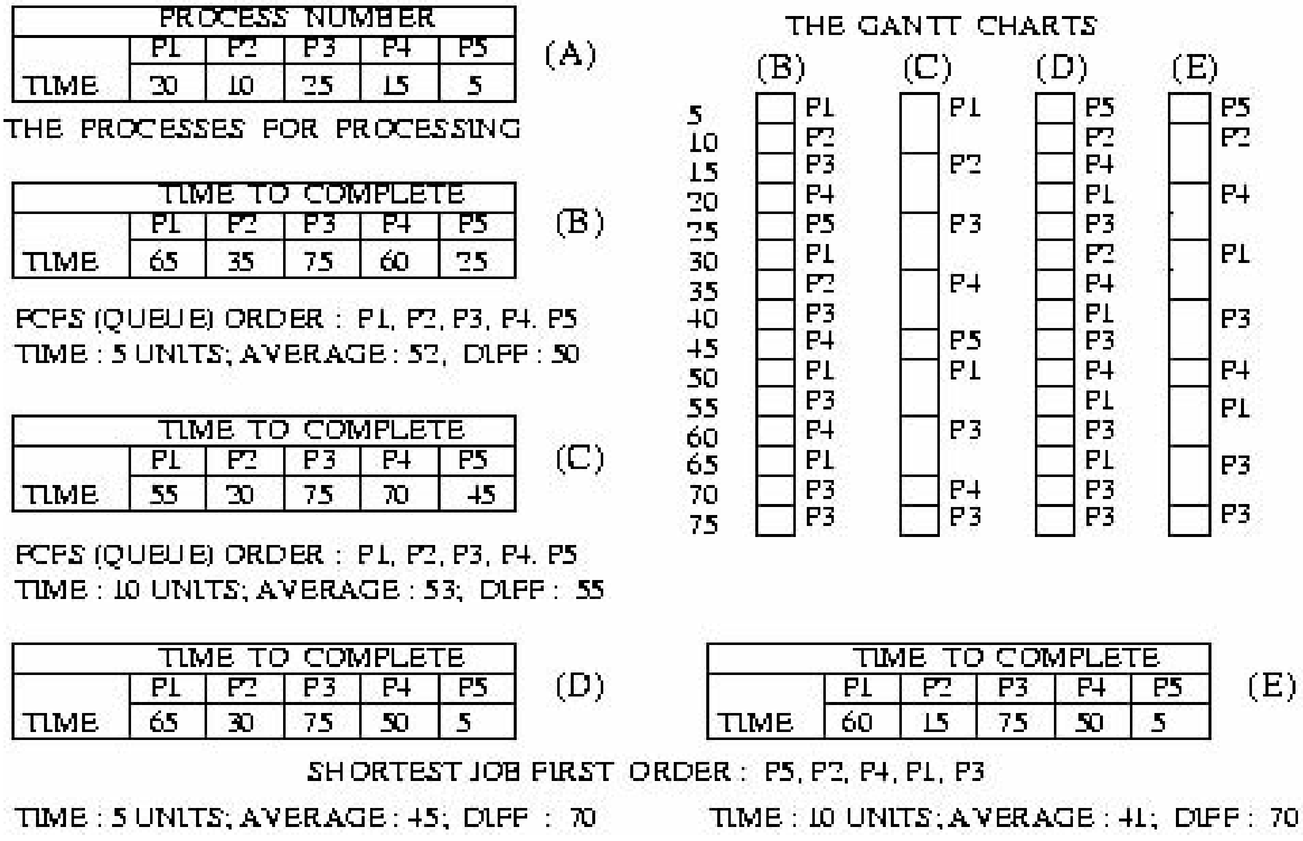
We assume non-preemptive operations for comparison of all the cases. We show the processing of jobs on a Gantt chart. Let us first assume processing in the FCFS or internal queue order i.e. p1, p2, p3, p4 and p5 (see 3.5(B)). Next we assume that jobs are arranged in a priority queue order (see3.5(C)). Finally, we assume shortest job first order. We compare the figures of merit for each policy. Note that in all we process 5 jobs over a total time of 75 time units. So throughput for all the three cases is same. However, the results are the poorest (52 units) for priority schedule, and the best for Shortest-job-first schedule. In fact, it is well known that shortest-job-first policy is optimal.

1.7.2 Pre-emptive Policies

We continue with our example to see the application of pre-emptive policies. These policies are usually followed to ensure fairness. First, we use a Round-Robin policy i.e. allocate time slots in the internal queue order. A very good measure of fairness is the difference between the maximum and minimum time to complete. Also, it is a good idea

to get some statistical measures of spread around the average value. In the figure 3.6 below we compare four cases. These cases are:

- The Round-Robin allocation with time slice = 5 units. (CASE B)
- The Round-Robin allocation with time slice = 10 units. (CASE C)
- Shortest Job First within the Round-Robin; time slice = 5 units. (CASE D)
- Shortest Job First within the Round-Robin; time slice = 10 units. (CASE E)



PROCESS NUMBER					
	P1	P2	P3	P4	P5
TIME	30	10	25	15	5

(A)

THE PROCESSES FOR PROCESSING

TIME TO COMPLETE					
	P1	P2	P3	P4	P5
TIME	65	35	75	60	25

(B)

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5
TIME : 5 UNITS; AVERAGE : 52, DIFF : 50

TIME TO COMPLETE					
	P1	P2	P3	P4	P5
TIME	55	30	75	70	45

(C)

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5
TIME : 10 UNITS; AVERAGE : 53; DIFF : 55

TIME TO COMPLETE					
	P1	P2	P3	P4	P5
TIME	65	30	75	50	5

(D)

THE GANTT CHARTS

	(B)	(C)	(D)	(E)
5	P1	P1	P5	P5
10	P2		P2	P2
15	P3	P2	P4	
20	P4		P1	P4
25	P5	P3	P3	
30	P1		P2	P1
35	P2	P4	P4	
40	P3		P1	P3
45	P4	P5	P3	
50	P1	P1	P4	P4
55	P3		P1	P1
60	P4	P3	P3	
65	P1		P1	P3
70	P3	P4	P3	
75	P3	P3	P3	P3

TIME TO COMPLETE					
	P1	P2	P3	P4	P5
TIME	60	15	75	50	5

(E)

SHORTEST JOB FIRST ORDER : P5, P2, P4, P1, P3

TIME : 5 UNITS; AVERAGE : 45; DIFF : 70

TIME : 10 UNITS; AVERAGE : 41; DIFF : 70

Figure 3.6: Comparison of Pre-emptive policy schedules

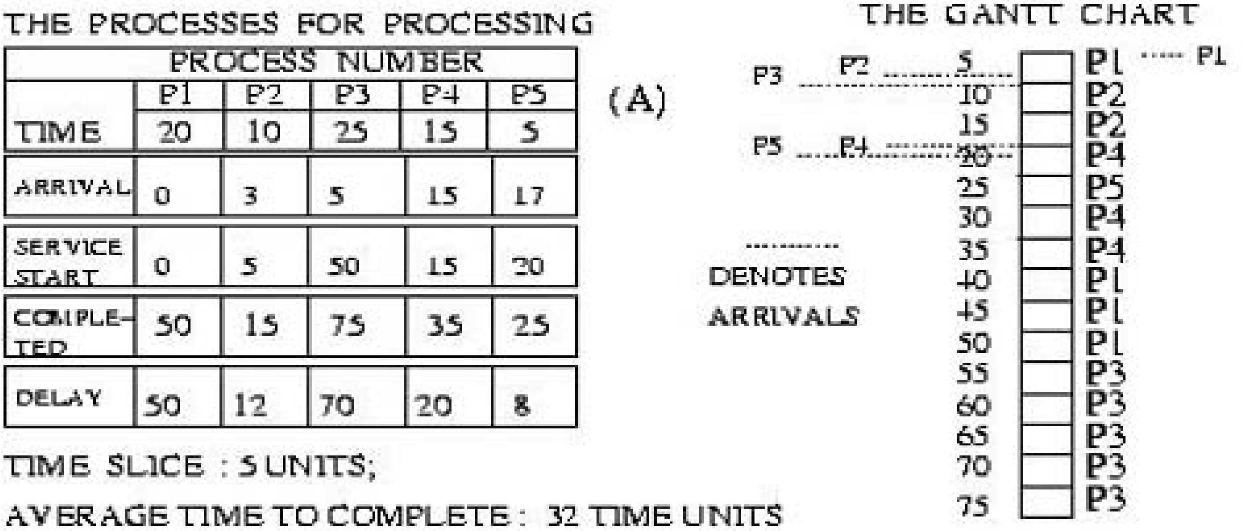
One of the interesting exercises is to find a good value for time slice for processor time allocation. OS designers spend a lot of time finding a good value for time slice.

Yet another Variation:

So far we had assumed that all jobs were present initially. However, a more realistic situation is processes arrive at different times. Each job is assumed to arrive with an estimate of time required to complete. Depending upon the arrival time and an estimated remaining time to complete jobs at hand, we can design an interesting variation of the shortest job first policy. It takes in to account the time which is estimated to be remaining to complete a job.

We could have used a job's service start time to compute the ``time required for completion"
as an alternative.

Also note that this policy may lead to starvation. This should be evident from the figure 3.7, the way job P3 keeps getting postponed. On the whole, though, this is a very good policy. However, some corrections need to be made for a job that has been denied service for a long period of time. This can be done by introducing some kind of priority (with jobs) which keeps getting revised upwards whenever a job is denied access for a long period of time. One simple way of achieving fairness is to keep a count of how often a job has been denied access to the processor. Whenever this count exceeds a certain threshold value this job must be scheduled during the next time slice.



THE PROCESSES FOR PROCESSING

	PROCESS NUMBER				
	P1	P2	P3	P4	P5
TIME	20	10	25	15	5
ARRIVAL	0	3	5	15	17
SERVICE START	0	5	50	15	20
COMPLE-TED	50	15	75	35	25
DELAY	50	12	70	20	8

TIME SLICE : 5 UNITS;

AVERAGE TIME TO COMPLETE : 32 TIME UNITS

THE GANTT CHART

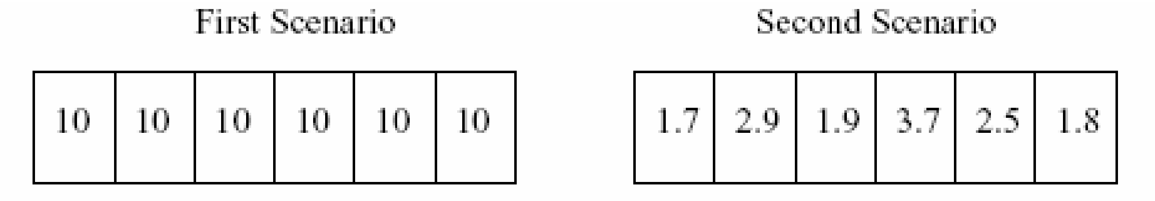
5	P1
10	P2
15	P2
20	P4
25	P5
30	P4
35	P4
40	P1
45	P1
50	P1
55	P3
60	P3
65	P3
70	P3
75	P3

(A)
P1
P3
P2
P5
P4
.... DENOTES ARRIVALS

Figure 3.7: Shortest Remaining Time Schedule

1.8 How to Estimate Completion Time?

We made an assumption that OS knows the processing time required for a process. In practice an OS estimates this time. This is done by monitoring a process's current estimate and past activity. This can be done by monitoring a process's current estimate and past activity as explained in the following example.



First Scenario

10	10	10	10	10	10

Second Scenario

1.7	2.9	1.9	3.7	2.5	1.8

Figure 3.9: Processor time utilisation.

Consider we have a process P. The OS allocates it a fixed time slice of 10 ms each time P gets to run. As shown in the Figure 3.9 in the first case it uses up all the time every time. The obvious conclusion would be that 10 ms is too small a time slice for the process P. May be it should be allocated higher time slices like 20 ms albeit at lower priority. In the
second scenario we notice that except once, P never really uses more than 3 ms time. Our obvious conclusion would be that we are allocating P too much time.

The observation made on the above two scenario offers us a set of strategies. We could base our judgment for the next time allocation using one of the following methods:

- Allocate the next larger time slice to the time actually used. For example, if time slices could be 5, 10, 15 ... ms then use 5 ms for the second scenario and 15 for the first (because 10 ms is always used up).
- Allocate the average over the last several time slice utilizations. This method gives all the previous utilizations equal weights to find the next time slice allocation.
- Use the entire history but give lower weights to the utilization in past, which means that the last utilization gets the highest, the previous to the last a little less and so on. This is what the exponential averaging technique does.

1.9 Exponential Averaging Technique

We denote our current, nth, CPU usage burst by t_n. Also, we denote the average of all past usage bursts up to now by τ_n . Using a weighting factor $0 \leq \alpha \leq 1$ with t_n and $1-\alpha$ with τ_n, we estimate the next CPU usage burst. The predicted value of τ_{n+1} is computed as : $\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n$

This formula is called an exponential averaging formula. Let us briefly examine the role of α . If it is equal to 1, then we note that the past history is ignored completely. The estimated next burst of usage is same as the immediate past utilization. If α is made 0 then we ignore the immediate past utilization altogether. Obviously both would be undesirable choices. In choosing a value of α in the range of 0 to 1 we have an opportunity to weigh the immediate past usage, as well as, the previous history of a process with decreasing weight. It is worthwhile to expand the formula further.

$$\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n = \alpha * t_n + \alpha * (1-\alpha) * t_{n-1} + (1-\alpha) * \tau_{n-1}$$

which on full expansion gives the following expression:

$$\tau_{n+1} = \alpha * t_n + (1-\alpha) * t_{n-1} + \alpha * (1-\alpha)^2 * t_{n-2} + \alpha * (1-\alpha)^3 * t_{n-3} \cdots$$

A careful examination of this formula reveals that successive previous bursts in history get smaller weights.

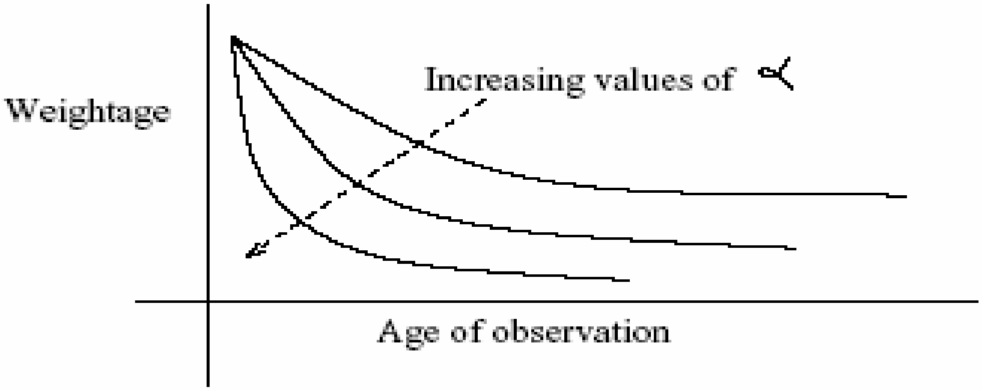


Figure 3.10: Processor time utilisation.

In Figure 3.10 we also see the effect of the choice of α has in determining the weights for past utilizations.

1.10 Multiple Queues Schedules

It is a common practice to associate some priority depending upon where the process may have originated. For instance, systems programs may have a higher priority over the user programs. Within the users there may be level of importance. In an on-line system the priority may be determined by the criticality of the source or destination. In such a case, an OS may maintain many process queues, one for each level of priority. In a real-time system we may even follow an earliest deadline first schedule. This policy introduces a notion priority on the basis of the deadline. In general, OSs schedule processes with a mix of priority and fairness considerations.

1.10.1 Two Level Schedules

It is also a common practice to keep a small number of processes as ready-to-run in the main memory and retain several others in the disks. As processes in the main memory block, or exit, processes from the disk may be loaded in the main memory. The process of moving processes in and out of main memory to disks is called swapping. The OSs have a swapping policy which may be determined by how “big" the process is. This may be determined by the amount of its storage requirement and how long it takes to execute. Also, what is its priority? We will learn more about on swapping in memory management chapter.

What Happens When Context Is Switched?

We will continue to assume that we have a uni-processor multi-programming environment. We have earlier seen that only ready-to-run, main memory resident processes can be scheduled for execution. The OS usually manages the main memory by dividing it into two major partitions. In one partition, which is entirely for OS management, it keeps a record of all the processes which are currently resident in memory. This information may be organized as a single queue or a priority multiple queue or any other form that the designer may choose. In the other part, usually for user processes, all the processes that are presently active are resident.

An OS maintains, and keeps updating, a lot of information about the resources in use for a running process. For instance, each process in execution uses the program counter, registers and other resources within the CPU. So, whenever a process is switched, the OS moves out, and brings in, considerable amount of context switching information as shown in Figure 3.11. We see that process P_x is currently executing (note that the program counter is pointing in executable code area of P_x). Let us now switch the context in favor of running process P y. The following must happen:

- All the current context information about process P_x must be updated in its own context area.
- All context information about process P_y must be downloaded in its own context area.
- The program counter should have an address value to an instruction of process P_y. and process P_y must be now marked as “running".

The process context area is also called process control block. As an example when the process P_x is switched the information stored is:

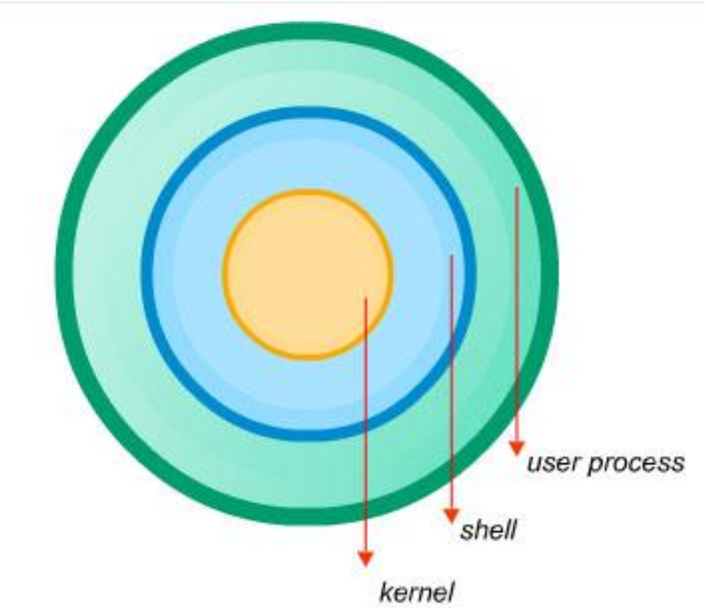
1. Program counter
2. Registers (like stack, index etc.) currently in use
3. Changed state (changed from Running to ready-to-run)
4. The base and limit register values
5. IO status (files opened; IO blocked or completed etc.)
6. Accounting
7. Scheduling information
8. Any other relevant information.

When the process P_y is started its context must be loaded and then alone it can run.

1.11 Kernel Architecture

Shells:

Most modern operating system distinguishes between a user process and a system process or utility. The user processes may have fewer privileges. For instance, the Unix and its derivatives permit user processes to operate within a shell (see figure).



This mode of operation shields the basic kernel of the operating system from direct access by a user process. The kernel is the one that provides OS services by processing system calls to perform IO or do any other form of process management activity – like delete a certain process. User processes can however operate within a shell and seek kernel services. The shell acts as a command interpreter. The command and its arguments are analyzed by the shell and a request is made to the kernel to provide the required service. There are times when a user needs to give a certain sequence of commands. These may form a batch file or a user may write a shell script to achieve the objective. This brings us essentially understand how operating systems handle system calls.

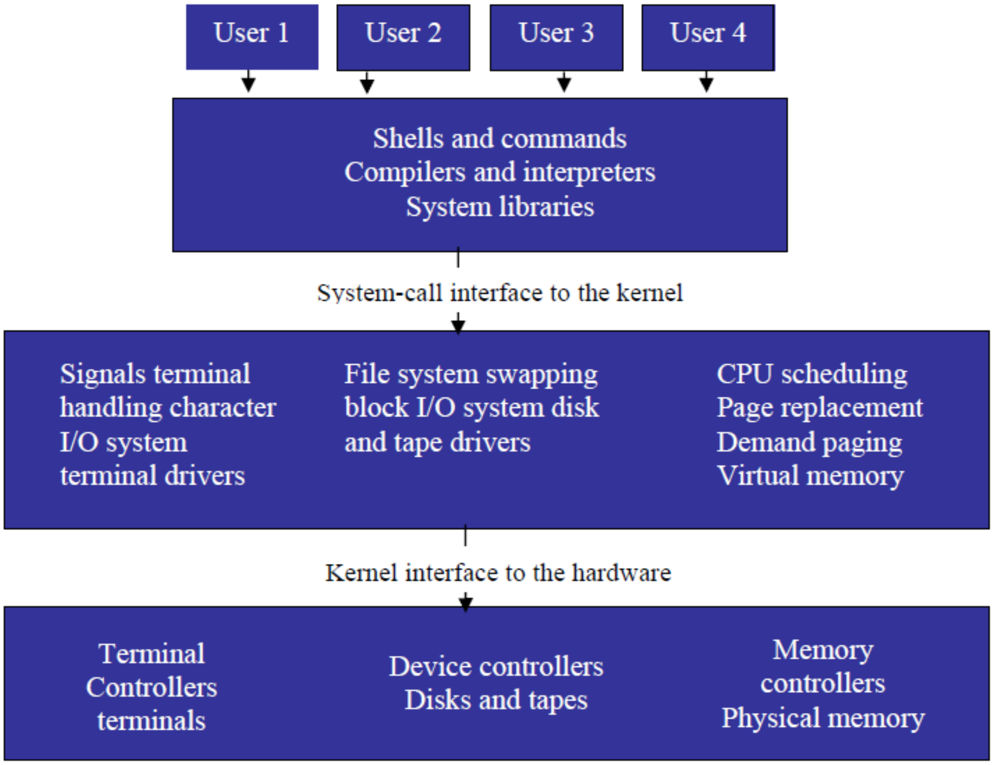
1.11.1 System Calls

As we explained earlier most user processes require a system call to seek OS services. Below we list several contexts in which user processes may need to employ a system call for getting OS services. The list below is only a representative list which shows a few user process activities that entail system calls. For instance it may need in process context (1-3), file and IO management context (4-6), or a network communication context (7-10).

1. To create or terminate processes.
2. To access or allocate memory.
3. To get or set process attributes.
4. To create, open, read, write files.
5. To change access rights on files.
6. To mount or un-mount devices in a file system.
7. To make network connections.
8. Set parameters for the network connection.
9. Open or close ports of communication.
10. To create and manage buffers for device or network communication.

1.12 Layered Design

A well-known software engineering principle in the design of systems is: the separation of concerns. This application of this concept leads to structured and modular designs. Such are also quite often more maintainable and extensible. This principle was applied in the design of Unix systems. The result is the layered design as shown in the figure. In the context of the layered design of Unix it should be remarked that the design offers easy to user layers hiding unnecessary details as is evident from the figure. Unix has benefited from this design approach. With layering and modularization, faults can be easily isolated and traceable to modules in Unix. This makes Unix more maintainable. Also, this approach offers more opportunities to add utilities in Unix – thus making it an extensible system.



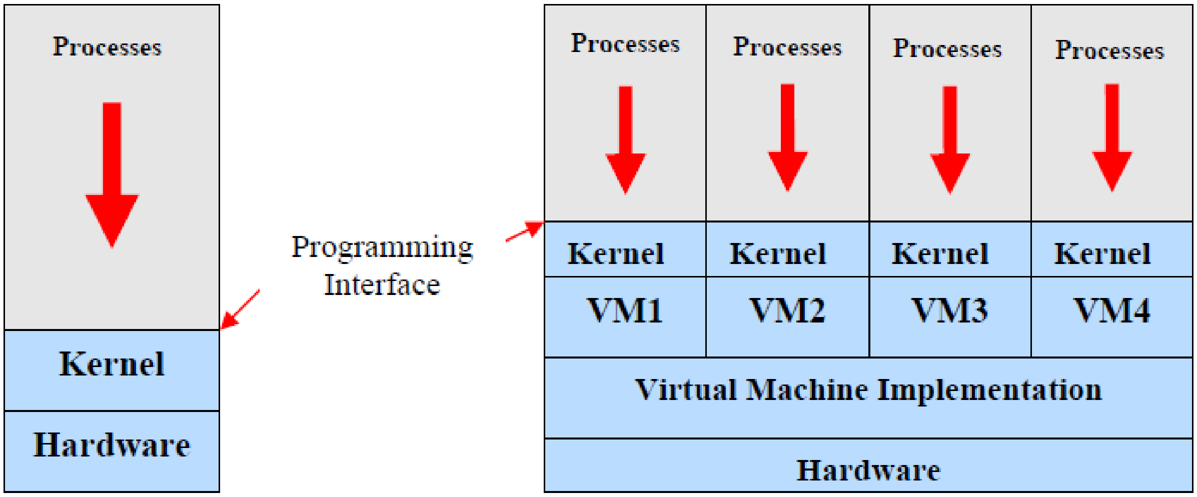
Unix Viewed as a Layered OS

1.13 The Virtual Machine Concept

One of the great innovations in the OS designs has been to offer a virtual machine. A virtual machine is an illusion of an actual machine by offering a form of replication of the same operating environment. This is achieved by clever scheduling as illustrated in the figure. As an illustration of such illusion consider spooling. Suppose a process seeks to output on a printer while the printer is busy. OS schedules it for later operation by spooling the output in an area of disk. This gives the process which sought the output, an impression that the print job has been attended to.

The figure depicts the manner in which the clever notion of virtual machine to support operation of multiple processes. OS ensures that each process gets an impression that all the resources of the system are available to each of the processes.

The notion of virtual machine has also been utilized to offer operating of one machine environment within the operative framework of another OS. For instance, it is a common knowledge that on a Sun machine one can emulate an offer operational environment of Windows-on-Intel (WINTEL).



System models. (1) Non-virtual machine. (2) Virtual machine.

As an avid reader may have observed, each process operates in its own virtual machine environment, the system security is considerably enhanced. This a major advantage of employing the virtual machine concept. A good example of a high level virtual machine is when uses Java Virtual machine. It is an example which also offers interoperability.

1.14 System Generation:

System generation is often employed at the time of installation as well as when upgrades are done. In fact, it reflects the ground reality to the OS. During system generation all the system resources are identified and mapped to the real resources so that the OS gets the correct characteristics of the resources. For instance, the type of modem used, its speed and protocol need to be selected during the system generation. The same applies for the printer, mouse and all the other resources used in a system. If we upgrade to augment RAM this also need to be reflected. In other words OS needs to selected the correct options to map to the actual devices used in a system.

1.15 Linux: An Introduction

Linux is a Unix like operating system for PCs. It is also POSIX complaint. It was first written by Linus Torvalds, a student from Finland, who started the work on it in 1991 as an academic project. His primary motivation was to learn more about the capabilities of a 386 processor for task switching. As for writing an OS, he was inspired by the Minix OS

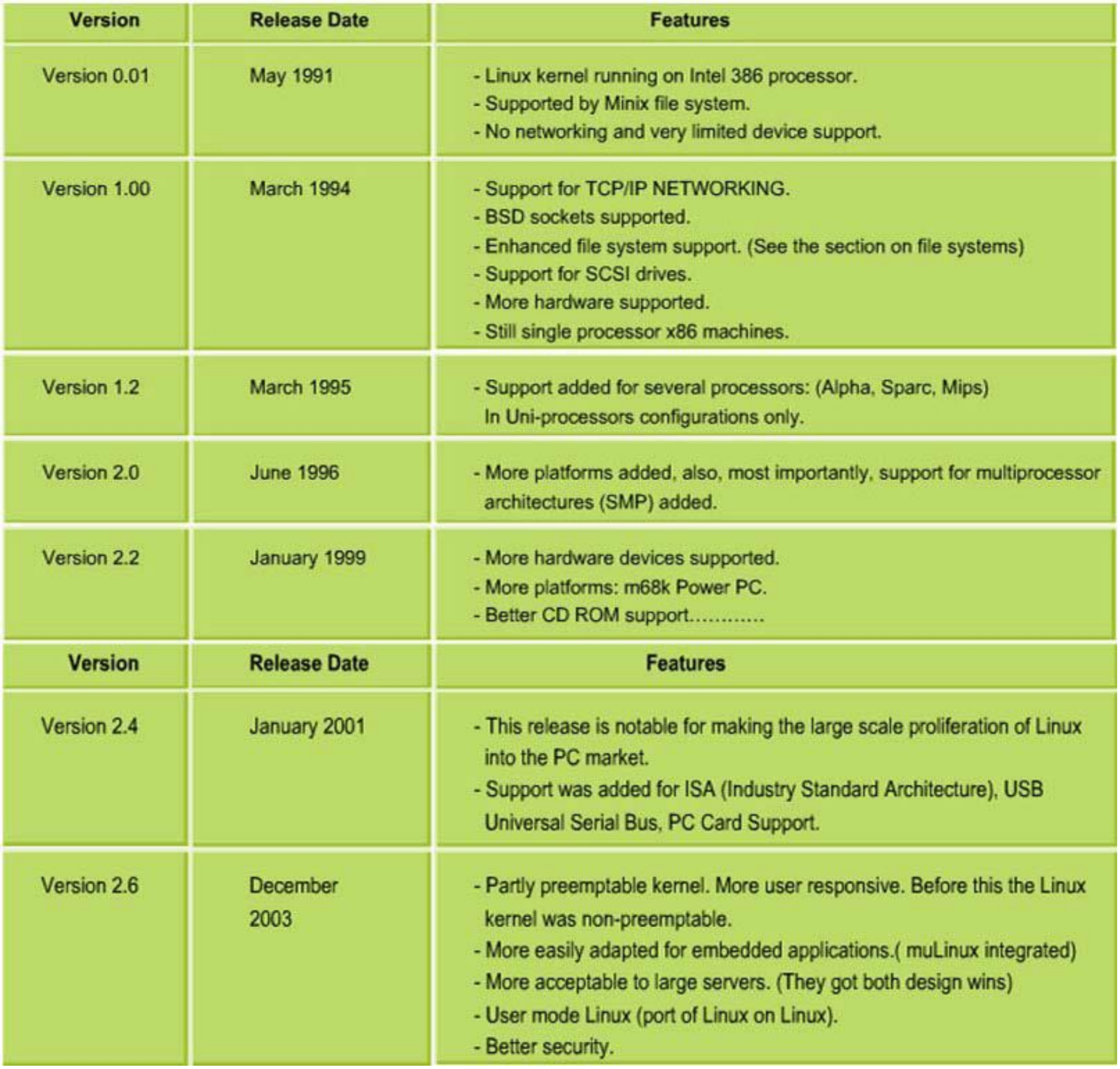
developed by Prof. Andrew Tanenbaum (from Vrije Universiteit, Amsterdam, The Netherlands Personal website http://www.cs.vu.nl/~ast/) Minix was offered by Prof. Tanenbaum as a teaching tool to popularize teaching of OS course in Universities. Here are two mails Mr. Torvalds had sent to the Minix mail group and which provide the genesis of Linux.

Truly speaking, Linux is primarily the kernel of an OS. An operating system is not just the kernel. Its lots of “other things” as well. Today an OS supports a lot of other useful software within its operative environments. OS quite commonly support compilers, editors, text formatters, mail software and many other things. In this case of the “other things” were provided by Richard Stallman's GNU project. Richard Stallman started the GNU movement in 1983. His desire was to have a UNIX like free operating system.

Linux borrows heavily from ideas and techniques developed for Unix. Many programs that now run under Linux saw their first implementation in BSD. X-windows system that Linux uses, was developed at MIT. So maybe we could think of Linux as Linux = Unix + Ideas from (BSD + GNU+ MIT+ ……) and still evolving.

Linux continues to evolve from the contributions of many independent developers who cooperate. The Linux repository is maintained by Linux Torvalds and can be accessed on the internet. Initially, Linux did not support many peripherals and worked only on a few processors. It is important to see how the Linux community has grown and how the contributions have evolved Linux into a full fledged OS in its own right.

The features have enhanced over time. The table below describes how incrementally the features got added, modified or deleted.



Version	Release Date	Features
Version 0.01	May 1991	- Linux kernel running on Intel 386 processor. - Supported by Minix file system. - No networking and very limited device support.
Version 1.00	March 1994	- Support for TCP/IP NETWORKING. - BSD sockets supported. - Enhanced file system support. (See the section on file systems) - Support for SCSI drives. - More hardware supported. - Still single processor x86 machines.
Version 1.2	March 1995	- Support added for several processors: (Alpha, Sparc, Mips) In Uni-processors configurations only.
Version 2.0	June 1996	- More platforms added, also, most importantly, support for multiprocessor architectures (SMP) added.
Version 2.2	January 1999	- More hardware devices supported. - More platforms: m68k Power PC. - Better CD ROM support…………
Version	**Release Date**	**Features**
Version 2.4	January 2001	- This release is notable for making the large scale proliferation of Linux into the PC market. - Support was added for ISA (Industry Standard Architecture), USB Universal Serial Bus, PC Card Support.
Version 2.6	December 2003	- Partly preemptable kernel. More user responsive. Before this the Linux kernel was non-preemptable. - More easily adapted for embedded applications.(muLinux integrated) - More acceptable to large servers. (They got both design wins) - User mode Linux (port of Linux on Linux). - Better security.

1.15.1 The Linux Distribution

The best known distributions are from RedHat, Debian and Slackware. There are other distributions like SuSE and Caldera and Craftworks.

There are many free down loads available. It is advisable to look up the internet for these. We shall list some of the URLs at the end of the session as references and the reader is encouraged to look these up for additional information on Linux.

1.15.6 Linux Design Considerations

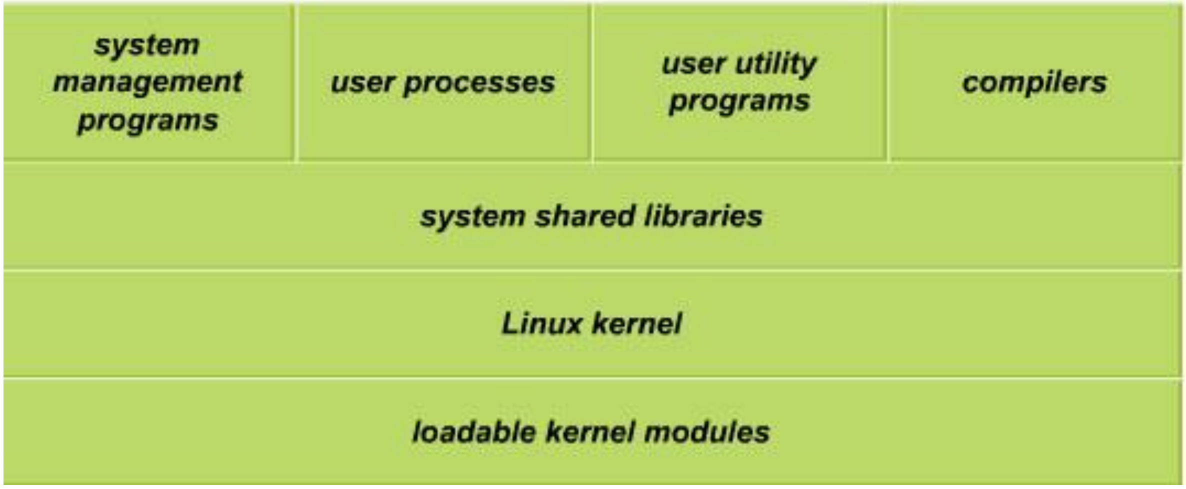
Linux is a Unix like system which implies it is a multi-user, multi-tasking system with its file system as well as networking environment adhering to the Unix semantics. From the very beginning Linux has been designed to be Posix compliant. One of the advantages today is the cluster mode of operation. Many organizations operate Linux clusters as servers, search engines. Linux clusters operate in multiprocessor environment. The most often cited and a very heavily used environment using Linux clusters is the famous Google search engine. Google uses geographically distributed clusters, each having any where up to 1000 Linux machines.

1.15.7 Components of Linux

Like Unix it has three main constituents. These are:

1. Kernel
2. System libraries
3. System utilities.

Amongst these the kernel is the core component. Kernel manages processes and also the virtual memory. System libraries define functions that applications use to seek kernel services without exercising the kernel code privileges. This isolation of privileges reduces the kernel overheads enormously. Like in Unix, the utilities are specialized functions like “sort” or daemons like login daemons or network connection management daemons.



Unit-4

Memory Management

1.1 Introduction to memory management

The von Neumann principle for the design and operation of computers requires that a program has to be primary memory resident to execute. Also, a user requires revisiting his programs often during its evolution. However, due to the fact that primary memory is volatile, a user needs to store his program in some non-volatile store. All computers provide a non-volatile secondary memory available as an online storage. Programs and files may be disk resident and downloaded whenever their execution is required. Therefore, some form of memory management is needed at both primary and secondary memory levels.

Secondary memory may store program scripts, executable process images and data files. It may store applications, as well as, system programs. In fact, a good part of all OS, the system programs which provide services (the utilities for instance) are stored in the secondary memory. These are requisitioned as needed.

The main motivation for management of main memory comes from the support for multiprogramming.

Several executable processes reside in main memory at any given time. In other words, there are several programs using the main memory as their address space. Also, programs move into, and out of, the main memory as they terminate, or get suspended for some IO, or new executable are required to be loaded in main memory. So, the OS has to have some strategy for main memory management. In this chapter we shall discuss the management issues and strategies for both main memory and secondary memory.

1.3 Main Memory Management

Let us begin by examining the issues that prompt the main memory management.

- **Allocation:** First of all the processes that are scheduled to run must be resident in the memory. These processes must be allocated space in main memory.
- **Swapping, fragmentation and compaction:** If a program is moved out or terminates, it creates a hole, (i.e. a contiguous unused area) in main memory. When a new process is to be moved in, it may be allocated one of the available holes. It is quite possible that main memory has far too many small holes at a certain time. In such a situation none of these holes is really large enough to be allocated to a new process that may be moving in. The main memory is too fragmented. It is, therefore, essential to attempt compaction.

Compaction means OS re-allocates the existing programs in contiguous regions and creates a large enough free area for allocation to a new process.

- **Garbage collection:** Some programs use dynamic data structures. These programs dynamically use and discard memory space. Technically, the deleted data items (from a dynamic data structure) release memory locations. However, in practice the OS does not collect such free space immediately for allocation. This is because that affects performance. Such areas, therefore, are called garbage. When such garbage exceeds a certain threshold, the OS would not have enough memory available for any further allocation. This entails compaction (or garbage collection), without severely affecting performance.
- **Protection:** With many programs residing in main memory it can happen that due to a programming error (or with malice) some process writes into data or instruction area of some other process. The OS ensures that each process accesses only to its own allocated area, i.e. each process is protected from other processes.
- **Virtual memory:** Often a processor sees a large logical storage space (a virtual storage space) though the actual main memory may not be that large. So some facility needs to be provided to translate a logical address available to a processor into a physical address to access the desired data or instruction.
- **IO support**: Most of the block-oriented devices are recognized as specialized files. Their buffers need to be managed within main memory alongside the other processes. The considerations stated above motivate the study of main memory management.

One of the important considerations in locating an executable program is that it should be possible to relocate it anywhere in the main memory. We shall dwell upon the concept of relocation next.

1.3 Memory Relocation Concept

Relocation is an important concept. To understand this concept we shall begin with a linear map (one-dimensional view) of main memory. If we know an address we can fetch its contents. So, a process residing in the main memory, we set the program counter to an absolute address of its first instruction and can initiate its run. Also, if we know the locations of data then we can fetch those too. All of this stipulates that we know the

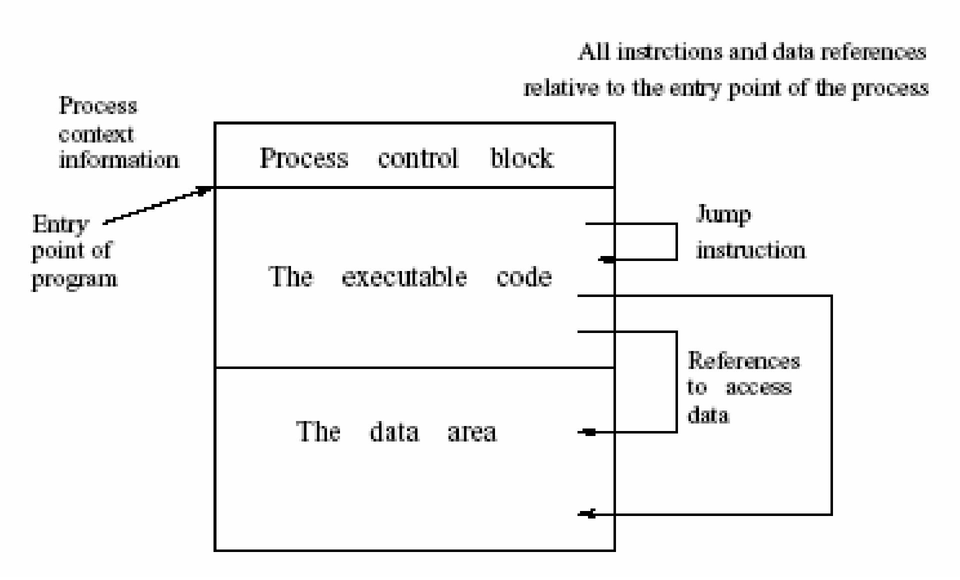

All instrctions and data references relative to the entry point of the process
Process context information
Process control block
Entry point of program
Jump instruction
The executable code
References to access data
The data area

Figure 4.1: The relocation concept.

absolute addresses for a program, its data and process context etc. This means that we can load a process with only absolute addresses for instructions and data, only when those specific addresses are free in main memory. This would mean we loose flexibility with regard to loading a process. For instance, we cannot load a process, if some other process is currently occupying that area which is needed by this process. This may happen even though we may have enough space in the memory. To avoid such a catastrophe, processes are generated to be relocatable. In Figure 4.1 we see a process resident in main memory.

Initially, all the addresses in the process are relative to the start address. With this flexibility we can allocate any area in the memory to load this process. Its instruction, data, process context (process control block) and any other data structure required by the process can be accessed easily if the addresses are relative. This is most helpful when processes move in and out of main memory. Suppose a process created a hole on moving out. In case we use non-relocatable addresses, we have the following very severe problem.

When the process moves back in, that particular hole (or area) may not be available any longer. In case we can relocate, moving a process back in creates no problem. This is so because the process can be relocated in some other free area. We shall next examine the linking and loading of programs to understand the process of relocation better.

1.3.1 Compiler Generated Bindings

The advantage of relocation can also be seen in the light of binding of addresses to variables in a program. Suppose we have a program variable *x* in a program *P*. Suppose the compiler allocated a fixed address to *x*. This address allocation by the compiler is called binding. If *x* is bound to a fixed location then we can execute program *P* only when *x* could be put in its allocated memory location. Otherwise, all address references to *x* will be incorrect.

If, however, the variable can be assigned a location relative to an assumed origin (or first address in program *P*) then, on relocating the program's origin anywhere in main memory, we will still be able to generate a proper relative address reference for *x* and execute the program. In fact, compilers generate relocatable code. In the next section we describe the linking, loading, and relocation of object code in greater detail.

1.4 Linking and Loading Concepts

In Figure 4.2 we depict the three stages of the way a HLL program gets processed.

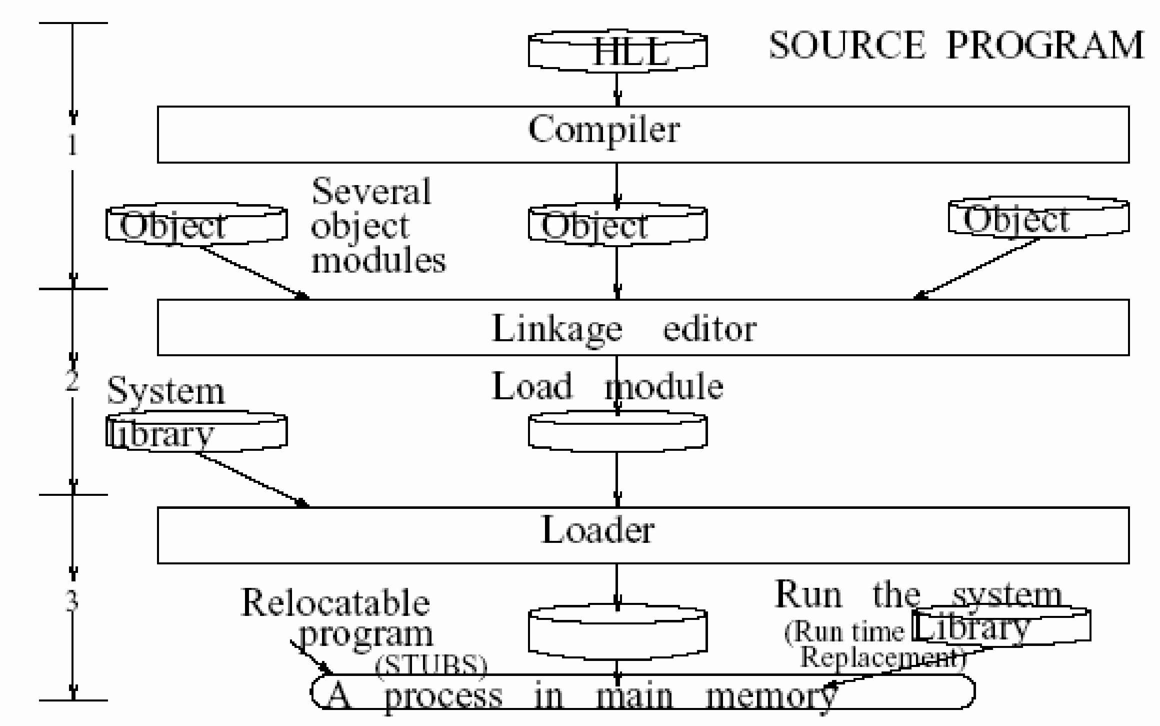


Figure 4.2: Linking and loading.

The three stages of the processing are:

- Stage 1: In the first stage the HLL source program is compiled and an object code is produced. Technically, depending upon the program, this object code may by itself be sufficient to generate a relocatable process. However many programs are compiled in parts, so this object code may have to link up with other object modules. At this stage the compiler may also insert stub at points where run time library modules may be linked.
- Stage 2: All those object modules which have sufficient linking information (generated by the compiler) for static linking are taken up for linking. The linking editor generates a relocatable code. At this stage, however, we still do not replace the stubs placed by compilers for a run time library link up.
- Stage3: The final step is to arrange to make substitution for the stubs with run time library code which is a relocatable code.

When all the three stages are completed we have an executable. When this executable is resident in the main memory it is a runnable process.

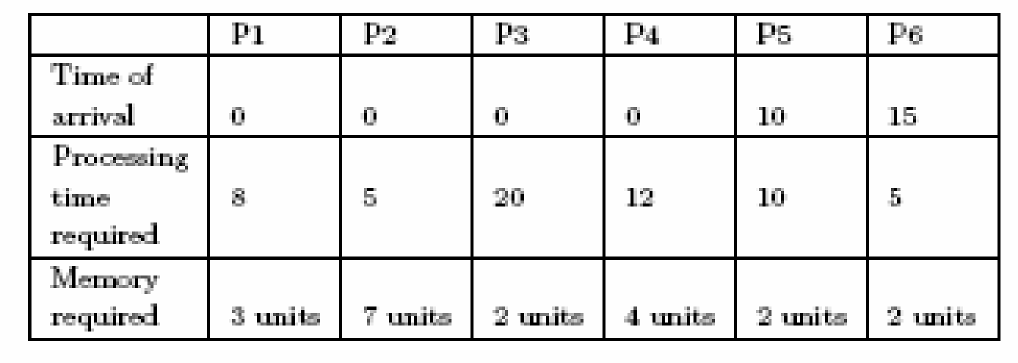
Recall our brief discussion in the previous section about the binding of variables in a program. The compiler uses a symbol table to generate addresses. These addresses are not bound, i.e. these do not have absolute values but do have information on sizes of data. The binding produced at compile time is generally relative. Some OSs support a linking loader which translates the relative addresses to relocatable addresses. In any event, the relocatable process is finally formed as an output of a loader.

1.5 Process and Main Memory Management

Once processes have been created, the OS organizes their execution. This requires interaction between process management and main memory management. To understand this interaction better, we shall create a scenario requiring memory allocations. For the operating environment we assume the following:

- A uni-processor, multi-programming operation.
- A Unix like operating system environment.

With a Unix like OS, we can assume that main memory is partitioned in two parts. One part is for user processes and the other is for OS. We will assume that we have a main memory of 20 units (for instance it could be 2 or 20 or 200 MB). We show the requirements and time of arrival and processing requirements for 6 processes in Table 4.1.

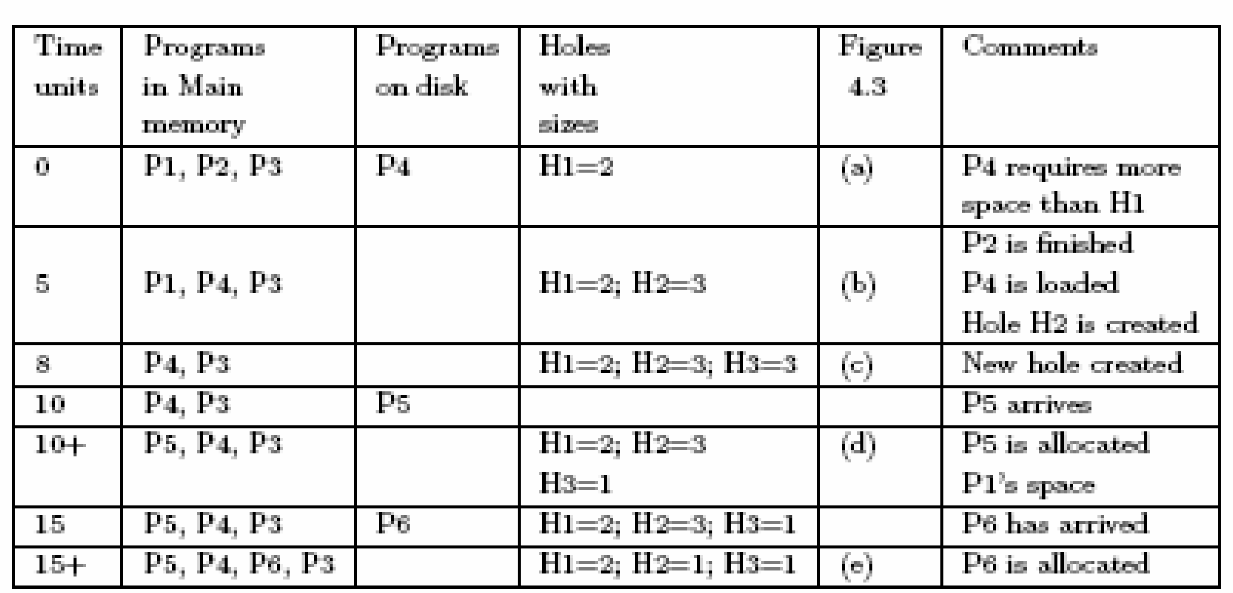


	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units

Table 4.1: The given data.

We shall assume that OS requires 6 units of space. To be able to compare various policies, we shall repeatedly use the data in Table 4.1 for every policy option. In the next section we discuss the first fit policy option.

With these requirements we can now trace the emerging scenario for the given data. We shall assume round robin allocation of processor time slots with no context switching



Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.3	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	New hole created
10	P4, P3	P5			P5 arrives
10+	P5, P4, P3		H1=2; H2=3 H3=1	(d)	P5 is allocated P1's space
15	P5, P4, P3	P6	H1=2; H2=3; H3=1		P6 has arrived
15+	P5, P4, P6, P3		H1=2; H2=1; H3=1	(e)	P6 is allocated

Table 4.2: FCFS memory allocation.

over-heads. We shall trace the events as they occur giving reference to the corresponding part in Table 4.2. This table also shows a memory map as the processes move in and out of the main memory.

1.6 The First Fit Policy: Memory Allocation

In this example we make use of a policy called first fit memory allocation policy. The first fit policy suggests that we use the first available hole, which is large enough to accommodate an incoming process. In Figure 4.3, it is important to note that we are following first-come first-served (process management) and first fit (memory allocation) policies. The process index denotes its place in the queue. As per first-come first-served policies the queue order determines the order in which the processes are allocated areas. In addition, as per first-fit policy allocation we scan the memory always from one end and find the first block of free space which is large enough to accommodate the incoming process.

In our example, initially, processes P1, P2, P3 and P4 are in the queue. The allocations for processes P1, P2, P3 are shown in 4.3(a). At time 5, process P2 terminates. So, process P4 is allocated in the hole created by process P2. This is shown at 4.3(b) in the figure. It still leaves a hole of size 3. Now on advancing time further we see that at time 8, process P1 terminates. This creates a hole of size 3 as shown at 4.3(c) in the figure.

This hole too is now available for allocation. We have 3 holes at this stage. Two of these 3 holes are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for the first hole which can accommodate it. This is the one created by the departure of process P1. Using the first-fit argument this is the hole allocated to process P5 as shown in Figure 4.3(d). The final allocation status is shown in Figure 4.3. The first-fit allocation policy is very easy to implement and is fast in execution.

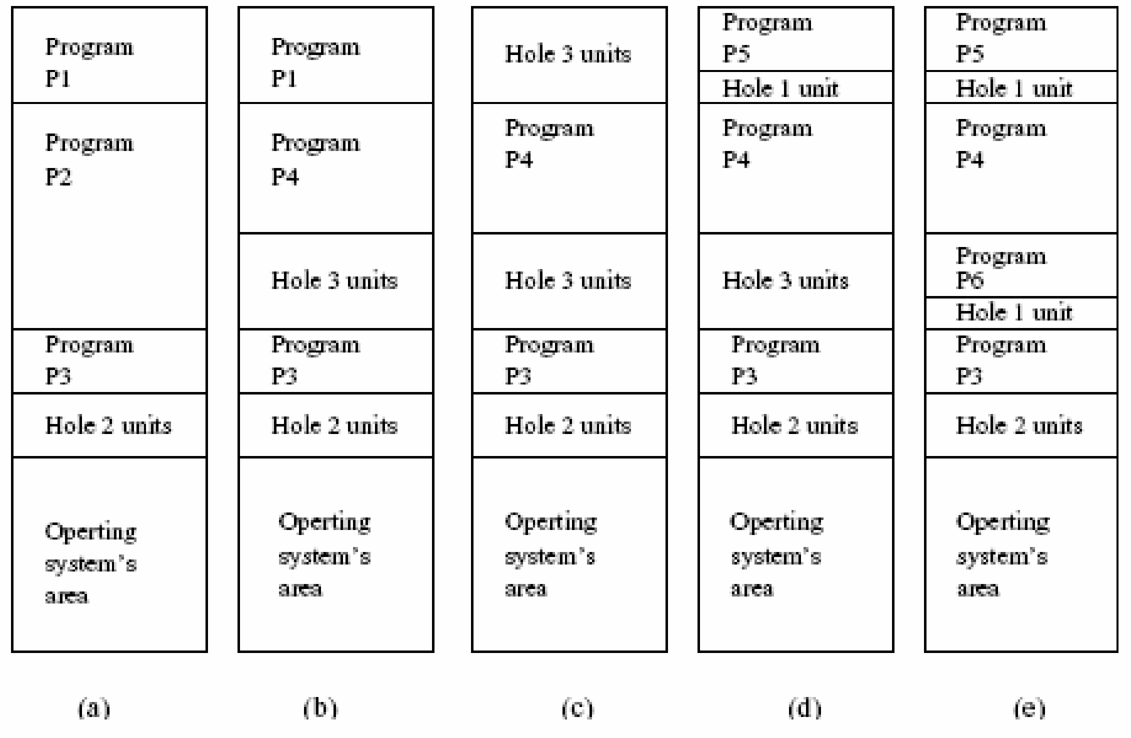


Figure 4.3: First-fit policy allocation.

1.7 The Best Fit Policy: Memory Allocation

The main criticism of first-fit policy is that it may leave many smaller holes. For instance, let us trace the allocation for process P5. It needs 2 units of space. At the time it moves into the main memory there is a hole with 2 units of space. But this is the last hole when we scan the main memory from the top (beginning). The first hole is 3 units. Using the first-fit policy process P5 is allocated this hole. So when we used this hole we also created a still smaller hole. Note that smaller holes are less useful for future allocations.

In the best-fit policy we scan the main memory for all the available holes. Once we have information about all the holes in the memory then we choose the one which is closest to the size of the requirement of the process. In our example we allocate the hole with size 2 as there is one available. Table 4.3 follows best-fit policy for the current example.

Also, as we did for the previous example, we shall again assume round-robin allocation of the processor time slots. With these considerations we can now trace the possible emerging scenario.

In Figure 4.4, we are following first-come first-served (process management) and best fit (memory allocation) policies. The process index denotes its place in the queue. Initially, processes P1, P2, P3 and P4 are in the queue. Processes P1, P2 and P3 are allocated as shown in Figure 4.4(a). At time 5, P2 terminates and process P4 is allocated in the hole so created. This is shown in Figure 4.4(b). This is the best fit. It leaves a space of size 3 creating a new hole. At time 8, process P1 terminates. We now have 3 holes. Two of these holes are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for a hole whose size is nearest to 2 and can accommodate P5. This is the last hole.

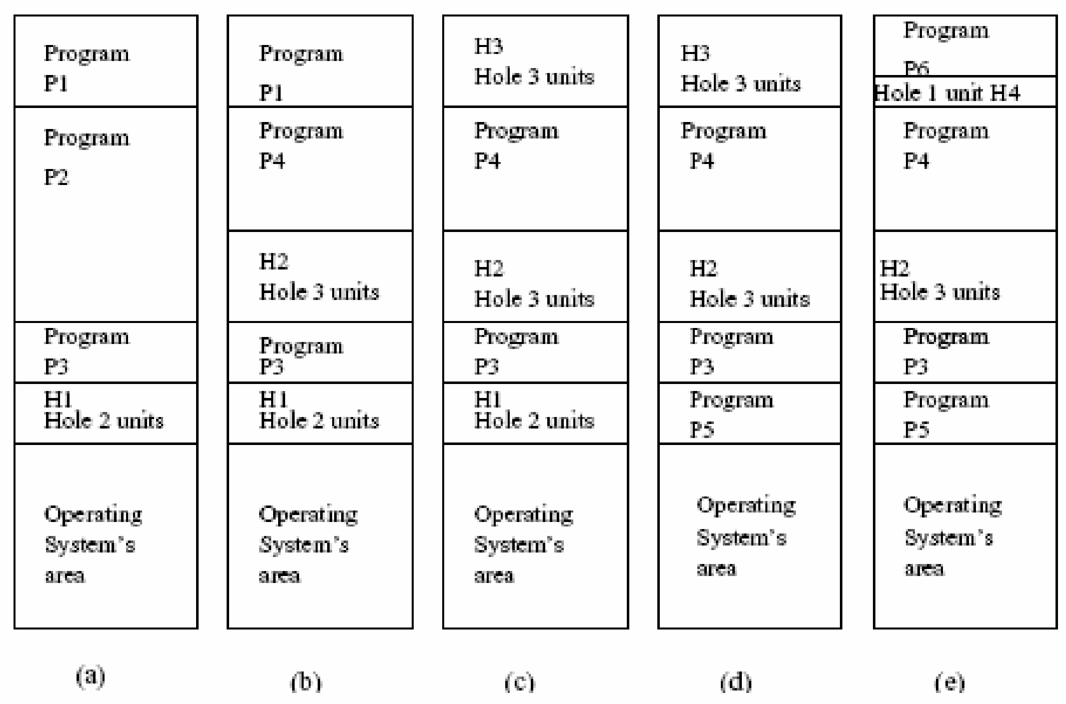
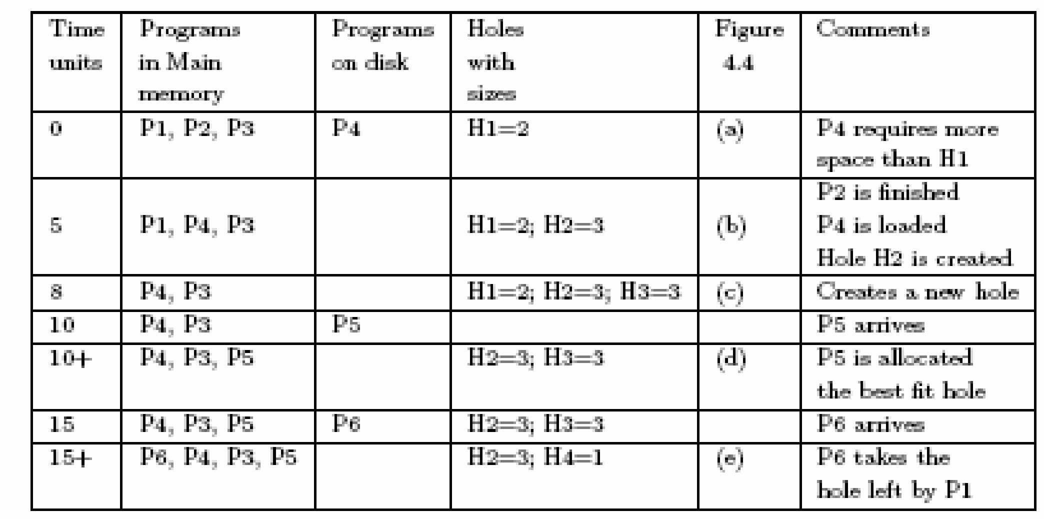


Figure 4.4: Best-fit policy allocation

Clearly, the best-fit (and also the worst-fit) policy should be expected to be slow in execution. This is so because the implementation requires a time consuming scan of all of main memory. There is another method called the next-fit policy. In the next-fit method the search pointer does not start at the top (beginning), instead it begins from where it ended during the previous search. Like the first-fit policy it locates the next first-fit hole that can be used. Note that unlike the first-fit policy the next-fit policy can be expected to distribute small holes uniformly in the main memory. The first-fit policy would have a tendency to create small holes towards the beginning of the main memory scan. Both first-fit and next-fit methods are very fast and easy to implement.

One of the important considerations in main memory management is: how should an OS allocate a chunk of main memory required by a process. One simple approach would be to somehow create partitions and then different processes could reside in different partitions. We shall next discuss how the main memory partitions may be created.



Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.4	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	Creates a new hole
10	P4, P3	P5			P5 arrives
10+	P4, P3, P5		H2=3; H3=3	(d)	P5 is allocated the best fit hole
15	P4, P3, P5	P6	H2=3; H3=3		P6 arrives
15+	P6, P4, P3, P5		H2=3; H4=1	(e)	P6 takes the hole left by P1

Table 4.3: Best-fit policy memory allocation.

1.8 Fixed and Variable Partitions

In a fixed size partitioning of the main memory all partitions are of the same size. The memory resident processes can be assigned to any of these partitions. Fixed sized partitions are relatively simple to implement. However, there are two problems. This scheme is not easy to use when a program requires more space than the partition size. In this situation the programmer has to resort to overlays. Overlays involve moving data and program segments in and out of memory essentially reusing the area in main memory. The second problem has to do with internal fragmentation. No matter what the size of the process is, a fixed size of memory block is allocated as shown in Figure 4.5(a). So there will always be some space which will remain unutilized within the partition.

In a variable-sized partition, the memory is partitioned into partitions with different sizes. Processes are loaded into the size nearest to its requirements. It is easy to always ensure the best-fit. One may organize a queue for each size of the partition as shown in the Figure 4.5(b). With best-fit policy, variable partitions minimize internal fragmentation.

However, such an allocation may be quite slow in execution. This is so because a process may end up waiting (queued up) in the best-fit queue even while there is space available elsewhere.

For example, we may have several jobs queued up in a queue meant for jobs that require 1 unit of memory, even while no jobs are queued up for jobs that require say 4 units of memory.

Both fixed and dynamic partitions suffer from external fragmentation whenever there are partitions that have no process in it. One of techniques that have been used to keep both internal and external fragmentations low is dynamic partitioning. It is basically a variable partitioning with a variable number of partitions determined dynamically (i.e. at run time).

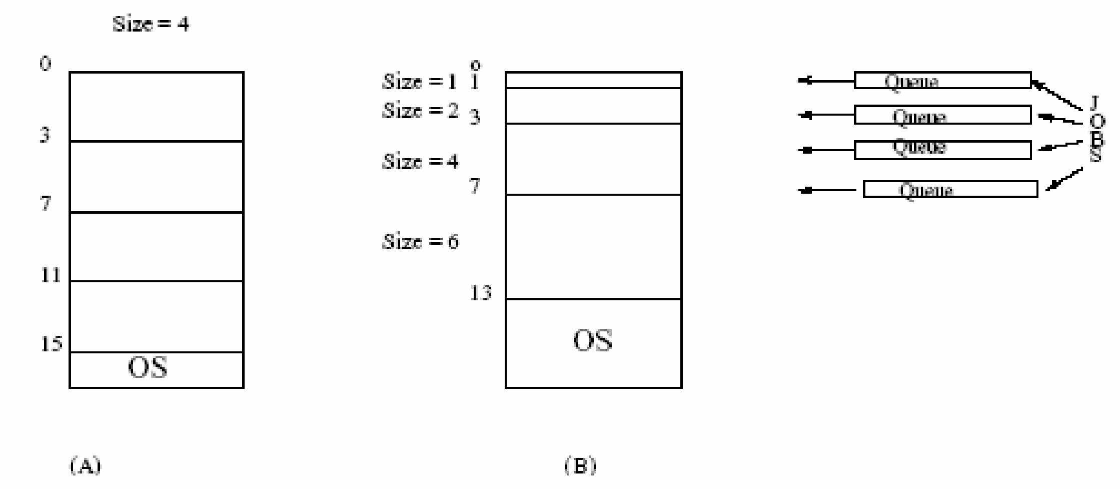


Figure 4.5: Fixed and variable sized partitions.

Such a scheme is difficult to implement. Another scheme which falls between the fixed and dynamic partitioning is a buddy system described next.

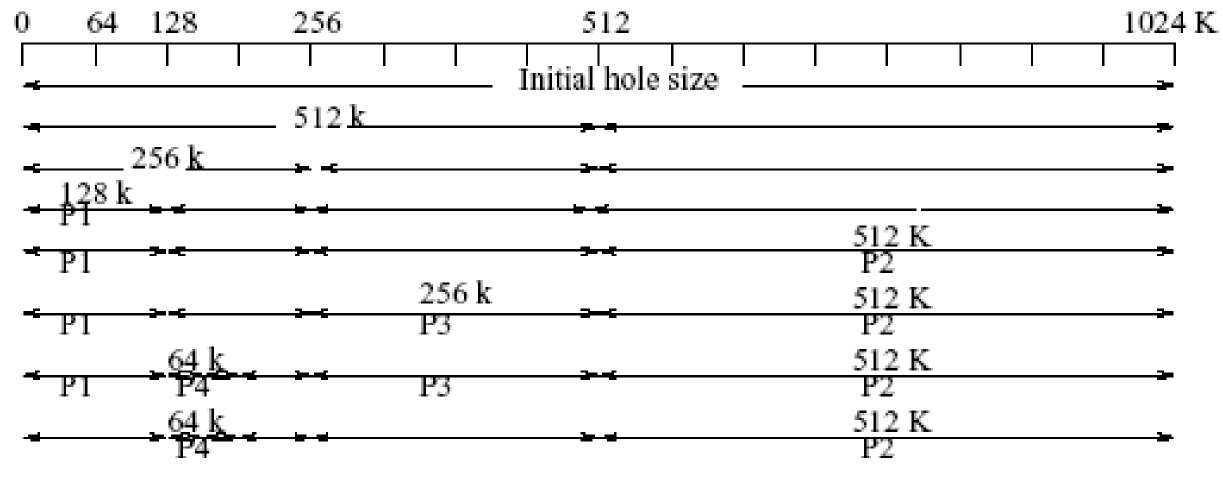


Figure 4.6: Buddy system allocation.

The Buddy system of partitioning: The buddy system of partitioning relies on the fact that space allocations can be conveniently handled in sizes of power of 2. There are two ways in which the buddy system allocates space. Suppose we have a hole which is the closest power of two. In that case, that hole is used for allocation. In case we do not have

that situation then we look for the next power of 2 hole size, split it in two equal halves and allocate one of these. Because we always split the holes in two equal sizes, the two are \buddies". Hence, the name buddy system. We shall illustrate allocation using a buddy system. We assume that initially we have a space of 1024 K. We also assume that processes arrive and are allocated following a time sequence as shown in figure 4.6.

With 1024 K or (1 M) storage space we split it into buddies of 512 K, splitting one of them to two 256 K buddies and so on till we get the right size. Also, we assume scan of memory from the beginning. We always use the first hole which accommodates the process. Otherwise, we split the next sized hole into buddies. Note that the buddy system begins search for a hole as if we had a fixed number of holes of variable sizes but turns into a dynamic partitioning scheme when we do not find the best-fit hole. The buddy system has the advantage that it minimizes the internal fragmentation. However, it is not popular because it is very slow. In Figure 4.6 we assume the requirements as (P1:80 K); (P2:312 K); (P3:164 K); (P4:38 K). These processes arrive in the order of their index and P1 and P3 finish at the same time.

1.9 Virtual Storage Space and Main Memory Partitions

Programming models assume the presence of main memory only. Therefore, ideally we would like to have an unlimited (infinite) main memory available. In fact, an unlimited main memory shall give us a Turing machine capability. However, in practice it is infeasible. So the next best thing is attempted. CPU designers support and generate a very large logical addressable space to support programming concerns. However, the directly addressable main memory is limited and is quite small in comparison to the logical addressable space. The actual size of main memory is referred as the physical memory.

The logical addressable space is referred to as virtual memory. The notion of virtual memory is a bit of an illusion. The OS supports and makes this illusion possible. It does so by copying chunks of disk memory into the main memory as shown in Figure 4.7. In other words, the processor is fooled into believing that it is accessing a large addressable space. Hence, the name virtual storage space. The disk area may map to the virtual space requirements and even beyond.

Besides the obvious benefit that virtual memory offers a very large address space, there is one other major benefit derived from the use of virtual storage. We now can have many more main memory resident active processes. This can be explained as follows. During much of the lifetime of its execution, a process operates on a small set of instructions within a certain neighborhood. The same applies for the data as well. In other words a process makes use of a very small memory area for doing most of the instructions and making references to the data. As explained in Section 4.9, this is primarily due to the locality of reference. So, technically, at any time we need a very small part of a process to really be memory resident. For a moment, let us suppose that this small part is only 1/10th of the process's overall requirements. Note in that case, for the same size of physical main memory, we can service 10 times as many memory resident programs. The next question then is how do we organize and allocate these small chunks of often required areas to be in memory. In fact, this is where paging and segmentation become important. In this context we need to understand some of the techniques of partitioning of main memory into pages or segments.

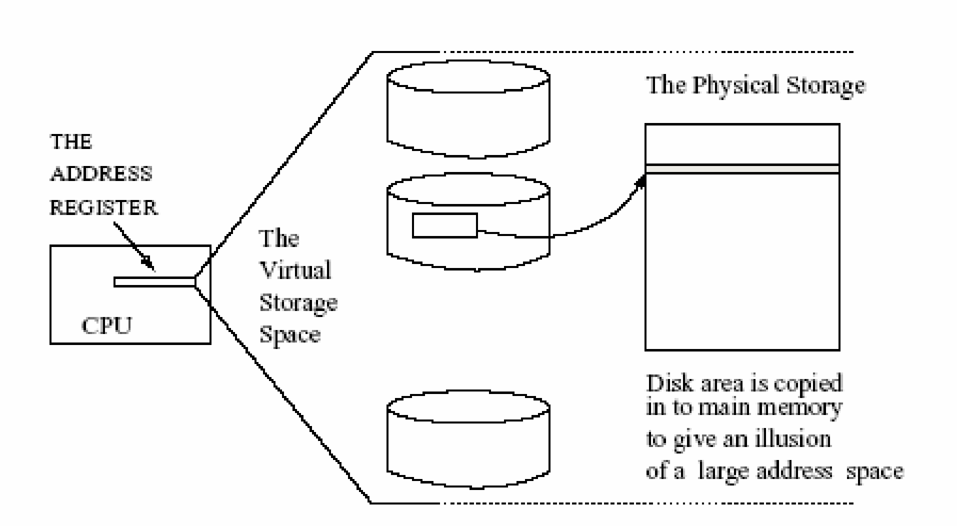


Figure 4.7: Virtual storage concept.

In addition, we need to understand virtual addressing concepts with paging and/or segmentation. We begin with some simple techniques of partitioning both these memories and management of processes.

1.10 Virtual Memory: Paging

In some sense, paging of virtual memory has an underlying mechanism which resembles reading of a book. When we read a book we only need to open only the current page to read. All the other pages are not visible to us. In the same manner, we can argue that even when we may have a large online main memory available, the processor only needs a small set of instructions to execute at any time. In fact, it often happens that for a brief while, all the instructions which the processor needs to execute are within a small proximity of each other. That is like a page we are currently reading in a book. Clearly, this kind of situation happens quite frequently.

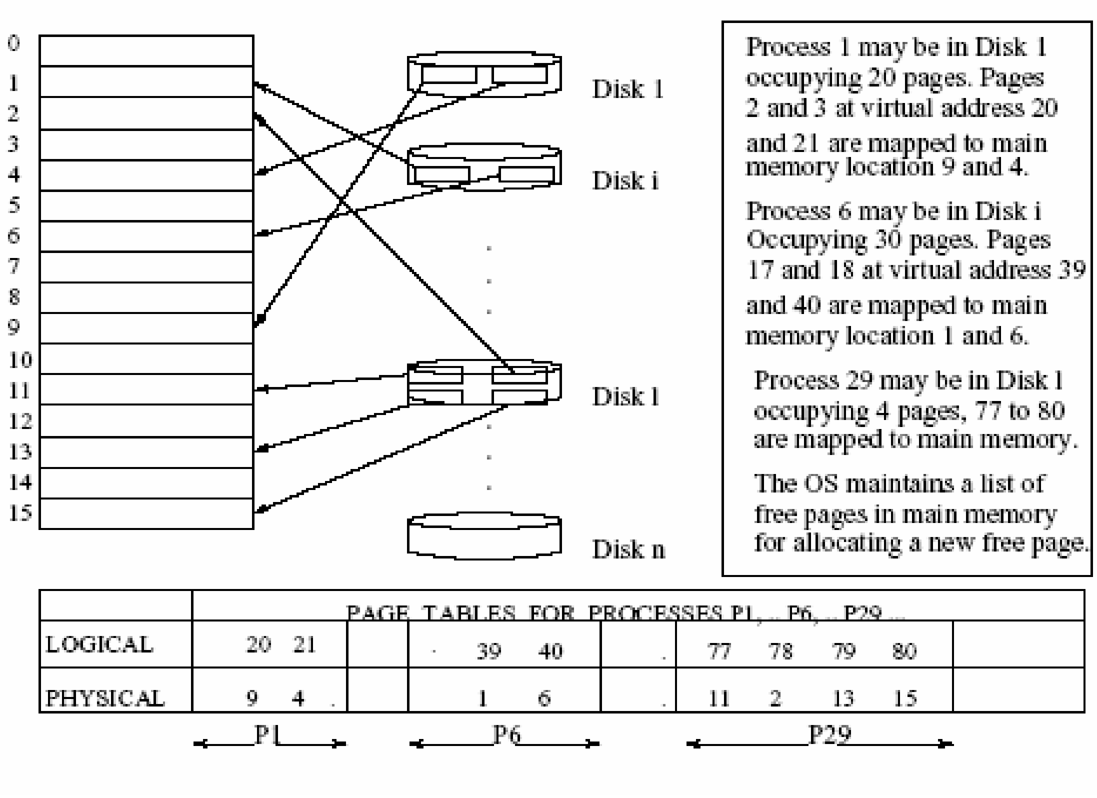
Essentially virtual memory is a large addressable space supported by address generating mechanisms in modern CPUs. Virtual address space is much larger than the physical main memory in a computer system. During its execution, a process mostly generates instruction and data references from within a small range. This is referred to as the locality of reference. Examples of locality of reference abound. For instance, we have locality of reference during execution of a *for* or *while* loop, or a call to a procedure. Even in a sequence of assignment statements, the references to instructions and data are usually within a very small range. Which means, during bursts of process execution, only small parts of all of the instruction and data space are needed, i.e. only these parts need be in the main memory. The remaining process, instructions and data, can be anywhere in the virtual space (i.e. it must remain accessible by CPU but not necessarily in main memory). If we are able to achieve that, then we can actually follow a schedule, in which we support a large address space and keep bringing in that part of process which is needed. This way we can comfortably support (a) multi-programming (b) a large logical addressable space giving enormous freedom to a programmer. Note, however, that this entails mapping of logical addresses into physical address space. Such a mapping assures that the instruction in sequence is fetched or the data required in computation is correctly used.

If this translation were to be done in software, it would be very slow. In fact, nowadays this address translation support is provided by hardware in CPUs. Paging is one of the popular memory management schemes to implement such virtual memory management schemes. OS software and the hardware address translation between them achieve this.

1.10.1 Mapping the Pages

Paging stipulates that main memory is partitioned into frames of sufficiently small sizes. Also, we require that the virtual space is divided into pages of the same size as the frames. This equality facilitates movement of a page from anywhere in the virtual space (on disks) to a frame anywhere in the physical memory. The capability to map “any page" to “any frame" gives a lot of flexibility of operation as shown in Figure 4.8

Division of main memory into frames is like fixed partitioning. So keeping the frame size small helps to keep the internal fragmentation small. Often, the page to frame movement is determined by a convenient size (usually a power of two) which disks also use for their own DMA data transfer. The usual frame size is 1024 bytes, though it is not unusual to have 4 K frame sizes as well. Paging supports multi-programming. In general there can be many processes in main memory, each with a different number of pages. To that extent, paging is like dynamic variable partitioning.



Process 1 may be in Disk 1 occupying 20 pages. Pages 2 and 3 at virtual address 20 and 21 are mapped to main memory location 9 and 4.

Process 6 may be in Disk i Occupying 30 pages. Pages 17 and 18 at virtual address 39 and 40 are mapped to main memory location 1 and 6.

Process 29 may be in Disk l occupying 4 pages, 77 to 80 are mapped to main memory.

The OS maintains a list of free pages in main memory for allocating a new free page.

	PAGE TABLES FOR PROCESSES P1, .. P6, .. P29 ...					
LOGICAL	20 21		· 39 40	.	77 78 79 80	
PHYSICAL	9 4 .		1 6	.	11 2 13 15	

Figure 4.8: Paging implementation.

1.11 Paging: Implementation

Paging implementation requires CPU (HW) and OS (SW) support. In Figure 4.8, we assume presence of three active processes. These processes need to have their pages mapped to the main memory page frames. The OS maintains a page table for every process to translate its logical to physical addresses. The page table may itself be resident in main memory.

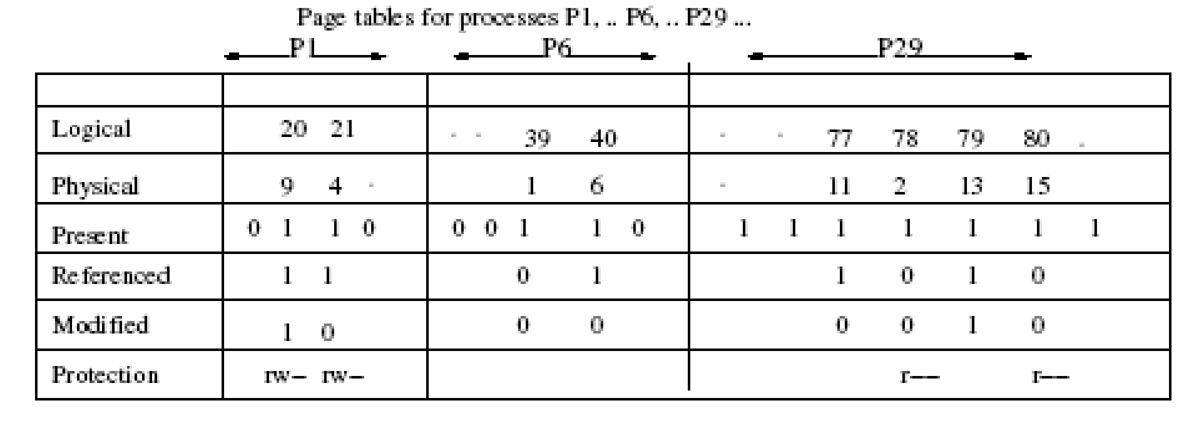
For a process, which is presently active, there are a number of pages that are in the main memory. This set of pages (being used by the process) forms its resident set. With the locality of reference generally observed, most of the time, the processes make reference within the resident set. We define the set of pages needed by a process at any time as the working set. The OS makes every effort to have the resident set to be the same as the working set. However, it does happen (and happens quite often), that a page required for continuing the process is not in the resident set. This is called a page fault. In normal course of operation, though whenever a process makes virtual address reference, its page table is looked up to find if that page is in main memory. Often it is there. Let us now suppose that the page is not in main memory, i.e. a page fault has occurred. In that case, the OS accesses the required page on the disk and loads it in a free page frame. It then makes an entry for this page in process page table. Similarly, when a page is swapped out, the OS deletes its entry from the page table. Sometimes it may well happen that all the page frames in main memory are in use. If a process now needs a page which is not in main memory, then a page must be forced out to make way for the new page. This is done using a page replacement policy discussed next.

1.12 Paging: Replacement

Page replacement policies are based on the way the processes use page frames. In our example shown in Figure 4.9, process P29 has all its pages present in main memory.

Process P6 does not have all its pages in main memory. If a page is present we record 1 against its entry. The OS also records if a page has been referenced to read or to write.

In both these cases a reference is recorded. If a page frame is written into, then a modified bit is set. In our example frames 4, 9, 40, 77, 79 have been referenced and page frames 9 and 13 have been modified. Sometimes OS may also have some information about protection using *rwe* information. If a reference is made to a certain virtual address and its corresponding page is not present in main memory, then we say a page fault has occurred. Typically, a page fault is followed by moving in a page. However, this may require that we move a page out to create a space for it. Usually this is done by using an appropriate page replacement policy to ensure that the throughput of a system does not suffer. We shall later see how a page replacement policy can affect performance of a system.



Page tables for processes P1, .. P6, .. P29 ...

	P1	P6	P29
Logical	20 21	. . 39 40	. . 77 78 79 80 .
Physical	9 4 .	1 6	. 11 2 13 15
Present	0 1 1 0	0 0 1 1 0	1 1 1 1 1 1 1
Referenced	1 1	0 1	1 0 1 0
Modified	1 0	0 0	0 0 1 0
Protection	rw– rw–		r–– r––

Figure 4.9: Replacement policy.

1.12.1 Page Replacement Policy

Towards understanding page replacement policies we shall consider a simple example of a process P which gets an allocation of four pages to execute. Further, we assume that the OS collects some information (depicted in Figure 4.10) about the use of these pages as this process progresses in execution. Let us examine the information depicted in figure 4.10 in some detail to determine how this may help in evolving a page replacement policy. Note that we have the following information available about P.

1. The time of arrival of each page. We assume that the process began at some time with value of time unit 100. During its course of progression we now have pages that have been loaded at times 112, 117 119, and 120.
2. The time of last usage. This indicates when a certain page was last used. This entirely depends upon which part of the process P is being executed at any time.

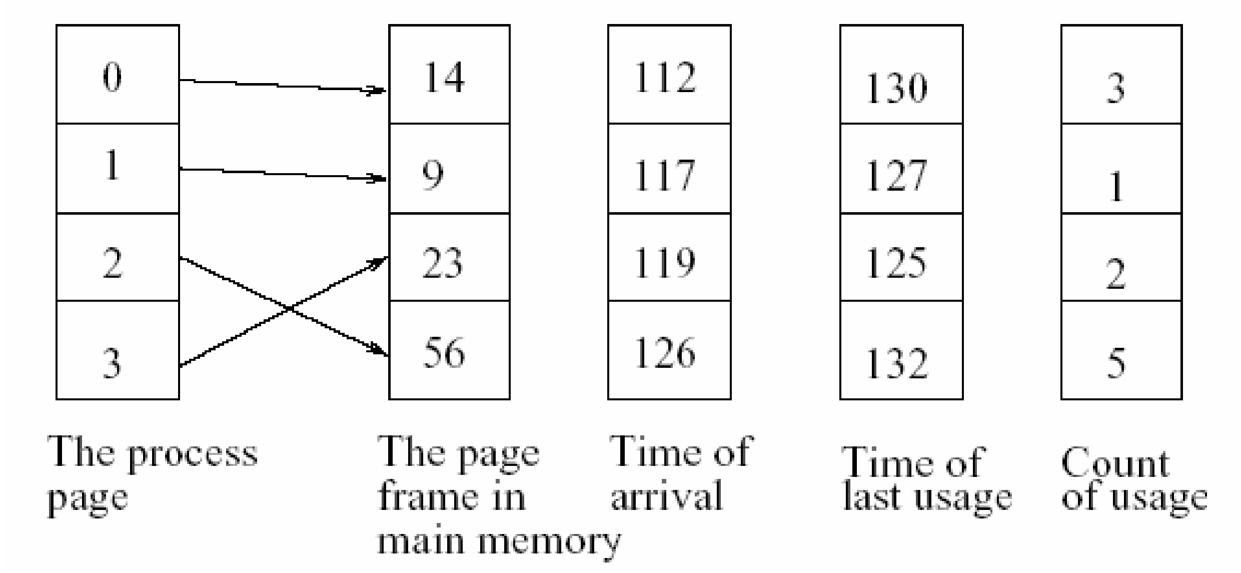


Figure 4.10: Information on page usage policy.

3. The frequency of use. We have also maintained the frequency of use over some fixed interval of time T in the immediate past. This clearly depends upon the nature of control flow in process P.

As an example we may say that page located at 23 which was installed at time 119, was last used at time unit 125 and over the time period T the process P made two references to it. Based on the above pieces of information if we now assume that at time unit 135 the process P experiences a page-fault, what should be done. Based on the choice of the policy and the data collected for P, we shall be able to decide which page to swap out to bring in a new page.

FIFO policy: This policy simply removes pages in the order they arrived in the main memory. Using this policy we simply remove a page based on the time of its arrival in the memory. Clearly, use of this policy would suggest that we swap page located at 14 as it arrived in the memory earliest.

LRU policy: LRU expands to least recently used. This policy suggests that we re- move a page whose last usage is farthest from current time. Note that the current time is 135 and the least recently used page is the page located at 23. It was used last at time unit 125 and every other page is more recently used. So, page 23 is the least recently used page and so it should be swapped if LRU replacement policy is employed.

NFU policy: NFU expands to not frequently used. This policy suggests to use the criterion of the count of usage of page over the interval T. Note that process P has not made use of page located at 9. Other pages have a count of usage like 2, 3 or even 5 times. So the basic argument is that these pages may still be needed as compared to the page at 9. So page 9 should be swapped.

Let us briefly discuss the merits of choices that one is offered. FIFO is a very simple policy and it is relatively easy to implement. All it needs is the time of arrival. However, in following such a policy we may end up replacing a page frame that is referred often during the lifetime of a process. In other words, we should examine how useful a certain page is before we decide to replace it. LRU and NFU policies are certainly better in that regard but as is obvious we need to keep the information about the usage of the pages by the process. In following the not frequently used (NFU) and least recently used (LRU) page replacement policies, the OS needs to define *recency*. As we saw recency is defined as a fixed time interval proceeding the current time. With a definition of recency, we can implement the policy framework like least recently used (LRU). So one must choose a proper interval of time. Depending upon the nature of application environment and the work load a choice of duration of recency will give different throughput from the system. Also, this means that the OS must keep a tab on the pages which are being used and how often these are in use. It is often the case that the most recently used pages are likely to be the ones used again. On the whole one can sense that the LRU policy should be statistically better than FIFO.

A more advanced technique of page replacement policy may look-up the likely future references to pages. Such a policy frame would require use of some form of predictive techniques. In that case, one can prevent too many frequent replacements of pages which prevents thrashing as discussed in the subsection. 1.12.2.

Let us for now briefly pay our attention to page references resulting in a page hit and a page miss. When we find that a page frame reference is in the main memory then we have a page hit and when page fault occurs we say we have a page miss. As is obvious from the discussion, a poor choice of policy may result in lot of page misses. We should be able to determine how it influences the throughput of a system. Let us assume that we have a system with the following characteristics.

- Time to look-up page table: 10 time units.
- Time to look-up the information from a page frame (case of a page hit): 40 time units.
- Time to retrieve a page from disk and load it and finally access the page frame (case of a page miss): 190 time units.

Now let us consider the following two cases when we have 50% and 80% page hits. We shall compute the average time to access.

- Case 1: With 50% page hits the average access time is ((10+40) * 0:5) + (10+190) * 0:5 =125 time units.
- Case 2: With 80% page hits the average access time is (10+40) * 0:8) + (10+190) * 0:2 = 80 time units.

Clearly, the case 2 is better. The OS designers attempt to offer a page replacement policy which will try to minimize the page miss. Also, sometimes the system programmers have to tune an OS to achieve a high efficacy in performance by ensuring that page miss cases are within some tolerable limits. It is not unusual to be able to achieve over 90% page hits when the application profile is very well known.

There is one other concern that may arise with regard to page replacement. It may be that while a certain process is operative, some of the information may be often required. These may be definitions globally defined in a program, or some terminal related IO information in a monitoring program. If this kind of information is stored in certain pages then these have to be kept at all times during the lifetime of the process. Clearly, this requires that we have these pages identified. Some programming environments allow

directives like keep to specify such information to be available at all the time during the lifetime of the process. In Windows there is a keep function that allows one to specify which programs must be kept at all the time. The Windows environment essentially uses the keep function to load TSR (terminate and stay resident) programs to be loaded in the memory 1. Recall, earlier we made a reference to thrashing which arises from the overheads generated from frequent page replacement. We shall next study that.

1.12.2 Thrashing

Suppose there is a process with several pages in its resident set. However, the page replacement policy results in a situation such that two pages alternatively move in and out of the resident set. Note that because pages are moved between main memory and disk, this has an enormous overhead. This can adversely affect the throughput of a system. The drop in the level of system throughput resulting from frequent page replacement is called thrashing. Let us try to comprehend when and how it manifests. Statistically, on introducing paging we can hope to enhance multi-programming as well as locality of reference. The main consequence of this shall be enhanced processor utilization and hence, better throughput. Note that the page size influences the number of pages and hence it determines the number of resident sets we may support. With more programs in main memory or more pages of a program we hope for better locality of reference. This is seen to happen (at least initially) as more pages are available. This is because, we may have more effective locality of reference as well as multi-programming. However, when the page size becomes too small we may begin to witness more page-faults.

Incidentally, a virus writer may employ this to mount an attack. For instance, the keep facility may be used to have a periodic display of some kind on the victim's screen. More page-faults would result in more frequent disk IO. As disk IO happens more often the throughput would drop. The point when this begins to happen, we say thrashing has occurred. In other words, the basic advantage of higher throughput from a greater level of utilization of processor and more effective multi-programming does not accrue any more. When the advantage derived from locality of reference and multi-programming begins to vanish, we are at the point when thrashing manifests. This is shown in Figure 4.11.

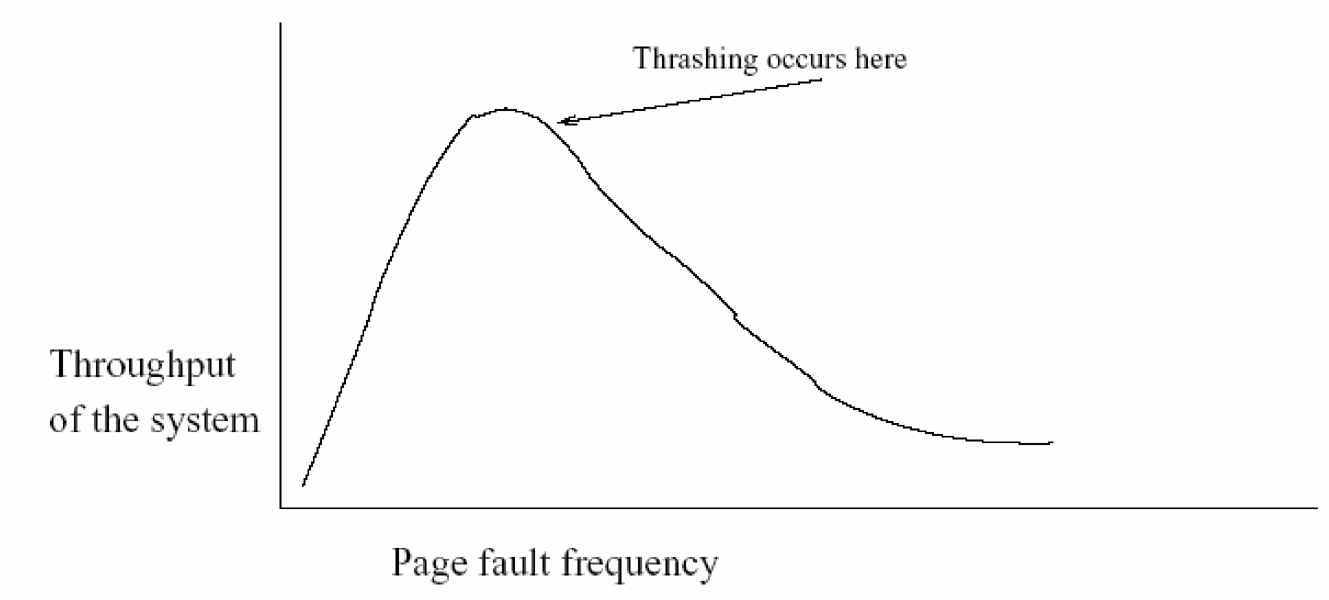


Figure 4.11: Thrashing on numerous page fault.

1.13 Paging: HW support

Recall we emphasized that we need HW within CPU to support paging. The CPU generates a logical address which must get translated to a physical address. In Figure 4.12 we indicate the basic address generation and translation.

Let us trace the sequence of steps in the generation of address.

- The process generates a logical address. This address is interpreted in two parts.
- The first part of the logical address identifies the virtual page.
- The second part of the logical address gives the offset within this page.
- The first part is used as an input to the page table to find out the following:

  * Is the page in the main memory?

  * What is the page frame number for this virtual page?
- The page frame number is the first part of the physical memory address.
- The offset is the second part of the correct physical memory location.

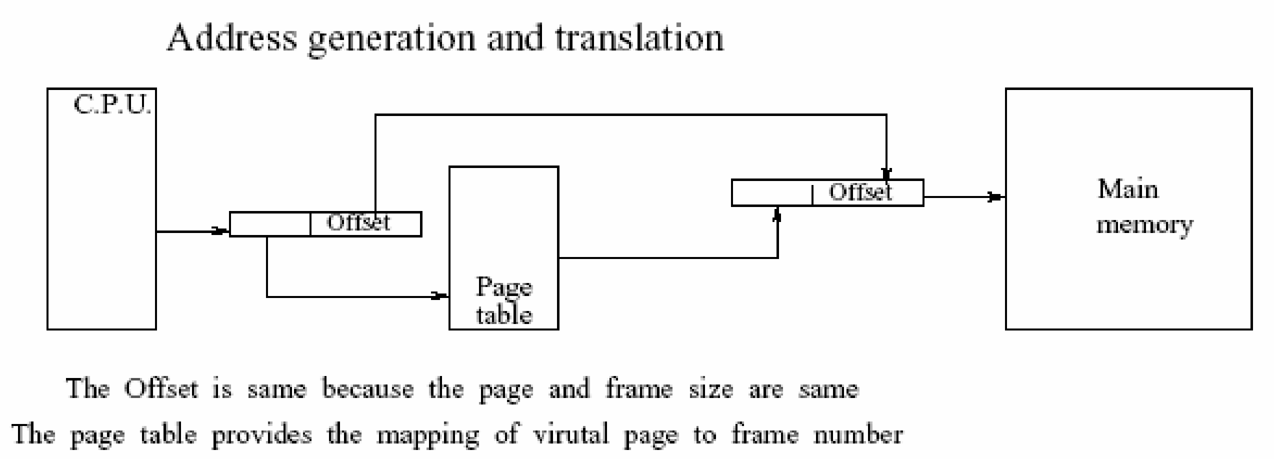

Address generation and translation
C.P.U.
Offset
Page table
Offset
Main memory
The Offset is same because the page and frame size are same
The page table provides the mapping of virutal page to frame number

Figure 4.12: Hardware support for paging.

If the page is not in the physical memory, a page-fault is generated. This is treated as a trap. The trap then suspends the regular sequence of operations and fetches the required page from the disk into main memory.

We next discuss a relatively simple extension of the basic paging scheme with hardware support. This scheme results in considerable improvement in page frame access.

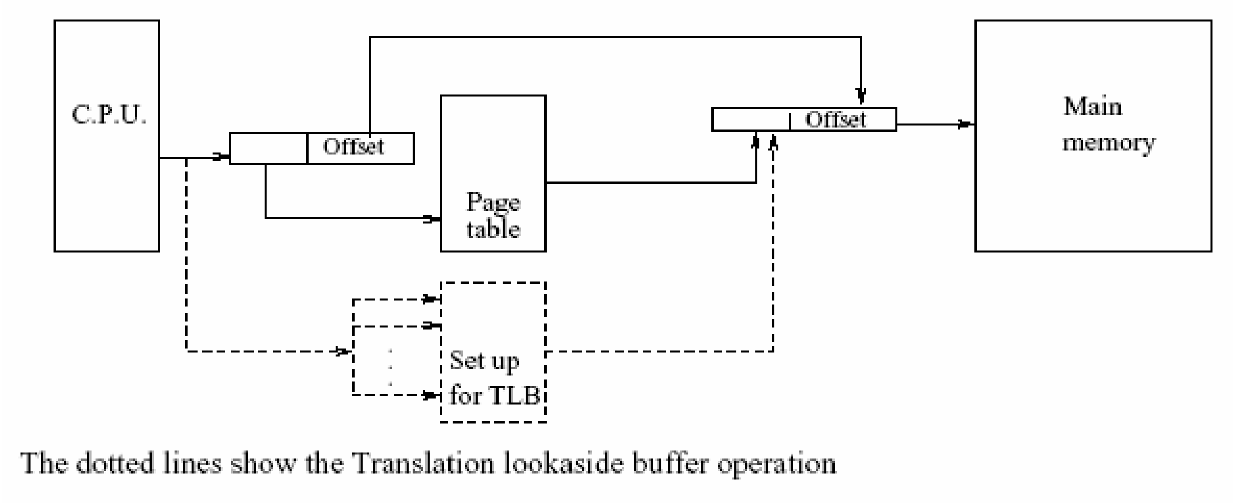

C.P.U.
Offset
Page table
Offset
Main memory
Set up for TLB
The dotted lines show the Translation lookaside buffer operation

Figure 4.13: Paging with translation look-aside buffer.

1.13.1 The TLB scheme

The basic idea in the translation look-aside buffer access is quite simple. The scheme is very effective in improving the performance of page frame access. The scheme employs a cache buffer to keep copies of some of the page frames in a cache buffer. This buffer is also interrogated for the presence of page frame copy. Note that a cache buffer is implemented in a technology which is faster than the main memory technology. So, a retrieval from the cache buffer is faster than that from the main memory. The hardware signal which looks up the page table is also used to look up (with address translation) to check if the cache buffer on a side has the desired page. This nature of look-up explains why this scheme is called Translation Look-aside Buffer (TLB) scheme. The basic TLB buffering scheme is shown in Figure 4.13. Note that the figure replicates the usual hardware support for page table look-up. So, obviously the scheme cannot be worse than the usual page table look-up schemes. However, since a cache buffer is additionally maintained to keep some of the frequently accessed pages, one can expect to achieve an improvement in the access time required for those pages which obtain a page hit for presence in the buffer. Suppose we wish to access page frame p. The following three possibilities may arise:

1. Cache presence: There is a copy of the page frame p. In this case it is procured from the look- aside buffer which is the cache.
2. Page table presence: The cache does not have a copy of the page frame p, but page table access results in a page hit. The page is accessed from the main memory.
3. Not in page table: This is a case when the copy of the page frame is neither in the cache buffer nor does it have an entry in the page table. Clearly, this is a case of page-fault. It is handled exactly as the page-fault is normally handled.

Note that if a certain page frame copy is available in the cache then the cache look-up takes precedence and the page frame is fetched from the cache instead of fetching it from the main memory. This obviously saves time to access the page frame. In the case the page hit occurs for a page not in cache then the scheme ensures its access from the main memory. So it is at least as good as the standard paging scheme with a possibility of improvement whenever a page frame copy is in cache buffer.

1.13.2 Some Additional Points

Since page frames can be loaded anywhere in the main memory, we can say that paging mechanism supports dynamic relocation. Also, there are other schemes like multi-level page support systems which support page tables at multiple levels of hierarchy. In addition, there are methods to identify pages that may be shared amongst more than one process. Clearly, such shareable pages involve additional considerations to maintain consistency of data when multiple processes try to have read and write access. These are usually areas of research and beyond the scope of this book.

1.14 Segmentation

Like paging, segmentation is also a scheme which supports virtual memory concept. Segmentation can be best understood in the context of a program's storage requirements.

One view could be that each part like its code segment, its stack requirements (of data, nested procedure calls), its different object modules, etc. has a contiguous space. This space would then define a process's space requirement as an integrated whole (or complete space). As a view, this is very uni-dimensional.

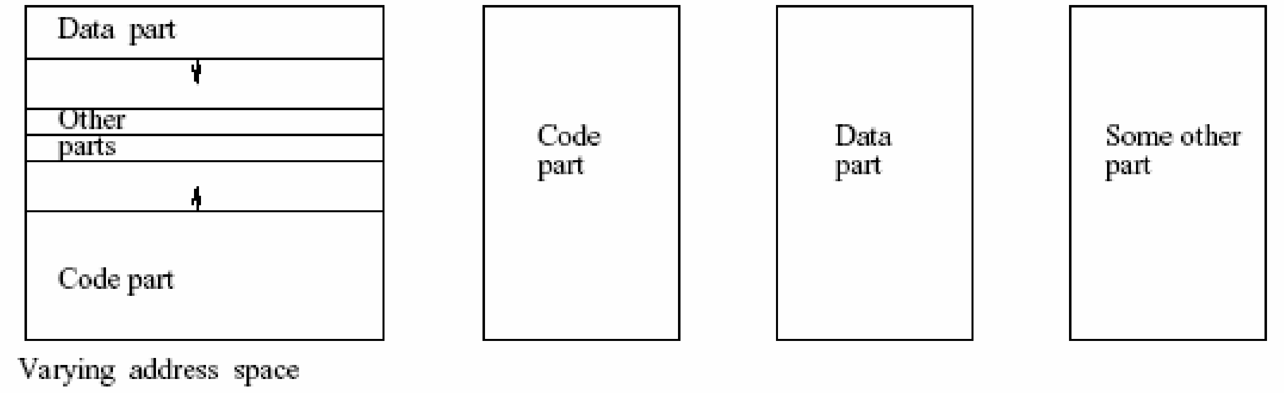


Figure 4.14: Segmentation scheme: A two dimensional view.

In using segmentation, one recognizes various segments such as the stack, object code, data area etc. Each segment has requirements that vary over time. For instance, stacks grow and shrink the memory requirements of object and data segments may change during the lifetime of the process. This may depend on which functions have been called and are currently active. It is, therefore, best to take a two-dimensional view of a process's memory requirement. In this view, each of the process segments has an opportunity to acquire a variable amount of space over time. This ensures that one area does not run into the space of any other segment. The basic scheme is shown in Figure 4.14. The implementation of segmentation is similar to paging, except that we now have segment table (in place of a page table) look-ups to identify addresses in each of the segments. HW supports a table look-up for a segment and an offset within that segment.

We may now compare paging with segmentation.

- Paging offers the simplest mechanism to effect virtual addressing.
- While paging suffers from internal fragmentation, segmentation suffers from external fragmentation.
- One of the advantages segmentation clearly offers is separate compilation of each segment with a view to link up later. This has another advantage. A user may develop a code segment and share it amongst many applications. He generates the required links at the time of launching the application. However, note that this also places burden on the programmer to manage linking. To that extent paging offers greater transparency in usage.
- In paging, a process address space is linear. Hence, it is uni-dimensional. In a segment based scheme each procedure and data segment has its own virtual space mapping. Thus the segmentation assures a much greater degree of protection.

- In case a program's address space fluctuates considerably, paging may result in frequent page faults. Segmentation does not suffer from such problems.
- Paging partitions a program and data space uniformly and is, therefore, muchsimpler to manage. However, one cannot easily distinguish data space fromprogram space in paging. Segmentation partitions process space requirementsaccording to a logical division of the segments that make up the process.Generally, this simplifies protection.

Clearly, a clever scheme with advantages of both would be: segmentation with paging. In such a scheme each segment would have a descriptor with its pages identified. Such a scheme is shown in Figure 4.15. Note that we have to now use three sets of offsets. First, a segment offset helps to identify the set of pages. Next, within the corresponding page table (for the segment), we need to identify the exact page table. This is done by using the page table part of the virtual address. Once the exact page has been identified, the offset is used to obtain main memory address reference. The final address resolution is exactly as we saw in Section 4.9 where we first discussed paging.

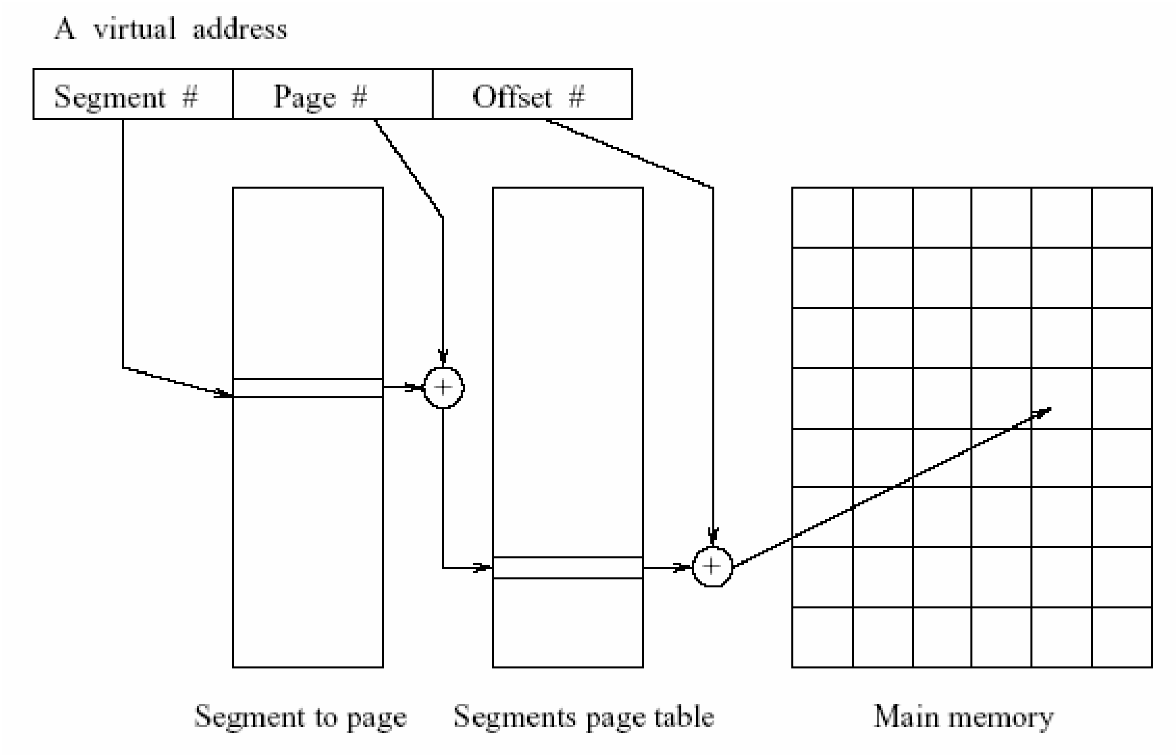


Figure 4.15: Segmentation with paging.

In practice, there are segments for the code(s), data, and stack. Each segment carries the *rwe* information as well. Usually the stack and data have read write permissions but no execute permissions. Code rarely has write permission but would have a read and execute permission.

Block -2

Unit-1

1.1 Introduction to Input Output (IO) Management

So far we have studied how resources like processor and main memory are managed. We shall now examine the I/O management. Humans interact with machines by providing information through IO devices. Also, much of whatever a computer system provides as on-line services is essentially made available through specialized devices such as screen displays, printers, keyboards, mouse, etc. Clearly, management of all these devices can affect the throughput of a system. For this reason, input output management also becomes one of the primary responsibilities of an operating system. In this chapter we shall examine the role of operating systems in managing IO devices. In particular, we shall examine how the end use of the devices determines the way they are regulated for communication with either humans or with systems.

1.2 Issues in IO Management

Let us first examine the context of input output in a computer system. We shall look at issues initially from the point of view of communication with a device. Later, in Section 1.2.1, we shall also examine issues from the point of view of managing events. When we analyze device communication, we notice that communication is required at the following three levels:

- ➢ The need for a human to input information and receive output from a computer.
- ➢ The need for a device to input information and receive output from a computer.
- ➢ The need for computers to communicate (receive/send information) over networks.

The first kind of IO devices operate at rates good for humans to interact. These may be character-oriented devices like a keyboard or an event-generating device like a mouse. Usually, human input using a key board will be a few key depressions at a time. This means that the communication is rarely more than a few bytes. Also, the mouse events can be encoded by a small amount of information (just a few bytes). Even though a human input is very small, it is stipulated that it is very important, and therefore requires an immediate response from the system. A communication which attempts to draw attention often requires the use of an interrupt mechanism or a programmed data mode of operation.

The second kind of IO requirement arises from devices which have a very high character density such as tapes and disks. With these characteristics, it is not possible to regulate communication with devices on a character by character basis. The information transfer, therefore, is regulated in blocks of information. Additionally, sometimes this may require some kind of format control to structure the information to suit the device and/or data characteristics. For instance, a disk drive differs from a line printer or an image scanner. For each of these devices, the format and structure of information is different. It should be observed that the rate at which a device may provide data and the rates at which an end application may consume it may be considerably different. In spite of these differences, the OS should provide uniform and easy to use IO mechanisms. Usually, this is done by providing a buffer. The OS manages this buffer so as to be able to comply with the requirements of both the producer and consumer of data. In section **1.5** we discuss the methods to determine buffer sizes.

The third kind of IO requirements emanate from the need to negotiate system IO with the communications infrastructure. The system should be able to manage communications traffic across the network. This form of IO facilitates access to internet resources to support e-mail, file-transfer amongst machines or Web applications. Additionally now we have a large variety of options available as access devices. These access devices may be in the form of Personal Digital Assistant (PDA), or mobile phones which have infrared or wireless enabled communications. This rapidly evolving technology makes these forms of communications very challenging. It is beyond the scope of this book to have a discussion on these technologies, devices or mechanisms. Even then it should be remarked that most network cards are direct memory access (DMA) enabled to facilitate DMA mode of IO with communication infrastructure. We shall discuss DMA in Section **1.3.5** Typically the character-oriented devices operate with speeds of tens of bytes per second (for keyboards, voice-based input, mouse, etc.). The second kind of devices operate over a much wider range. Printers operate at 1 to 2 KB per second, disks transfer at rates of 1 MB per second or more. The graphics devices fall between these two ranges while the graphics cards may in fact be even faster. The devices communicate with a machine using a data bus and a device controller. Essentially, all these devices communicate large data blocks. However, the communication with networks differs from the way the communication takes place with the block devices. Additionally, the communication with wireless devices may differ from that required for internet services. Each of these cases has its own information management requirements and OS must negotiate with the medium to communicate. Therefore, the nature of the medium controls the nature of protocol which may be used to support the needed communication.

One of the important classes of OS is called Real-time Operating Systems or RTOS for short. We shall study RTOS in a later chapter. For now, let us briefly see what distinguishes an RTOS from a general purpose OS. RTOSs are employed to regulate a process and generate responses to events in its application environments within a stipulated time considered to be real-time response time. RTOS may be employed to regulate a process or even offer transaction oriented services like on-line reservation system, etc. The main point of our concern here is to recognize the occurrence of certain events or event order. A key characteristic of embedded systems is to recognize occurrence of events which may be by monitoring variable values or identifying an event.

1.2.1 Managing Events

Our next observation is that a computer system may sometimes be embedded to interact with a real-life activity or process. It is quite possible that in some operational context a process may have to synchronize with some other process. In such a case this process may actually have to wait to achieve a rendezvous with another process. In fact, whichever of the two synchronizing processes arrives first at the point of rendezvous would have to wait. When the other process also reaches the point of synchronization, the first process may proceed after recognizing the synchronizing event. Note that events may be communicated using signals which we shall learn about later.

In some other cases a process may have to respond to an asynchronous event that may occur at any time. Usually, an asynchronous input is attended to in the next instruction cycle as we saw in Section 1.2. In fact, the OS checks for any event which may have occurred in the intervening period. This means that an OS incorporates some IO event recognition mechanism. IO handling mechanisms may be like polling, or a programmed data transfer, or an interrupt mechanism, or even may use a direct memory access (DMA) with cycle stealing. We shall examine all these mechanisms in some detail in Section 1.3. The unit of data transfer may either be one character at a time or a block of characters. It may require to set up a procedure or a protocol. This is particularly the case when a machine-to-machine or a process-to-process communication is required. Additionally, in these cases, we need to account for the kind of errors that may occur.

We also need procedure to recover when such an error occurs. We also need to find ways to ensure the security and protection of information when it is in transit. Yet another important consideration in protection arises when systems have to share devices like printers. We shall deal with some of these concerns in the next module on resource sharing. In the discussions above we have identified many issues in IO. For now let us look at how IO mechanisms are organized and how they operate.

1.3 IO Organization

In the previous section we discussed various issues that arise from the need to support a wide range of devices. To meet these varied requirements, a few well understood modalities have evolved over time. The basic idea is to select a mode of communication taking device characteristics into account or a need to synchronize with some event, or to just have a simple strategy to ensure a reliable assured IO.

Computers employ the following four basic modes of IO operation:

1. Programmed mode
2. Polling mode
3. Interrupt mode
4. Direct memory access mode.

We shall discuss each of these modes in some detail now.

1.3.1 Programmed Data Mode

In this mode of communication, execution of an IO instruction ensures that a program shall not advance till it is completed. To that extent one is assured that IO happens before anything else happens. As depicted in Figure 5.1, in this mode an IO instruction is issued to an IO device and the program executes in “busy-waiting” (idling) mode till the IO is completed. During the busy-wait period the processor is continually interrogating to check if the device has completed IO. Invariably the data transfer is accomplished through an identified register and a flag in a processor. For example, in Figure 5.1 depicts

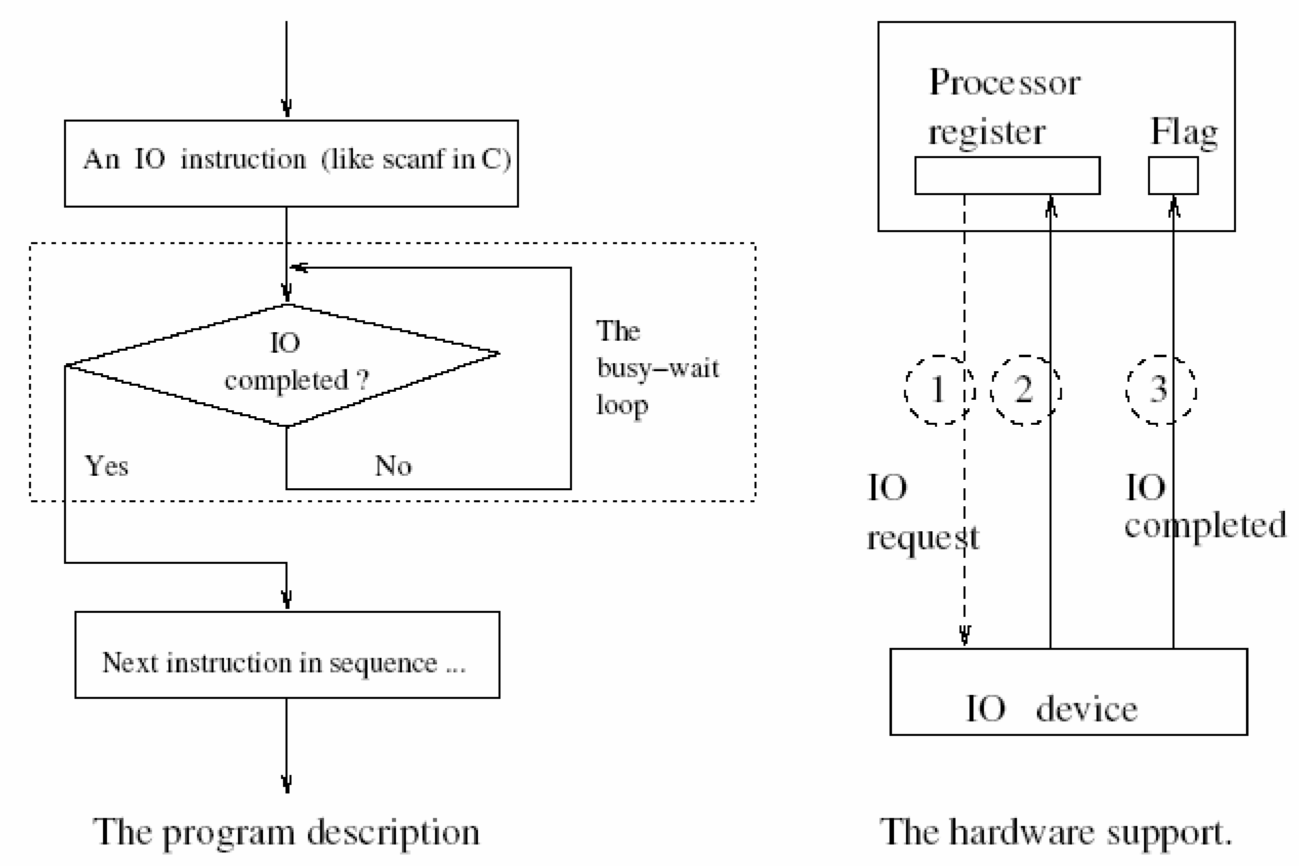


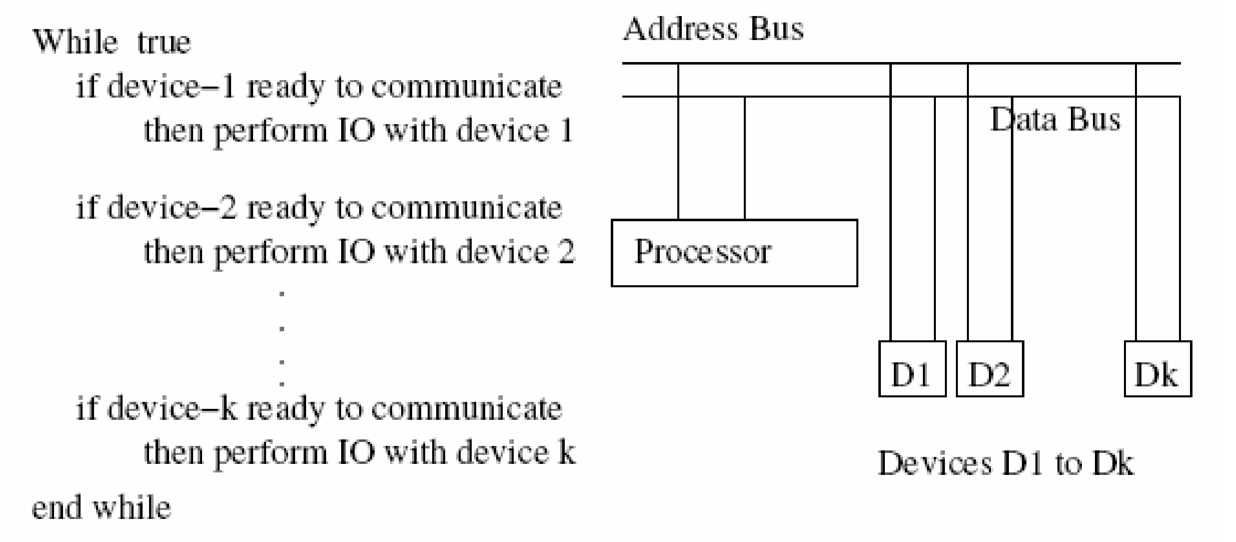
Figure 5.1: Programmed mode of IO.

how an input can happen using the programmed data mode. First the processor issues an IO request (shown as 1), followed by device putting a data in a register (shown as 2) and finally the flag (which is being interrogated) is set (shown as 3). The device either puts a data in the register (as in case of input) or it picks up data from the register (in case of output). When the IO is accomplished it signals the processor through the flag. During the busy-wait period the processor is busy checking the flag. However, the processor is idling from the point of view of doing anything useful. This situation is similar to a car engine which is running when the car is not in motion – essentially "idling".

1.3.2 Polling

In this mode of data transfer, shown in Figure 5.2, the system interrogates each device in turn to determine if it is ready to communicate. If it is ready, communication is initiated and subsequently the process continues again to interrogate in the same sequence. This is just like a round-robin strategy. Each IO device gets an opportunity to establish Communication in turn. No device has a particular advantage (like say a priority) over other devices. Polling is quite commonly used by systems to interrogate ports on a network. Polling may also be scheduled to interrogate at some pre-assigned time intervals. It should be remarked here that most daemon software operate in polling mode. Essentially, they use a while true loop as shown in Figure 5.2. In hardware, this may typically translate to the following protocol:

1. Assign a distinct address to each device connected to a bus.
2. The bus controller scans through the addresses in sequence to find which device wishes to establish a communication.
3. Allow the device that is ready to communicate to leave its data on the register.
4. The IO is accomplished. In case of an input the processor picks up the data. In case of an output the device picks up the data.
5. Move to interrogate the next device address in sequence to check if it is ready to communicate.



```
While  true
   if device−1 ready to communicate
         then perform IO with device 1

   if device−2 ready to communicate
         then perform IO with device 2
                     .
                     .
                     .
   if device−k ready to communicate
         then perform IO with device k
end while
```

Figure 5.2: Polling mode of IO.

As we shall see next, polling may also be used within an interrupt service mode to identify the device which may have raised an interrupt.

1.3.3 Interrupt Mode

Let us begin with a simple illustration to explain the basic rationale behind interrupt mode of data transfer. Suppose a program needs input from a device which communicates using interrupt. Even with the present-day technology the devices are one thousand or more times slower than the processor. So if the program waits on the input device it would cycle through many processor cycles just waiting for the input device to be ready to communicate. This is where the interrupt mode of communication scores. To begin with, a program may initiate IO request and advance without suspending its operation. At the time when the device is actually ready to establish an IO, the device raises an interrupt to seek communication. Immediately the program execution is suspended temporarily and current state of the process is stored. The control is passed on to an interrupt service routine (which may be specific to the device) to perform the desired input. Subsequently, the suspended process context is restored to resume the program from the point of its suspension.

Interrupt processing may happen in the following contexts:

- **Internal Interrupt**: The source of interrupt may be a memory resident process or a function from within the processor. We regard such an interrupt as an internal interrupt. A processor malfunction results in an internal interrupt. An attempt to divide by zero or execute an illegal or non-existent instruction code results in an internal interrupt as well. A malfunction arising from a division by zero is called a trap. Internal interrupt may be caused by a timer as well. This may be because either the allocated processor time slice to a process has elapsed or for some reason the process needs to be pre-empted. Note that an RTOS may pre-empt a running process by using an interrupt to ensure that the stipulated response time required is met. This would also be a case of internal interrupt.
- **External Interrupt:** If the source of interrupt in not internal, i.e. it is other than a process or processor related event then it is an external interrupt. This may be caused by a device which is seeking attention of a processor. As indicated earlier, a program may seek an IO and issue an IO command but proceed. After a while, the device from which IO was sought is ready to communicate. In that case the device may raise an interrupt. This would be a case of an external interrupt.

- **Software Interrupt:** Most OSs offer two modes of operation, the user mode and the system mode. Whenever a user program makes a system call, be it for IO or a special service, the operation must have a transition from user mode to system mode. An interrupt is raised to effect this transition from user to system mode of operation. Such an interrupt is called a software interrupt.

We shall next examine how an interrupt is serviced. Suppose we are executing an instruction at *i* in program P when interrupt signal has been raised. Let us also assume that we have an interrupt service routine which is to be initiated to service the interrupt. The following steps describe how a typical interrupt service may happen.

- Suspend the current program P after executing instruction i.
- Store the address of instruction at *i + 1* in P as the return address. Let us denote this address as PADDR_{i+1}. This is the point at which program P shall resume its execution following the interrupt service. The return address is essentially the incremented program counter value. This may be stored either in some specific location or in some data structure (like a stack or an array). The transfer of control to an interrupt service routine may also be processed like a call to a subroutine. In that case, the return address may even be stored in the code area of the interrupt service routine. Let us identify the location where we stored PADDR_{i+1} as the address *RESTORE*. Later, in step-4, we shall see how storing the return address helps to restore the original sequence of program starting at PADDR_{i+1}.
- Execute a branch unconditionally to transfer control to the interrupt service instructions. The immediately following instruction cycle initiates the interrupt service routine.
- Typically the last instruction in the service routine executes a branch indirect from the location RESTORE. This restores the program counter to take the next instruction at PADDR_{i+1}. Thus the suspended program P obtains the control of the processor again.

1.3.4 Issues in Handling Interrupts

There are many subtle issues and points that need clarification. We shall examine some of these in some detail next.

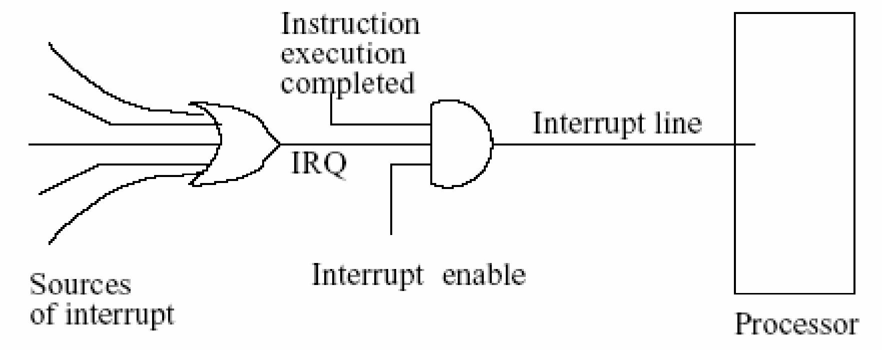


Figure 5.3: How an interrupt is detected.

- **Detecting an interrupt**: In Figure 5.3 we depict how a processor may detect an interrupt request. When a processor has an interrupt enable signal up, then at the end of an instruction cycle we shall recognize an interrupt if the interrupt request (IRQ) line is up. Once an interrupt is recognized, interrupt service shall be required. Two minor points now arise. One is when is interrupt enabled. The other is how a device which raises the interrupt is identified.
- **Identifying the source of interrupt:** As we saw in Figure 5.3 many devices that wish to raise interrupt are gated in via an OR gate to raise an interrupt request (IRQ). Usually, the requests from a set of devices with equal priority would be pooled. Such a pool may have a set of devices under a cluster controller or an IO processor of some kind. One way of determining is to use polling which we discussed earlier. Another mechanism may be to have a daisy-chain arrangement of devices as shown in Figure 5.4. We notice that the normally closed position of the switch allows interrupt to be raised from the devices lower down in the chain.

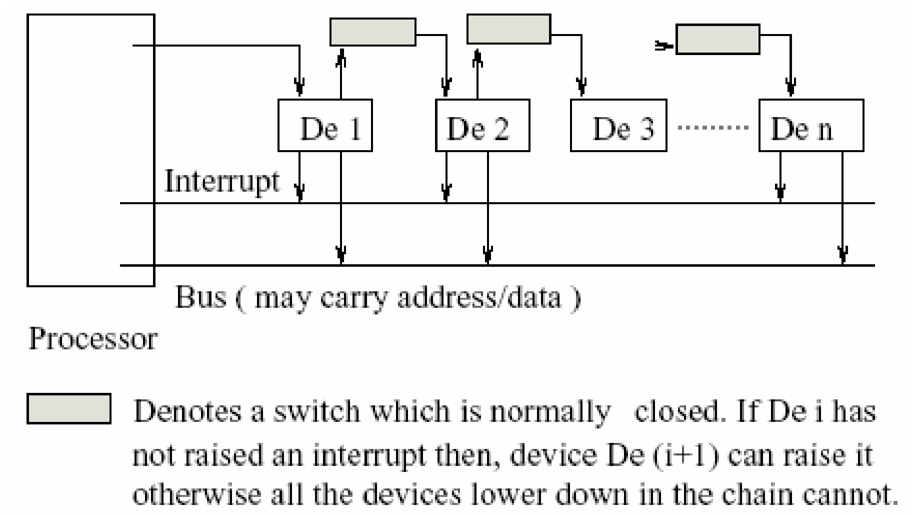

De 1
De 2
De 3
De n
Interrupt
Bus (may carry address/data)
Processor
Denotes a switch which is normally closed. If De i has not raised an interrupt then, device De (i+1) can raise it otherwise all the devices lower down in the chain cannot.

Figure 5.4: Daisy chained devices.

The interrupting device raises the interrupt and the address or data may then be asserted on the bus as shown in Figure 5.4.

It is also possible that there may be an interrupt raised through an IO cluster (which may be for a SCSI device). In that case there would have to be a small hardware in which the specific address of the device as well as data may be stored. A small protocol would then resolve the IO.

- **Interrupt received when a different process is executing:** Suppose the process Pi initiated an IO. But subsequently it relinquishes the processor to process Pj . This may well happen because the process Pi may have finished its time slice or it may have reached a point when it must process a data expected from the device. Now let us suppose that priority of process Pi is higher than that of Pj. In that case the process Pj shall be suspended and Pi gets the processor, and the IO is accomplished. Else the process Pj shall continue and the IO waits till the process Pi is able to get the control of the processor. One other way would be to let the device interrupt the process Pj but store the IO information in a temporary buffer (which may be a register) and proceed. The process Pj continues. Eventually, the process Pi obtains the CPU. It would pick up the data from the temporary buffer storage. Clearly, the OS must provide for such buffer management. We next examine nested interrupts i.e. the need to service an interrupt which may occur during an interrupt service.
- **An Interrupt during an interrupt service:** This is an interesting possibility. Often devices or processes may have priorities. A lower priority process or device cannot cause an interrupt

while a higher priority process is being serviced. If, however, the process seeking to interrupt is of higher priority then we need to service the interrupt.

In the case where we have the return address deposited in the interrupt service routine code area, this can be handled exactly as the nested subroutine calls are handled. The most nested call is processed first and returns to the address stored in its code area. This shall always transfer the control to next outer layer of call. In the case we have a fixed number of interrupt levels, the OS may even use a stack to store the return addresses and other relevant information which is needed.

- **Interrupt vector:** Many systems support an interrupt vector (IV). As depicted in Figure 5.5, the basic idea revolves around an array of pointers to various interrupt service routines. Let us consider an example with four sources of interrupt. These may be a trap, a system call, an IO, or an interrupt initiated by a program. Now we may associate an index value 0 with trap, 1 with system call, 2 with IO device and 3 with the program interrupt. Note that the source of interrupt provides us the index in the vector. The interrupt service can now be provided as follows:

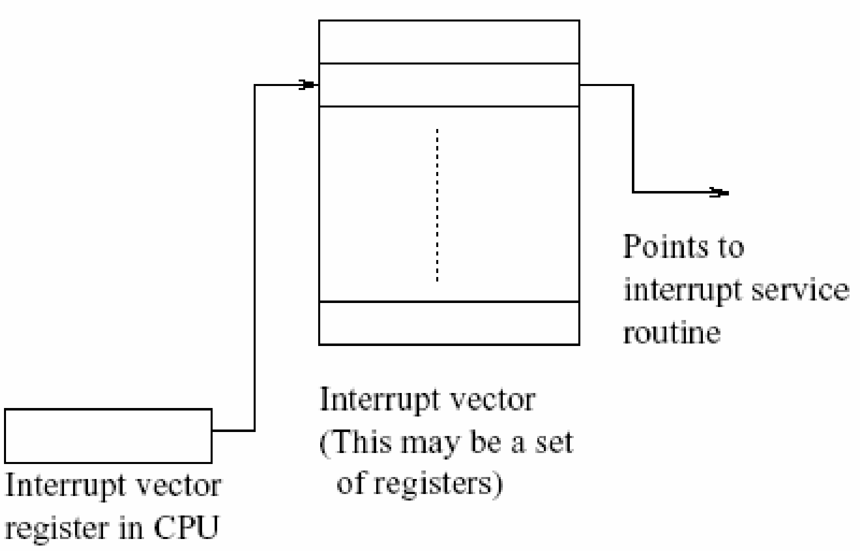


Figure 5.5: Interrupt vectors.

- Identify the source of interrupt and generate index i.
- Identify the interrupt service routine address by looking up IVR(i), where IVR stands for the interrupt vector register. Let this address be ISRi.
- Transfer control to the interrupt service routine by setting the program counter to ISRi.

Note that the interrupt vector may also be utilized in the context of a priority-based interrupt in which the bit set in a bit vector determines the interrupt service routine to be selected. It is very easy to implement this in hardware.

1.3.5 DMA Mode of Data Transfer

This is a mode of data transfer in which IO is performed in large data blocks. For instance, the disks communicate in data blocks of sizes like 512 bytes or 1024 bytes. The direct memory access, or DMA ensures access to main memory without processor intervention or support. Such independence from processor makes this mode of transfer extremely efficient.

When a process initiates a direct memory access (DMA) transfer, its execution is briefly suspended (using an interrupt) to set up the DMA control. The DMA control requires the information on starting address in main memory and size of data for transfer. This information is stored in DMA controller. Following the DMA set up, the program resumes from the point of suspension. The device communicates with main memory stealing memory access cycles in competition with other devices and processor. Figure 5.6 shows the hardware support.

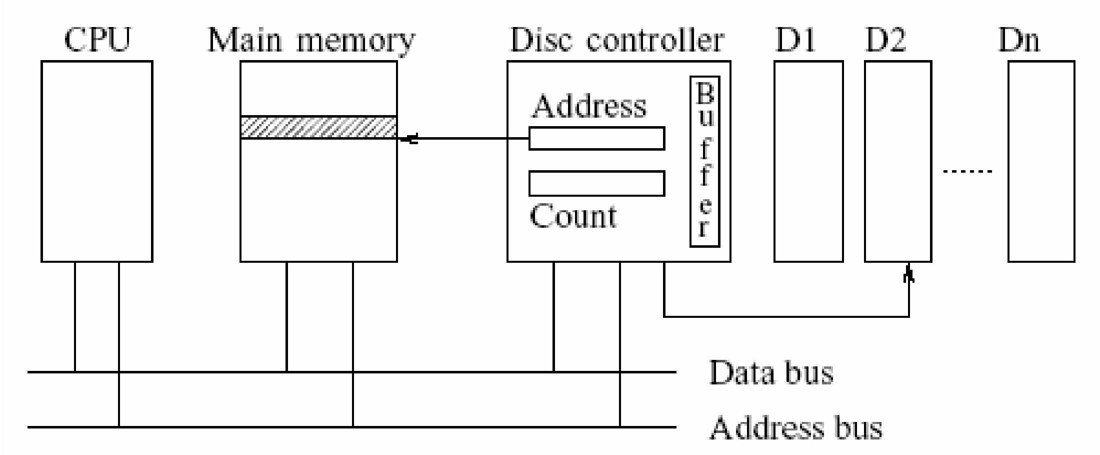
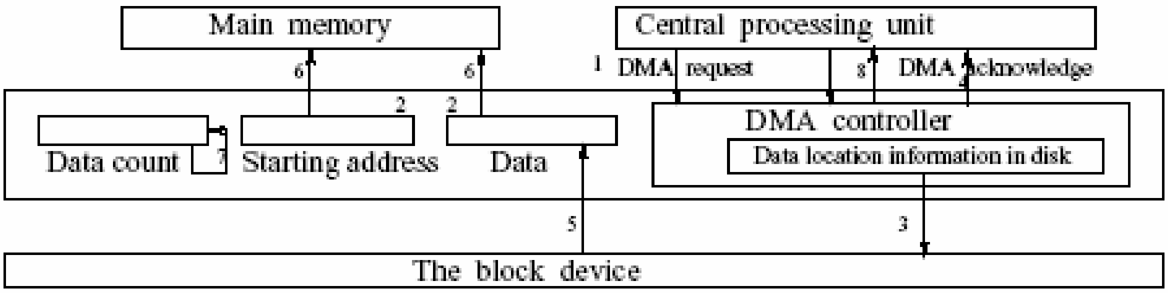


Figure 5.6: DMA : Hardware support.

Let us briefly describe the operations shown in Figure 5.6. Also, we shall assume a case of disk to main memory transfer in DMA mode. We first note that there is a disk controller to regulate communication from one or more disk drives. This controller essentially isolates individual devices from direct communication with the CPU or main memory. The communication is regulated to first happen between the device and the controller, and later between the controller and main memory or CPU if so needed. Note that these devices communicate in blocks of bits or bytes as a data stream. Clearly, an unbuffered communication is infeasible via the data bus. The bus has its own timing control protocol. The bus cannot, and should not, be tied to device transfer bursts. The byte stream block needs to be stored in a buffer isolated from the communication to processor or main memory. This is precisely what the buffer in the disk controller accomplishes. Once the controller buffer has the required data, then one can envisage to put the controller in contention with CPU and main memory or CPU to obtain an access to the bus. Thus if the controller can get the bus then by using the address and data bus it can directly communicate with main memory. This transfer shall be completely independent of program control from the processor. So we can effect a transfer of one block of data from the controller to main memory provided the controller has the address where data needs to be transferred and data count of the transfer required. This is the kind of information which initially needs to be set up in the controller address and count registers. Putting this information may be done under a program control as a part of DMA set up. The program that does it is usually the device controller. The device controller can then schedule the operations with much finer control. Data location information in disk



The DMA controller has request, acknowledge, interrupt and read / write lines of control

Figure 5.7: DMA : Direct memory access mode of data transfer.

Let us now recap the above mode of transfer within a logical framework with a step-by step description. This procedure can be understood in the context of Figure 5.7. In the figure a few lines do not have an accompanying label. However, by following the numbered sequence and its corresponding label, one can find the nature of the operations considered.

The procedure can be understood by following the description given below:

We shall assume that we have a program P which needs to communicate with a device D and get a chunk of data D to be finally stored starting in address A. The reader should follow through the steps by correlating the numbers in Figure 5.7.

1. The program makes a DMA set-up request.
2. The program deposits the address value A and the data count D.the program also indicates the virtual memory address of the data on disk.
3. The DMA controller records the receipt of relevant information and acknowledges the DMA complete.
4. The device communicates the data to the controller buffer.
5. The controller grabs the address bus and data bus to store the data, one word at a time.
6. The data count is decremented.
7. The above cycle is repeated till the desired data transfer is accomplished. At which time a DMA data transfer complete signal is sent to the process.

The network-oriented traffic (between machines) may be handled in DMA mode. This is so because the network cards are often DMA enabled. Besides, the network traffic usually corresponds to getting information from a disk file at both the ends. Also, because network traffic is in bursts, i.e. there are short intervals of large data transfers. DMA is the most preferred mode of communication to support network traffic.

1.3.6 A Few Additional Remarks

In every one of the above modes of device communication, it must be remarked that the OS makes it look as if we are doing a read or a write operation on a file. In the next section we explore how this illusion of a look and feel of a file is created to effect device communication. Also note that we may have programmed IO for synchronizing information between processes or when speed is not critical. For instance, a process may be waiting for some critical input information required to advance the computation further. As an example of programmed IO, we may consider the PC architecture based on i386 CPU which has a notion of listening to an IO port. Some architectures may even support polling a set of ports. The interrupt transfer is ideally suited for a small amount of critical information like a word, or a line i.e. no more than tens of bytes.

1.4 HW/SW Interface

IO management requires that a proper set-up is created by an application on computer system with an IO device. An IO operation is a combination of HW and SW instructions as shown in Figure 5.8. Following the issuance of an IO command, OS kernel resolves it, and then communicates with the concerned device driver. The device drivers in turn communicate with IO devices.

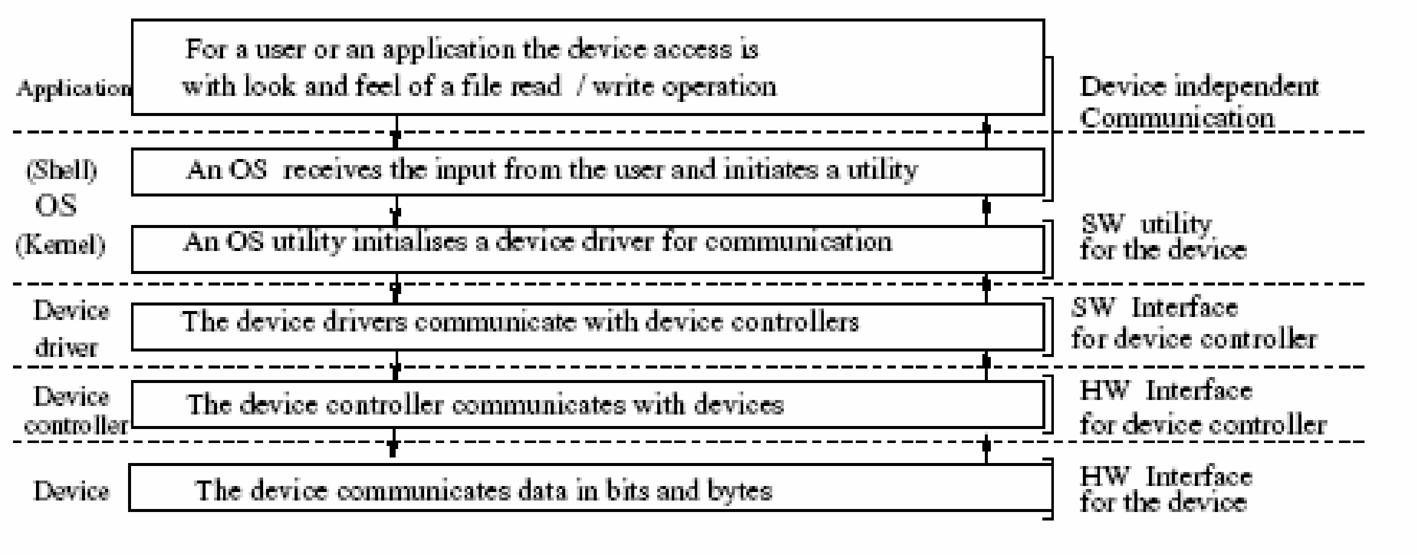

Application
For a user or an application the device access is with look and feel of a file read / write operation
Device independent Communication
(Shell)
OS
(Kernel)
An OS receives the input from the user and initiates a utility
An OS utility initialises a device driver for communication
SW utility for the device
Device driver
The device drivers communicate with device controllers
SW Interface for device controller
Device controller
The device controller communicates with devices
HW Interface for device controller
Device
The device communicates data in bits and bytes
HW Interface for the device

Figure 5.8: Communication with IO devices.

The application at the top level only communicates with the kernel. Each IO request from an application results in generating the following:

- Naming or identification of the device to communicate.
- Providing device independent data to communicate;

The kernel IO subsystem arranges for the following:

- The identification of the relevant device driver. We discuss device drivers in Section 1.4.1
- Allocation of buffers. We discuss buffer management in Section 1.5
- Reporting of errors.
- Tracking the device usage (is the device free, who is the current user, etc.) The device driver transfers a kernel IO request to set up the device controller. A device controller typically requires the following information:
- Nature of request: read/write.

- ➢ Set data and control registers for data transfer (initial data count = 0; where to look for data, how much data to transfer, etc.)
- ➢ Keep track when the data has been transferred (when fresh data is to be brought in). This may require setting flags.

1.4.1 Device Drivers

A device driver is specialized software. It is specifically written to manage communication with an identified class of devices. For instance, a device driver is specially written for a printer with known characteristics. Different make of devices may differ in some respect, and therefore, shall require different drivers. More specifically, the devices of different makes may differ in speed, the sizes of buffer and the interface characteristics, etc. Nevertheless device drivers present a uniform interface to the OS. This is so even while managing to communicate with the devices which have different characteristics.

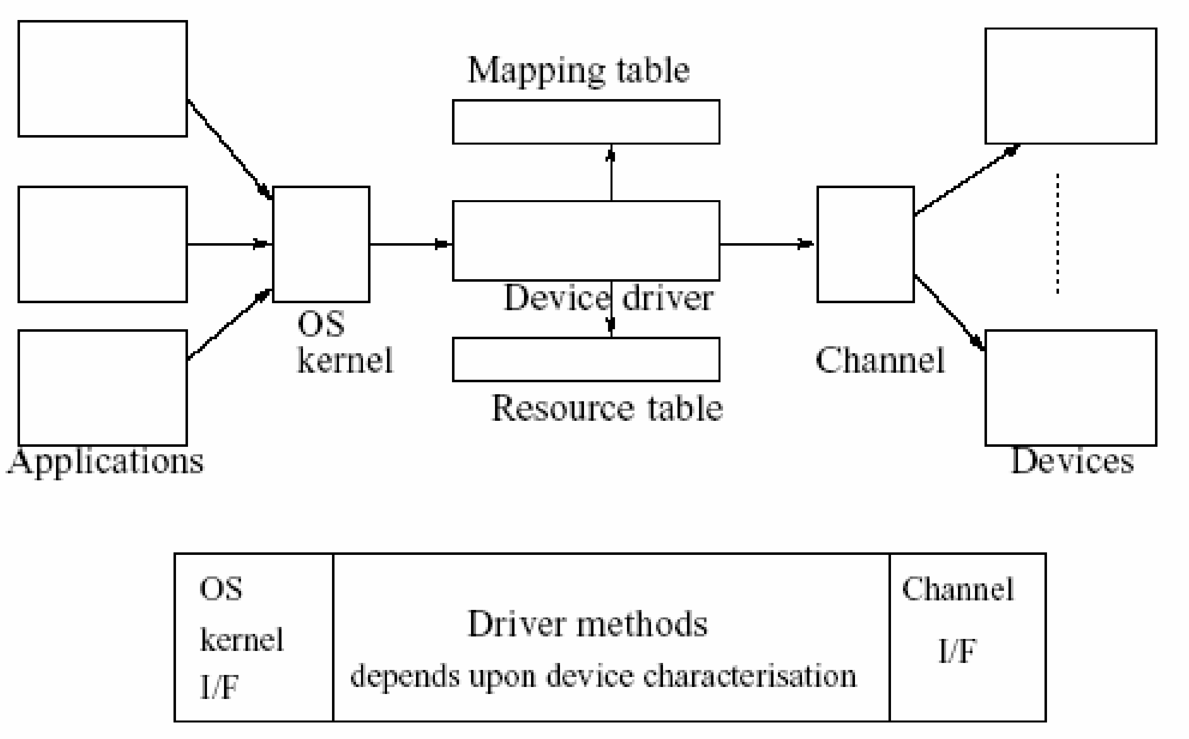


Figure 5.9: Device-driver interface.

In a general scenario, as shown in Figure 5.9, n applications may communicate with m devices using a common device driver. In that case the device driver employs a mapping table to achieve communication with a specific device. Drivers may also need to use specific resources (like a shared bus). If more than one resource is required, a device driver may also use a resource table. Sometimes a device driver may need to block a certain resource for its exclusive use by using a semaphore 1.

The device driver methods usually have device specific functionalities supported through standard function calls. Typically, the following function calls are supported. *open(), close(), lseek(), read(), write ()*

These calls may even be transcribed as *hd-open(), hd-close(),* etc. to reflect their use in the context of hard-disk operations. Clearly, each driver has a code specific to the device (or device controller). Semantically, the user views device communication as if it were a communication with a file. Therefore, he may choose to transfer an arbitrary amount of data. The device driver, on the other hand, has to be device specific. It cannot choose an arbitrary sized data transfer. The driver must manage fixed sizes of data for each data transfer. Also, as we shall see during the discussion on buffer transfer in Section 1.5 , it is an art to decide on the buffer size. Apparently, with n applications communicating with m devices the device driver methods assume greater levels of complexity in buffer management.

Sometimes the device drives are written to emulate a device on different hardware. For instance, one may emulate a RAM-disk or a fax-modem. In these cases, the hard-ware (on which the device is emulated) is made to appear like the device being emulated. The call to seek service from a device driver is usually a system call as device driver methods are often OS resident. In some cases where a computer system is employed to handle IO exclusively, the device drivers may be resident in the IO processor. In those cases, the communication to the IO processor is done in kernel mode. As a good design practice device drivers may be used to establish any form of communication, be it interrupt or DMA. The next section examines use of a device driver support for interrupt-based input.

1.4.2 Handling Interrupt Using Device Drivers

Let us assume we have a user process which seeks to communicate with an input device using a device driver process. Processes communicate by signaling. The steps in figure 5.10 describe the complete operational sequence (with corresponding numbers).

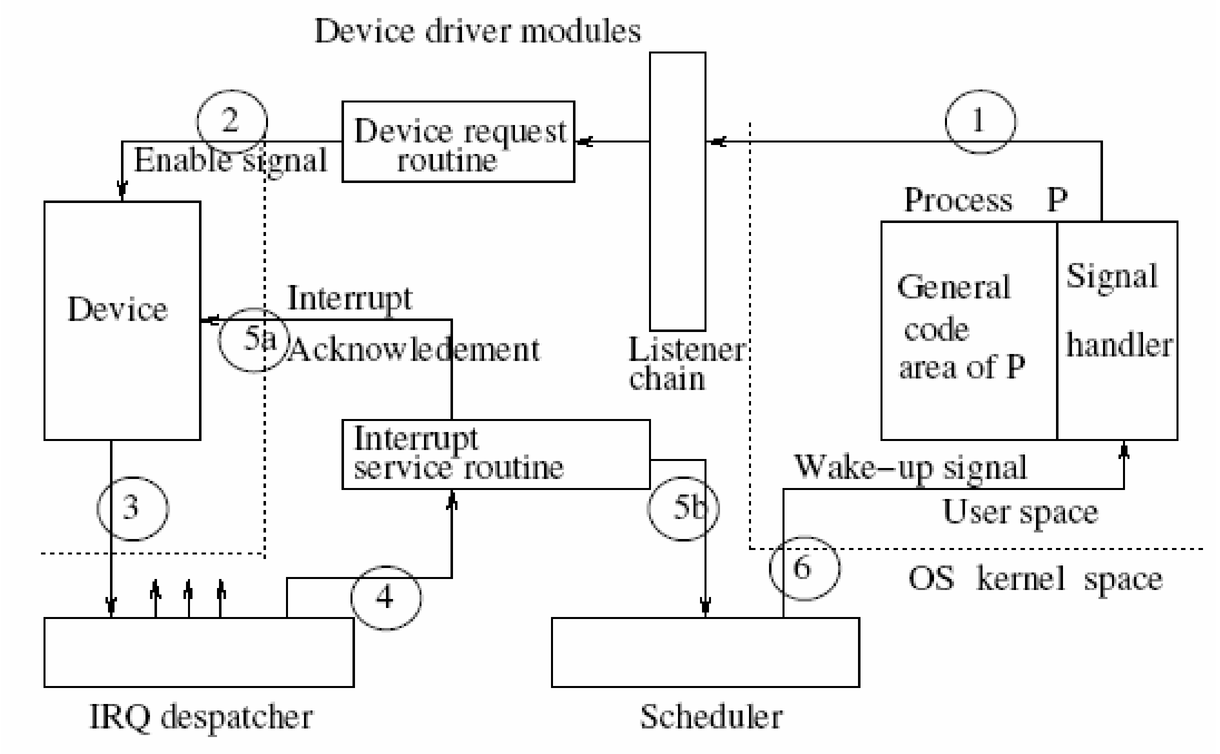


Figure 5.10: Device-driver operation.

1. Register with listener chain of the driver: The user process P signals the device driver as process DD to register its IO request. Process DD maintains a list data structure, basically a listener chain, in which it registers requests received from processes which seek to communicate with the input device.
2. Enable the device: The process DD sends a device enable signal to the device.
3. Interrupt request handling: After a brief while the device is ready to communicate and sends an interrupt request IRQ to process DD. In fact, the interrupt dispatcher in DD ensures that interrupt service is initiated.
4. Interrupt acknowledge and interrupt servicing: The interrupt service routine ISR acknowledges to the device and initiates an interrupt service routine 2.
5. Schedule to complete the communication: Process DD now schedules the data transfer and follows it up with a wake-up signal to P. The process receives the data to complete the input.
6. Generate a wake up signal.

Just as we illustrated an example of use of a device driver to handle an input device, we could think of other devices as well. One of the more challenging tasks is to write device driver for a pool of printers. It is not uncommon to pool print services. A printer requires that jobs fired at the pool are duly scheduled. There may be a dynamic assignment based on the load or there may even be a specific request (color printing on glazed paper for instance) for these printers.

In supporting device drivers for DMA one of the challenges is to manage buffers. In particular, the selection of buffer size is very important. It can very adversely affect the throughput of a system. In the next section we shall study how buffers may be managed.

1.5 Management of Buffers

A key concern, and a major programming issue from the OS point of view, is the management of buffers. The buffers are usually set up in the main memory. Device drivers and the kernel both may access device buffers. Basically, the buffers absorb mismatch in the data transfer rates of processor or memory on one side and device on the other. One key issue in buffer management is buffer-size. How buffer-sizes may be determined can be explained by using a simple analogy. The analogy we use relates to production and distribution for a consumable product like coffee. The scenario, depicted

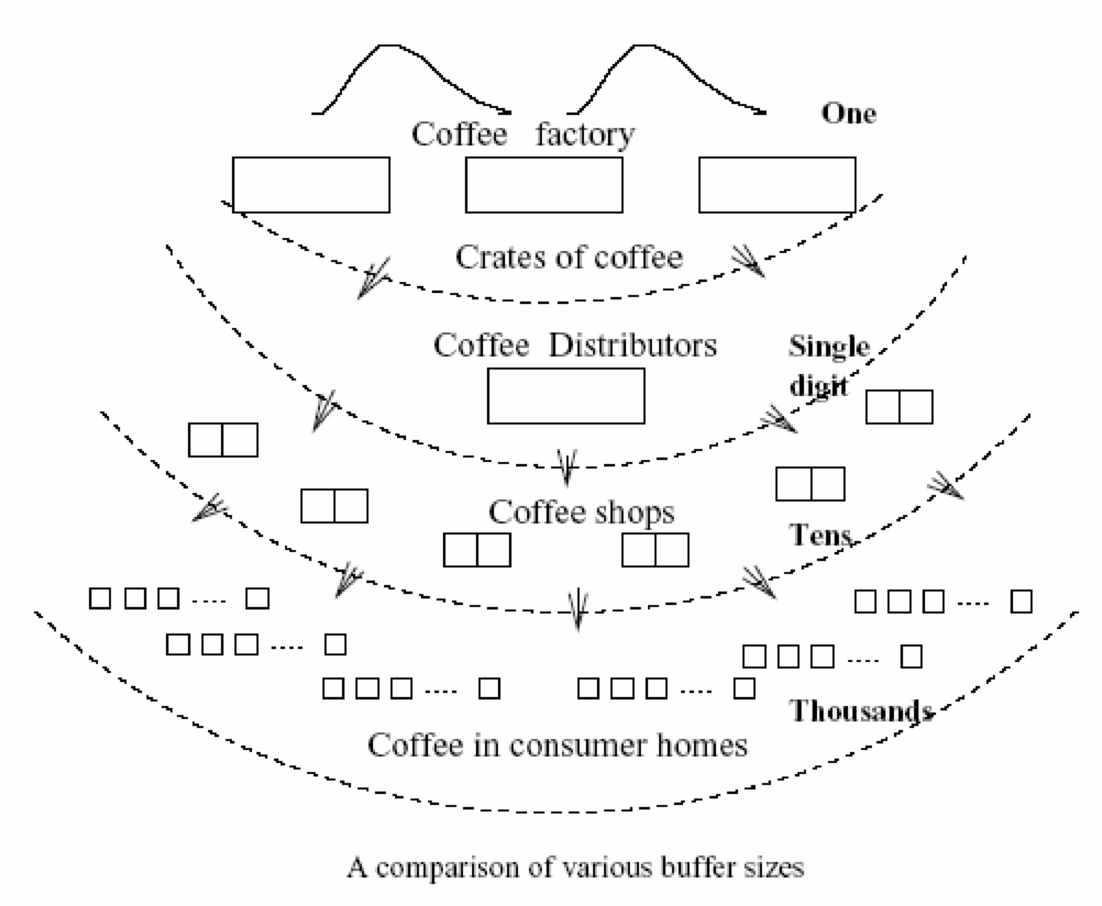


Figure 5.11: Coffee buffers

in Figure 5.11 shows buffer sizes determined by the number of consumers and the rate of consumption. Let us go over this scenario. It is easy to notice that a coffee factory would produce mounds of coffee. However, this is required to be packaged in crates for the distribution. Each crate may hold several boxes or bottles. The distributors collect the crates of boxes in tens or even hundreds for distribution to shops. Now that is buffer management. A super-market may get tens of such boxes while smaller shops like popand- mom stores may buy one or possibly two boxes. That is their buffer capacity based on consumption by their clients. The final consumer buys one tin (or a bottle). He actually consumes only a spoonful at one time. We should now look at the numbers and volumes involved. There may be one factory, supplying a single digit of distributors who distribute it to tens of super-markets and / or hundreds of smaller stores. The ultimate consumers number thousands with a very small rate of consumption. Now note that the buffer sizes for factory should meet the demands from a few bulk distributors. The buffer sizes of distributors meet the demand from tens of super-markets and hundreds of smaller shops. The smaller shops need the supplies of coffee bottles at most in tens of bottles (which may be a small part of the capacity of a crate). Finally, the end consumer has a buffer size of only one bottle. The moral of the story is that at each interface the producers and consumers of commodity balance the demand made on each other by suitably choosing a buffer size. The effort is to minimize the cost of lost business by being out of stock or orders. We can carry this analogy forward to computer operation. Ideally, the buffer sizes should be chosen in computer systems to allow for free flow of data, with neither the producer (process) of data nor the consumer (process) of data is required to wait on the other to make the data available.

Next we shall look at various buffering strategies (see Figure 5.12).

Single buffer: The device first fills out a buffer. Next the device driver hands in its control to the kernel to input the data in the buffer. Once the buffer has been used up, the device fills it up again for input.

Double buffer: In this case there are two buffers. The device fills up one of the two buffers, say buffer-0. The device driver hands in buffer-0 to the kernel to be emptied and the device starts filling up buffer-1 while kernel is using up buffer-0. The roles are switched when the buffer-1 is filled up.

Circular buffer: One can say that the double buffer is a circular queue of size two. We can extend this notion to have several buffers in the circular queue. These buffers are filled up in sequence. The kernel accesses the filled up buffers in the same sequence as these are filled up. The buffers are organized as a circular queue data structure, i.e. in case of output, the buffer is filled up from the CPU(or memory) end and used up by the output device, i.e. buffer n = buffer 0.

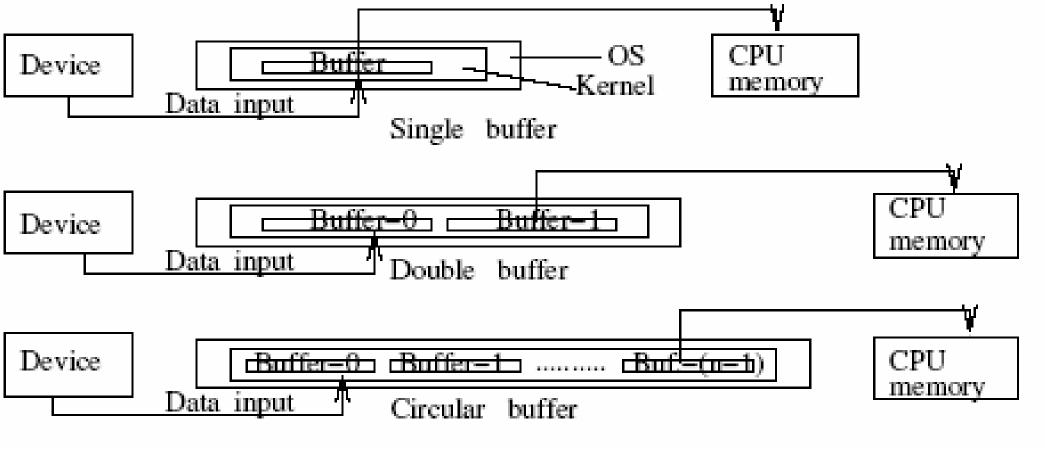


Figure 5.12: Buffering schemes.

Note that buffer management essentially requires managing a queue data structure. The most general of these is the circular buffer. One has to manage the pointers for the queue head and queue tail to determine if the buffer is full or empty. When not full or not empty the queue data structure can get a data item from a producer or put a data item into the consumer. This is achieved by carefully monitoring the head and tail pointers. A double buffer is a queue of length two and a single buffer is a queue of length one. Before moving on, we would also like to remark that buffer status of full or empty may be communicated amongst the processes as an event as indicated in Section 1.2.1 earlier.

1.6 Some Additional Points

In this section we discuss a few critical services like clocks and spooling. We also discuss many additional points relevant to IO management like caches.

Spooling: Suppose we have a printer connected to a machine. Many users may seek to use the printer. To avoid print clashes, it is important to be able to queue up all the print requests. This is achieved by spooling. The OS maintains all print requests and schedules each users' print requests. In other words, all output commands to print are intercepted by the OS kernel. An area is used to spool the output so that a users' job does not have to wait for the printer to be available. One can examine a print queue status by using *lpq* and *lpstat* commands in Unix.

Clocks: The CPU has a system clock. The OS uses this clock to provide a variety of system- and application-based services. For instance, the print-out should display the date and time of printing. Below we list some of the common clock-based services.

- ➢ Maintaining time of day. (Look up date command under Unix.)
- ➢ Scheduling a program run at a specified time during systems' operation. (Look up *at* and *cron* commands under Unix.)
- ➢ Preventing overruns by processes in preemptive scheduling. Note that this is important for real-time systems. In RTOS one follows a scheduling policy like the earliest deadline first. This policy may necessitate preemption of a running process.
- ➢ Keeping track of resource utilization or reserving resource use.
- ➢ Performance related measurements (like timing IO, CPU activity).

Addressing a device: Most OSs reserve some addresses for use as exclusive addresses for devices. A system may have several DMA controllers, interrupt handling cards (for some process control), timers, serial ports (for terminals) or terminal concentrators, parallel ports (for printers), graphics controllers, or floppy and CD ROM drives, etc. A fixed range of addresses allocated to each of these devices. This ensures that the device drives communicate with the right ports for data.

Caching: A cache is an intermediate level fast storage. Often caches can be regarded as fast buffers. These buffers may be used for communication between disk and memory or memory and CPU. The CPU memory caches may used for instructions or data. In case cache is used for instructions, then a group of instructions may be pre-fetched and kept there. This helps in overcoming the latency experienced in instruction fetch. In the same manner, when it is used for data it helps to attain a higher locality of reference.

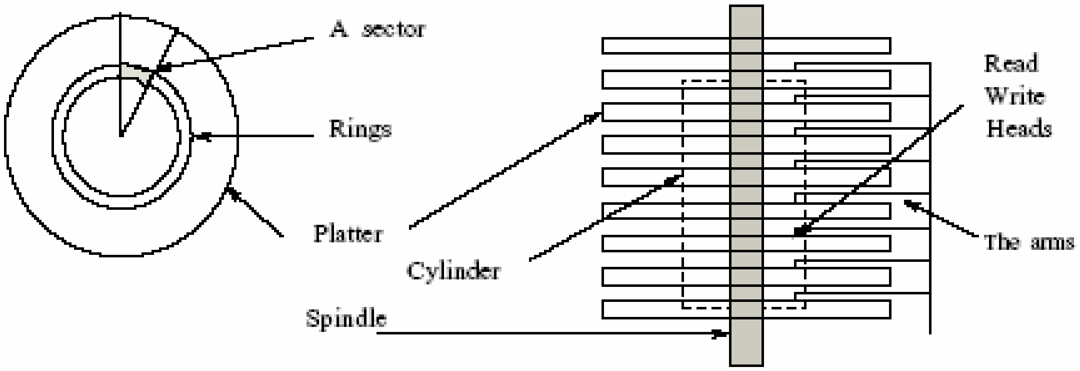
As for the main memory to disk caches, one use is in disk rewrites. The technique is used almost always to collect all the write requests for a few seconds before actually a disk is written into. Caching is always used to enhance the performance of systems.

IO channels: An IO channel is primarily a small computer to basically handle IO from multiple sources. It ensures that IO traffic is smoothed out.

OS and CDE: The common desk top environment (CDE) is the norm now days. An OS provides some terminal-oriented facilities for operations in a CDE. In particular the graphics user interface (GUI) within windows is now a standard facility. The kernel IO system recognizes all cursor and mouse events within a window to allow a user to bring windows up, iconize, scroll, reverse video, or even change font and control display. The IO kernel provides all the screen management functions within the framework of a CDE.

1.7 Motivation For Disk Scheduling

Primary memory is volatile whereas secondary memory is non-volatile. When power is switched off the primary memory loses the stored information whereas the secondary memories retain the stored information. The most common secondary storage is a disk. In Figure 5.13 we describe the mechanism of a disk and information storage on it. A disk has several platters. Each platter has several rings or tracks. The rings are divided into sectors where information is actually stored. The rings with similar position on different platters are said to form a cylinder. As the disk spins around a spindle, the heads transfer the information from the sectors along the rings. Note that information can be read from the cylinder surface without any additional lateral head movement. So it is always a good idea to organize all sequentially-related information along a cylinder. This is done by first putting it along a ring and then carrying on with it across to a different platter on the cylinder. This ensures that the information is stored on a ring above or below this ring.



Note that the rings on the disk platters on the spindle form a cylinder. Since all heads are on a particular ring at the same time, so it is easy to organise information on a cylinder. The information is stored in the sectors that can be identified on the rings. sectors are seperated from each other. All sectors can hold equal amount of information.

Figure 5.13: Information storage organisation on disks.

Information on different cylinders can be read by moving the arm by relocating the head laterally. This requires an additional arm movement resulting in some delay, often referred to as seek latency in response. Clearly, this delay is due to the mechanical structure of disk drives. In other words, there are two kinds of mechanical delays involved in data transfer from disks. The seek latency, as explained earlier, is due to the time required to move the arm to position the head along a ring. The other delay, called rotational latency, refers to the time spent in waiting for a sector in rotation to come under the read or write head. The seek delay can be considerably reduced by having a head per track disk. The motivation for disk scheduling comes from the need to keep both the delays to a minimum. Usually a sector which stores a block of information, additionally has a lot of other information. In Figure 5.14 we see that a 512 byte block has nearly

100 bytes of additional information which is utilized to synchronize and also check correctness of the information transfer as it takes place. Note that in figure 5.14 we have two pre-ambles each of 25 bytes, two synchronizing bytes, 6 bytes for checking errors in data transfer and a post-amble.

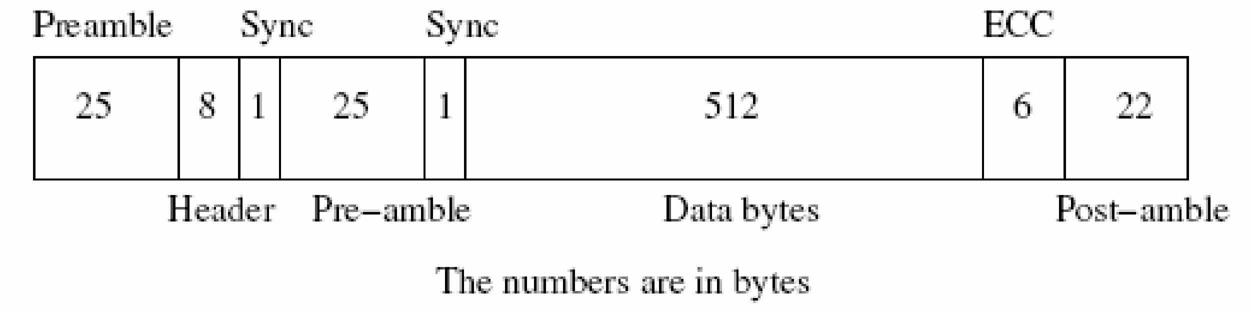


Figure 5.14: Information storage in sectors.

Scheduling Disk Operations: A user as well as a system spends a lot of time of operation communicating with files (programs, data, system utilities, etc.) stored on disks. All such communications have the following components:

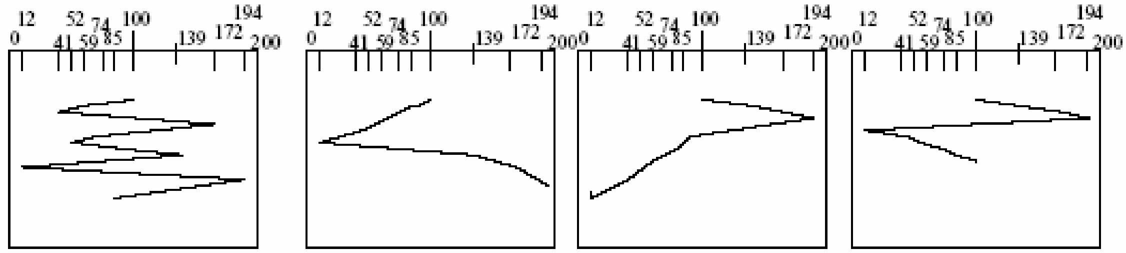
1. The IO is to read from, or write into, a disk.
2. The starting address for communication in main memory
3. The amount of information to be communicated (in number of bytes / words)
4. The starting address in the disk and the current status of the transfer.

The disk IO is always in terms of blocks of data. So even if one word or byte is required we must bring in (or write in) a block of information from (to) the disk. Suppose we have only one process in a system with only one request to access data. In that case, a disk access request leads finally to the cylinder having that data. However, because processor and memory are much faster than disks, it is quite possible that there may be another request made for disk IO while the present request is being serviced. This request would queue up at the disk. With multi-programming, there will be many user jobs seeking disk access. These requests may be very frequent. In addition, the information for different users may be on completely different cylinders. When we have multiple requests pending on a disk, accessing the information in a certain order becomes very crucial. Some policies on ordering the requests may raise the throughput of the disk, and therefore, that of the system.

Let us consider an example in which we have some pending requests. We only need to identify the cylinders for these requests. Suppose, there are 200 tracks or rings on each platter. We may have pending requests that may have come in the order 59, 41, 172, 74, 52, 85, 139, 12, 194, and 87.

The FCFS policy: The first come first served policy entails that the service be provided strictly in the sequence in which the requests arrived. If we do that then we service in the sequence 59, 41, 172, 74, 52, 85, 139, 12, 194, and 87. It is a good practice to analyze the effect of implementing a certain policy. In this case we try to analyze it by mapping the arm movements. The arm movement captures the basic disk activity. In the next Section we consider other policies as well. We also compare the arm movement required for FCFS policy with those required for the other policies.

1.8 Disk Scheduling Policies



First come first served   Shortest seek first   Elevator algorithm   Circular scan algorithm

In FCFS policy the information seek takes place in the sequence of cylinder numbers which are in the request queue. In figure 5.15 we plot map for FCFS access. In all the examples we assume that the arm is located at the cylinder number 100. The arm seek fluctuates sometimes very wildly. For instance, in our example there are wild swings from 12 to 194 and 41 to 172.

Shortest seek first: We look at the queue and compute the nearest cylinder location from the current location. Following this argument, we shall access in the order 87, 85, 4, 59, 52, 41, 12, 139, 172, and finally 194.

Elevator algorithm: Let us assume an initial arm movement in a certain direction. All the requests in that direction are serviced first. We have assumed that at 100 the arm as moving in the direction of increasing cylinder numbers. In that case it shall follow the sequence 139, 172, 194 and then descend to 87, 85, 74, 59, 41, and 12.

Circular scan: In the C-scan policy service is provided in one direction and then wraps round. In our example if the requests are serviced as the cylinder numbers increase then he sequence we follow would be 139, 172, and 174 and then wrap around to 12, 41, 52, 59, 74, 85, and finally 87.

From the response characteristics we can sense that FCFS is not a very good policy. In contrast, the shortest seek first and the elevator algorithm seems to perform well as these have the least arm movements. The circular scan too could be a very good scheduling mechanism, if the fly-back time for the disk arm is very short. In this chapter we have explored IO mechanisms. The IO devices are also resources.

Besides the physical management of these resources, OS needs to have a strategy for logical management of the resources as well. In the next chapter we shall discuss resource management strategies.

Unit-2

1.1 Introduction to Resource sharing and Management

The main motivation for scheduling various OS services is to maximize the usage of CPU resource, memory, and other IO resources. Consider the usage of a printer as an output resource. A user takes printouts only once in a while. A printer usage, therefore, can be shared amongst many users. The motivation to share a resource may come from several reasons. Sharing enhances utilization of resources immensely.

Sharing a resource is imperative in cases where we have a very expensive and specialized resource. For instance, an image processing resource, connected to a computer system, is a special resource. Such a resource is used in short periods of time, i.e. it is sparingly used. Similarly, in the context of a large project, there may be a file or a data-base which is shared amongst many users. Such a shared file may need to be updated from several sources. The shared file is then a shared resource. In this case, the sequencing of updates may be very critical for preserving data integrity and consistency. It may affect temporal semantics of the shared data. This is particularly true in transaction processing systems. In this chapter we shall study how the resources may be scheduled for shared usage. In particular, we shall study two very important concepts relating to mutual exclusion and deadlocks.

1.2 Need for Scheduling

Resources may be categorized depending upon the nature of their use. To enforce temporal sharing of a common resource, the OS needs a policy to schedule its usage. The policy may depend upon the nature of resource, frequency of its use and the context of its usage. In the case of a printer, the OS can spool printout requests. Printing, additionally, requires that once a process is engaged in printing, it must have its exclusive usage till it finishes the current print job. If that is not the case then the printouts from the printer shall be garbled. Some specialized resources, like a flat-bed plotter, require an elaborate initial set-up. So once assigned to a process, its usage better not be pre-empted. A process that gets such a resource should be permitted to keep it till either the process terminates or releases the resource. This is also true of a transaction which updates a shared data record. The transaction should complete the record's update before another process is given the access to the record.

Processes may need more than one resource. It is quite possible that a process may not be able to progress till it gets all the resources it needs. Let us suppose that a process P_1 needs resources r_1 and r_2. Process P_2 needs resources r_2 and r_3. Process P_1 can proceed only when it has both r_1 and r_2. If process P_2 has been granted r_2 then process P_1 has to wait till process P_2 terminates or releases r_2. Clearly, the resource allocation policy of an OS can affect the overall throughput of a system.

1.3 Mutual Exclusion

The mutual exclusion is required in many situations in OS resource allocation. We shall portray one such situation in the context of management of a print request. The print process usually maintains a queue of print jobs. This is done by maintaining a queue of pointers to the files that need to be printed. Processes that need to print a file store the file address (a file pointer) into this queue. The printer spooler process picks up a file address from this queue to print files. The spooler queue is a shared data structure amongst processes that are seeking the printer services and the printer spooler process. The printer spooler stores and manages the queue as shown in Figure 6.1. Let us consider just two

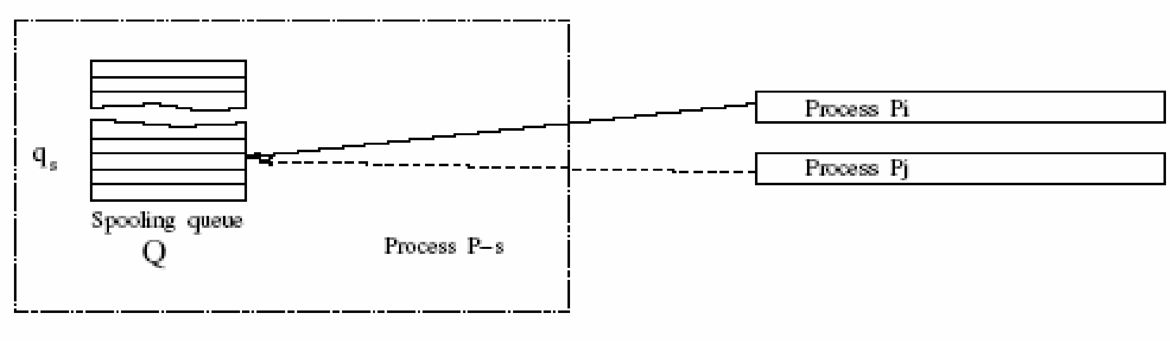


Figure 6.1: An example of mutual exclusion.

processes P_i, P_j that need to use printer. Let Ps denote the printer spooler process. The shared queue data area is denoted by Q. Let us now envision the situation as depicted below:

- Pi accesses Q and finds that a certain slot q_s is free.
- Pi decides to copy in this area the address of the file it needs to print.
- Next P_j accesses Q and also finds the slot q_s is free.
- P_j decides to copy in this area the address of file it needs to print.
- Pi copies the address of the area from which a file needs to be printed.
- Next P_j copies the address of the area from which a file needs to be printed.
- Ps reaches the slot at q_s and prints the file pointed to by the address in q_s.

On examining the above sequence we find that both the processes P_i, and P_j may record that their print jobs are spooled. As a matter of fact only the job from P_j was spooled. The print job of Pi never gets done. If we had mutual exclusion then either process Pi or process P_j , but not both, could have had an access to q_s . A point to be noted here is that Q is a shared data area (a resource) used by three processes P_i, P_j , and P_s. It also establishes an inter-process communication (IPC) between processes that need printing and the process which prints. Access to shared data resource that establishes inter-process communication must be mutually exclusive. We shall later revisit mutual exclusion in more detail in Section 1.6 . There we shall discuss how to ensure mutually exclusive access to resources. For now let us examine the conditions for deadlocks.

1.4 Deadlocks

We can understand the notion of a deadlock from the following simple real-life example. To be able to write a letter one needs a letter pad and a pen. Suppose there in one letter pad and one pen on a table with two persons seated around the table. We shall identify these two persons as Mr. A and Ms. B. Both Mr. A and Ms. B are desirous of writing a letter. So both try to acquire the resources they need. Suppose Mr. A was able to get the letter pad. In the meantime, Ms. B was able to grab the pen. Note that each of them has one of the two resources they need to proceed to write a letter. If they hold on to the resource they possess and await the release of the resource by the other, then neither of them can proceed. They are deadlocked. We can transcribe this example for processes seeking resources to proceed with their execution.

Consider an example in which process P_1 needs three resources r_1; r_2, and r_3 before it can make any further progress. Similarly, process P_2 needs two resources r_2 and r_3. Also, let us assume that these resources are such that once granted, the permission to use is not withdrawn till the processes release these resources. The processes proceed to acquire these resources. Suppose process P_1 gets resources r_1 and r_3 and process P_2 is able to get resource r_2 only. Now we have a situation in which process P_1 is waiting for process P_2 to release r_2 before it can proceed. Similarly, process P_2 is waiting for process P_1 to release resource r_3 before it can proceed. Clearly, this situation can be recognized as a deadlock condition as neither process P_1 nor process P_2 can make progress. Formally, a deadlock is a condition that may involve two or more processes in a state such that each is waiting for release of a resource which is currently held by some other process.

A graph model: In Figure 6.2 we use a directed graph model to capture the sense of deadlock. The figure uses the following conventions.

- There are two kinds of nodes - circles and squares. Circles denote processes and squares denote resources.
- A directed arc from a process node (a circle) to a resource node denotes that the process needs that resource to proceed with its execution.
- A directed arc from a square (a resource) to a circle denotes that the resource is held by that process.

With the conventions given above, when a process has all the resources it needs, it can execute. This condition corresponds to the following.

- The process node has no arcs directed out to a resource node.
- All the arcs incident into this process node are from resource nodes.

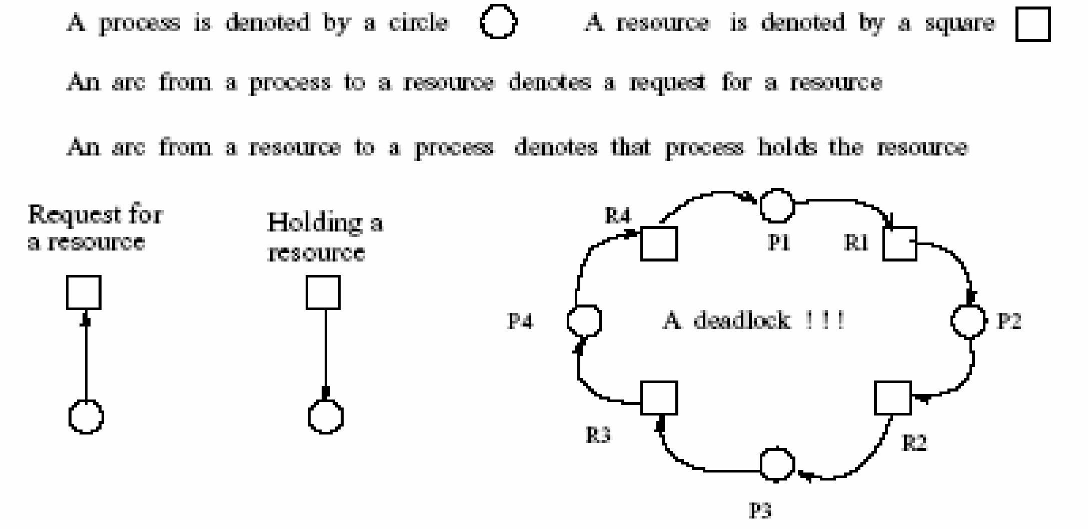

A process is denoted by a circle
A resource is denoted by a square
An arc from a process to a resource denotes a request for a resource
An arc from a resource to a process denotes that process holds the resource
Request for a resource
Holding a resource
R4
P1
R1
P4
A deadlock ! ! !
P2
R3
R2
P3

Figure 6.2: A directed graph model.

In Figure 6.2, P_1 holds r_4 but awaits release of r_1 to proceed with execution; P_2 holds r_1 but awaits release of r_2 to proceed with execution; P_3 holds r_2 but awaits release of r_3 to proceed with execution; P_4 holds r_3 but awaits release of r_4 to proceed with execution.

Clearly, all the four processes are deadlocked. Formally, a deadlock occurs when the following four conditions are present simultaneously.

- **Mutual exclusion**: Each resource can be assigned to at most one process only.
- **Hold and wait:** Processes hold a resource and may seek an additional resource.
- **No pre-emption:** Processes that have been given a resource cannot be preempted to release their resources.
- **Circular wait:** Every process awaits release of at least one resource held by some other processes.

Dead-lock Avoidance: A deadlock requires the above four conditions to occur at the same time, i.e. mutual exclusion, hold and wait, no pre-emption and circular wait to occur at the same time. An analysis and evaluation of the first three conditions reveals that these are necessary conditions. Also, we may note that the circular wait implies hold and wait. The question is how does one avoid having a deadlock? We shall next examine a few arguments. The first one favors having multiple copies of resources. The second one argues along preventive lines, i.e. do not permit conditions for deadlock from occurring. These arguments bring out the importance of pre-empting.

The infinite resource argument: One possibility is to have multiple resources of the same kind. In that case, when one copy is taken by some process, there is always another copy available. Sometimes we may be able to break a deadlock by having just a few additional copies of a resource. In Figure 6.3 we show that there are two copies of resource r_2. At the moment, processes P_1 and P_2 are deadlocked. When process P_3 terminates a copy of resource r_2 is released. Process P_2 can now have all the resources it needs and the deadlock is immediately broken. P_1 will get r_1 once P_2 terminates and releases the resources held.

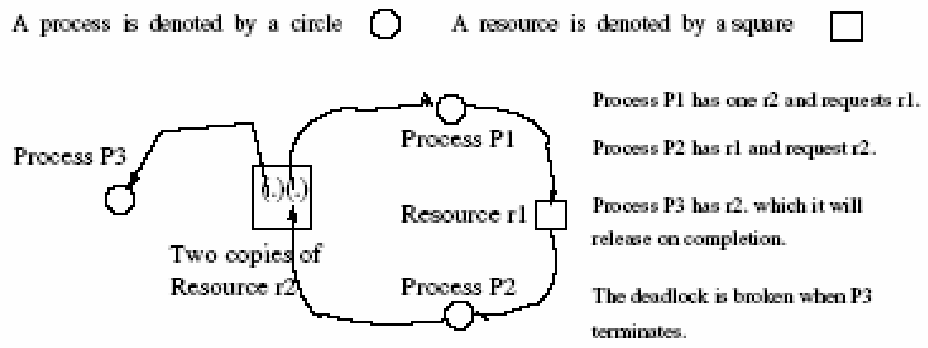

A process is denoted by a circle
A resource is denoted by a square
Process P3
Process P1
Resource r1
Two copies of Resource r2
Process P2
Process P1 has one r2 and requests r1.
Process P2 has r1 and request r2.
Process P3 has r2. which it will release on completion.
The deadlock is broken when P3 terminates.

Figure 6.3: Multiple resource availability.

The next pertinent question is: how many copies of each resource do we need? Unfortunately, theoretically, we need an infinite number of copies of each resource!! Note even in this example, if P_3 is deadlocked, then the deadlock between P_1 and P_2 cannot be broken. So, we would need one more copy of resource r_2. That clearly demonstrates the limitation of the multiple copies argument.

Never let the conditions occur: It takes some specific conditions to occur at the same time to cause deadlock. This deadlock avoidance simply states that do not let these conditions occur at the same time. Let us analyze this a bit deeper to determine if we can indeed prevent these conditions from occurring at the same time. The first condition is mutual exclusion. Unfortunately, many resources do require mutual exclusion!! So we must live with it and design our systems bearing in mind that mutual exclusion would have to be provided for. Next, let us consider the condition of hold and wait. Since, hold and wait is also implied by circular wait, we may look at the possibility of preventing any circular waits. This may be doable by analyzing program structures. Now let us examine pre-emption. It may not be the best policy to break a deadlock, but it works. Pre-emption is clearly enforceable in most, if not all, situations. Pre-emption results in releasing resources which can help some processes to progress, thereby breaking the deadlock. In fact, many real-time OSs require pre-emption for their operation. For example, when a certain critical condition arises, alarms must be set or raised. In some other cases an emergency process may even take over by pre-empting the currently running process.

1.4.1 A Deadlock Prevention Method

In a general case, we may have multiple copies of resources. Also, processes may request multiple copies of a resource. Modeling such a scenario as a graph is difficult. In such a situation, it is convenient to use a matrix model. We shall use Figure 6.4 to explain the matrix-based method. In Figure 6.4 we assume n processes and m kinds of resources. We denote the *i*th resource by ri. We now define two vectors, each of size m.

Vector R = (r_1; r_2; ::::; r_m) : r_i = resources of type i with the system. Vector A = (a_1; a_2; ::::; a_m) : a_i = resources of type i presently available for allocation. Initially with no allocations made, we have R = A. However, as allocations happen, vector A shall be depleted. Also, when processes terminate and release their resources, vector A gets updated to show additional resources that become available now. We also define two matrices to denote allocations made and requests for the resources. There is a row for each process and a column for each resource. Matrix AM and matrix RM respectively have entries for allocation and requests. An entry $c_{i,j}$ in matrix AM denotes the number of resources of type j currently allocated to process Pi. Similarly, $q_{i,j}$ in matrix RM denotes the number of resources of type j requested by process Pi. This is depicted in Figure 6.4. Below we state the three conditions which capture the constraints for the model. The first condition always holds. The second condition holds when requests on resources exceed capacity. In this condition not all processes can execute simultaneously.

1. $\left[\sum_{i=1}^{n} c_{i,j} + a_j\right] \le r_j$ .This condition states that the allocation of resource j to all the processes plus the now available resource of kind j is always less than the ones available with the system.
2. $\left[\sum_{i=1}^{n} q_{i,j}\right] \ge r_j$. This condition states that the requests for resources made by every process may exceed what is available on the system.
3. In addition, we have the physical constraint $\left[\forall j c_{i,j}\right] \le q_{i,j}$. This condition states that allocation of a resource j to a process may be usually less than the request made by the process. At the very best the process's request may be fully granted.

The matrix model captures the scenario where n processes compete to acquire one or more copies of the m kinds of resources.

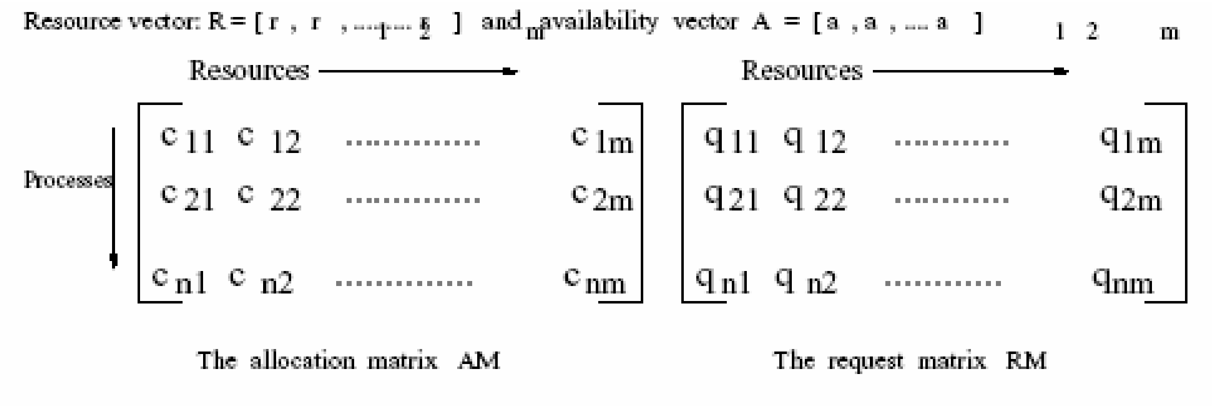

Resource vector: R = [r , r , ... r] and availability vector A = [a , a , ... a]
1 2 m
Resources
Resources
Processes
c 11 c 12 .... c 1m
c 21 c 22 .... c 2m
c n1 c n2 .... c nm
q 11 q 12 .... q 1m
q 21 q 22 .... q 2m
q n1 q n2 .... q nm
The allocation matrix AM
The request matrix RM

Figure 6.4: Matrix model of requests and allocation.

1.5 Deadlock Detection and Prevention Algorithms

In this section we shall discuss a few deadlock detection and prevention algorithms. We shall employ both the graph and matrix models. We begin with the simple case when we have one copy of each resource and have to make allocation to a set of processes. A graph based detection algorithm: In our digraph model with one resource of one kind, the detection of a deadlock requires that we detect a directed cycle in a processor resource digraph. This can be simply stated as follows.

- Choose a process node as a root node (to initiate a depth first traversal).
- Traverse the digraph in depth first mode.
- Mark process nodes as we traverse the graph.
- If a marked node is revisited then a deadlock exists.

In the above steps we are essentially trying to detect the presence of a directed cycle.

Bankers' Algorithm: This is a simple deadlock prevention algorithm. It is based on a banker's mind-set: “offer services at no risk to the bank". It employs a policy of resource denial if it suspects a risk of a deadlock. In other words, for a set of processes, resources are allocated only when it shall not lead to a deadlock. The algorithm checks for the four necessary conditions for deadlock. A deadlock may happen when any one of the four conditions occurs. If a deadlock is likely to happen with some allocation, then the algorithm simply does not make that allocation.

The manner of operation is as follows: The request of process i is assessed to determine whether the process request can be met from the available resources of each kind. This means $\forall j \left(q_{i,j}\right) \leq a_j$. In that case, process i may be chosen to execute. In fact, the policy always chooses that subset amongst the processes which can be scheduled to execute without a deadlock.

Let us now offer a critique of the algorithm.

1. If there are deadlocked processes, they shall remain deadlocked. Bankers' algorithm does not eliminate an existing deadlock.
2. Bankers' algorithm makes an unrealistic assumption. It stipulates that the resource requirements for processes are known in advance. This may not be rare but then there are processes which generate resource requests on the fly. These dynamically generated requirements may change during the lifetime of a process.
3. With multi-programming, the number of live processes at any one time may not be known in advance.
4. The algorithm does not stipulate any specific order in which the processes should be run. So in some situations, it may choose an order different from the desired order. Sometimes we do need processes to follow a specific order. This is true when the processes must communicate in a particular sequence.
5. Also, the algorithm assumes a fixed number of resources initially available on a system. This too may vary over time.

A matrix based deadlock detection method: When multiple resources of each kind are available, we use the matrix model shown in Figure 6.4 to detect deadlocks. We analyze the requests of the processes (matrix RM) against initially available copies of each resource (vector A). This is what the algorithm below indicates. Following the description of the algorithm there is a brief explanation of the same.

```
Procedure detect deadlock
begin
    $\forall i, 1 \le i \le n$ set marked(i) = false. These flags help us to detect marked processes whose requirements can be satisfied.
    deadlockpresent = false
    All entries in matrix AM are initialized to zero.
    While there are processes yet to be examined do
    {
        Pick a process Pi whose requests have not been examined yet.
        For process $P_i$ check if $RM_i \le A$ then
        {
            allocate the resources;
            marked(i) = true
            Add allocation made to row $AM_i$
            Subtract this allocation from A to update A
        }
    }
    If $\forall i$ , marked(i) is true then deadlockpresent = false else deadlockpresent = true
end
```

We offer an alternative explanation for the above algorithm; let us assume that processes P_1 through P_n are to be allocated m kinds of resources. We begin with some tentative allocation of resources starting with, say, P_1 in sequence. Now let us consider an intermediate step: the allocation for process during which we are determining allocation for process P_i. The row corresponding to process P_i in matrix RM denotes its resource requests. This row gives the number for each kind of resource requested by P_i. Recall that vector A denotes the resources presently available for allocation. Now, let us suppose that resource requests of process Pi can be met. In that case, vector $RM_i \leq A$. This means that this process could be scheduled to execute and no deadlock as yet has manifested. This allocation can then be reflected in matrix AM_i. Also, vector A needs to be modified accordingly. Next, with the revised vector A, we try to meet the requirements of the next process P_{i+1} which is yet to be allocated with its resources. If we can exhaustively run through the sequence then we do not have a deadlock. However, at some stage we may find that its requirement of resources exceeds the available resources. Suppose this happens during the time we attempt allocation for process P_{i+k}. In that case, we have a deadlock for the subset of processes P_1 through P_{i+k}. Recall that we are marking the processes that obtain their allocation. So if we have all the processes marked then there is no deadlock. If there is a set of processes that remain unmarked then we have a deadlock. Notwithstanding the non-deterministic nature of this algorithm it always detects a deadlock. Like the bankers' algorithm, this algorithm also does not help to eliminate an existing deadlock. Deadlock elimination may require pre-emption or release of resources. This may also result in a roll back in some transaction-oriented systems. This further reinforces pre-emption as an effective deadlock elimination strategy.

1.6 Mutual Exclusion Revisited: Critical Sections

We have earlier seen that for devices like printers, an OS must provide for mutual exclusion in operation. It is required for memory access as well. Whenever there is a shared area accessed by two or more processes, we have to ensure that only one process has write access at one time. The main motivation is to avoid a race condition amongst processes. When a race occurs between processes (acting independent of each other), each may annul others' operations (as we saw in our spooler example). Transaction oriented processing is notorious for the race conditions as it may lead to data integrity problems. These problems are best taken care of by ensuring the process has exclusive access to the data area (or the resource) where it has to perform the operation. So each process operates in exclusion of access to the others. Hence, the term mutual exclusion.

Next we define a critical section of a program code in this context. By a “critical section” we mean that section of code (in a process) which is executed exclusively, i.e. none of its operations can be annulled. To prevent shared variables from being overwritten by another process, a process must enter its critical section. Operating in critical sections ensures mutual exclusion of processes. Well, how does one ensure such an operation?

OSs, including Unix, provides a facility called semaphore to allow processes to make use of critical sections in exclusion to other processes. A semaphore is essentially a variable which is treated in a special way. Access to a semaphore and operations on it are permitted only when it is in a free state. If a process locks a semaphore, others cannot get access to it. However, every process must release the semaphore upon exiting its critical section. In other words, a process may enter a critical section by checking and manipulating a semaphore. When a process has entered its critical section, other processes are prevented from accessing this shared variable. When a process leaves the critical section, it changes the state of semaphore from locked to free. This permits anyone of the waiting processes to now enter their critical sections and use the shared variable. To make sure that the system actually works correctly, a notion of atomicity or indivisibility is invoked, i.e. semaphore operations are run to completion without interruptions as explained in the next section.

1.6.1 Basic Properties of Semaphores

Semaphores have the following properties.

- A semaphore takes only integer values. We, however, would limit to semaphores that take only binary values. In general, we may even have a data-structure in which every entry is a semaphore. Usually, such structures are required for establishing a set of processes that need to communicate. These are also required when a complex data structure like a record is shared.
- There are only two operations possible on a semaphore.
  - A *wait* operation on a semaphore decreases its value by one.
    wait(s): **while** *s < 0* **do** noop; s := s - 1;
  - A signal operation increments its value, i.e. signal(s): s : = s + 1;

A semaphore operation is atomic and indivisible. This essentially ensures both *wait* and *signal* operations are carried out without interruption. This may be done using some hardware support. The interrupt signal is recognized by the processor only after these steps have been carried out. Essentially, this means it is possible to use disable and enable signals to enforce indivisibility of operations. The wait and signal operations are carried out indivisibly in this sense.

Note that a process is blocked (busy waits) if its wait operation evaluates to a negative semaphore value. Also, a blocked process can be unblocked when some other process executes signal operation to increment semaphore to a zero or positive value. We next show some examples using semaphores.

1.6.2 Usage of Semaphore

Suppose we have two processes P1 and P2 that need mutual exclusion. We shall use a shared semaphore variable use with an initial value = 0. This variable is accessible from both P1 and P2. We may require that both these processes have a program structure that uses repeat – until pair as a perpetual loop. The program shall have the structure as shown below:

```
repeat
Some process code here
wait (use);
enter the critical section (the process manipulates a shared area);
signal (use);
the rest of the process code.
until false;
```

With the repeat{until sequence as defined above, we have an infinite loop for both the processes. On tracing the operations for P_1 and P_2 we notice that only one of these processes can be in its critical section. The following is a representative operational sequence. Initially, neither process is in critical section and, therefore, use is 0.

- Process P_1 arrives at the critical section first and calls *wait (use)*.
- It succeeds and enters the critical section setting *use* = - 1.
- Process P_2 wants to enter its critical section. Calls *wait* procedure.
- As *use* < 0. P_2 does a busy wait.
- Process P_1 executes *signal* and exits its critical section. *use* = 0 now.
- Process P_2 exits busy wait loop. It enters its critical section *use* = -1.

The above sequence continues.

Yet another good use of a semaphore is in synchronization amongst processes. A process typically may have a synchronizing event. Typically one process generates an event and the other process awaits the occurrence of that event to proceed further. Suppose we have our two processes, P_i and P_j . P_j can execute some statement S_j only after a statement S_i in process Pi has been executed. This synchronization can be achieved with a semaphore se initialized to -1 as follows:

- In Pi execute the sequence S_i; signal (se);
- In P_j execute wait (se); S_j ;

Now process P_j must wait completion of S_i before it can execute S_j . With this example, we have demonstrated use of semaphores for inter-process communication. We shall next discuss a few operational scenarios where we can use semaphores gainfully.

1.6.3 Some Additional Points

The primary use of semaphore which we have seen so far was to capture when a certain resource is “in use" or “free". Unix provides a wait instruction to invoke a period of indefinite wait till a certain resource may be in use. Also, when a process grabs a resource, the resource is considered to be “locked".

So far we have seen the use of a two valued or binary semaphore. Technically, one may have a multi-valued semaphore. Such a semaphore may have more than just two values to capture the sense of multiple copies of a type of resource. Also, we can define an array of semaphores i.e. each element of the array is a semaphore. In that case, the array can be used as a combination for several resources or critical operations. This is most useful in databases where we sometimes need to lock records, and even lock fields. This is particularly true of transaction processing systems like bank accounts, airlines ticket booking, and such other systems.

Since, semaphore usually has an integer value which is stored somewhere, it is information the system can use. Therefore, there are processes with permission to access, a time stamp of creation and other system-based attributes. Lastly, we shall give syntax for defining semaphore in Unix environment.

semaphoreId = semget(key_sem, no_sem, flag_sem)

Here *semget* is a system call, *key_sem* provides a key to access, *no_sem*= defines the number of semaphores required in the set. Finally, *flag_sem* is a standard access control defined by

IPC_CREAT | 644 to give a rw-r--r-- access control. In the next chapter we shall see the use of semaphore, as also, the code for other inter process communication mechanisms.

Unit-3

1.1 Introduction to Inter-Process communication

Processes execute to accomplish specified computations. An interesting and innovative way to use a computer system is to spread a given computation over several processes. The need for such communicating processes arises in parallel and distributed processing contexts. Often it is possible to partition a computational task into segments which can be distributed amongst several processes. Clearly, these processes would then form a set of communicating processes which cooperate in advancing a solution. In a highly distributed, multi-processor system, these processes may even be resident on different machines. In such a case the communication is supported over a network. We shall study some of the basics to be able to do the following:

- How to spawn (or create) a new process.
- How to assign a task for *exec*ution to this newly spawned process.
- A few mechanisms to enable communication amongst processes.
- Synchronization amongst these processes.

In most cases an IPC package is used to establish inter-process communication. Depending upon the nature of the chosen IPC, the package sets up a data structure in kernel space. These data structures are often persistent. So once the purpose of IPC has been fulfilled, this set-up needs to be deleted (a cleanup operation). The usage pattern of the IPC package in a system (like system V Unix) can be seen by using explicit commands like *ipcs*. A user can also remove any unused kernel resources by using a command like *ipcrm.*

For our discussions here, we shall assume Unix like environment. Hopefully, the discussion here offers enough information to partially satisfy and raise the level of curiosity about the distributed computing area.

1.2 Creating A New Process: The *fork()* System Call

One way to bring in a new process into an existing *exec*ution environment is to *exec*ute *fork()* system call. Just to recap how system calls are handled, the reader may refer to Figure 7.1. An application raises a system call using a library of call functions. A system call in turn invokes its service (from the kernel) which may result in memory allocation,

device communication or a process creation. The system call *fork()* spawns a new process which, in fact, is a copy of the parent process from where it was invoked!! The newly spawned process inherits its parent's *exec*ution environment. In Table 7.1 we list some of the attributes which the child process inherits from its parent.

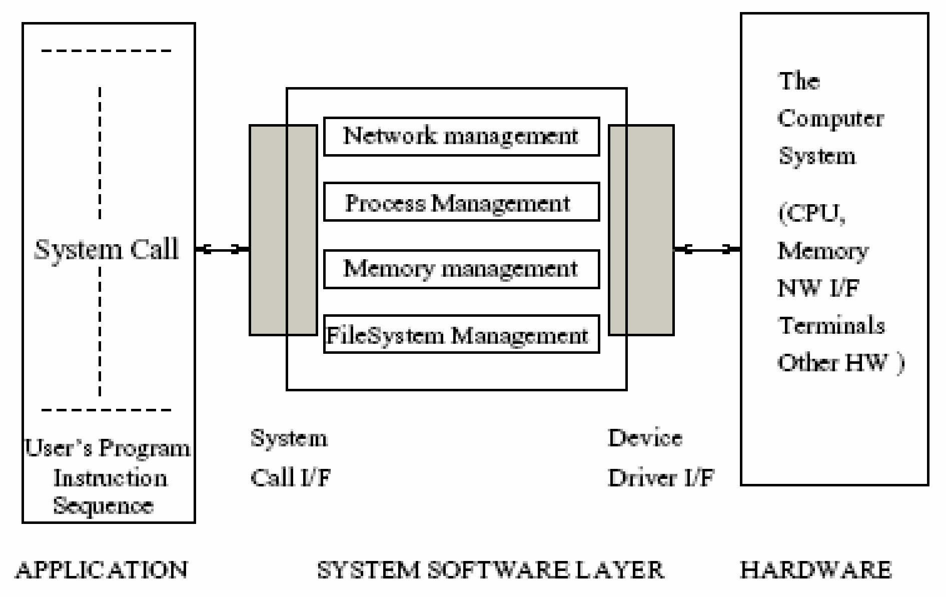
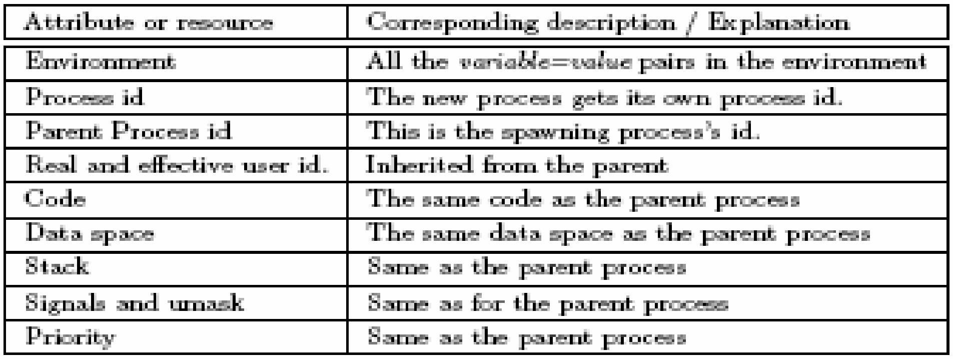


Figure 7.1: Processing system calls.

Note that a child process is a process in its own right. It competes with the parent process to get processor time for *exec*ution. In fact, this can be easily demonstrated (as we shall later see). The questions one may raise are:

- Can one identify when the processor is *exec*uting the parent and when it is *exec*uting the child process?
- What is the nature of communication between the child and parent processes?



Attribute or resource	Corresponding description / Explanation
Environment	All the *variable=value* pairs in the environment
Process id	The new process gets its own process id.
Parent Process id	This is the spawning process's id.
Real and effective user id.	Inherited from the parent
Code	The same code as the parent process
Data space	The same data space as the parent process
Stack	Same as the parent process
Signals and umask	Same as for the parent process
Priority	Same as the parent process

Table 7.1: Properties inherited by a newly forked process.

The answer to the first question is yes. It is possible to identify when the parent or child is in *exec*ution. The return value of *fork()* system call is used to determine this. Using the return value, one can segment out the codes for *exec*ution in parent and child. We will show that in an example later.

The most important communication from parent to child is the *exec*ution environment which includes data and code segments. Also, when the child process terminates, the parent process receives a signal. In fact, a signal of the termination of a child process, is one feature very often exploited by programmers. For instance, one may choose to keep parent process in wait mode till all of its own child processes have terminated. Signaling is a very powerful inter-process communication mechanism (using *signal*s) which we shall learn in Section 1.4.5. The following program demonstrates how a child process may be spawned.

The program: Demonstration of the use of *fork()* system call

```
main()
{ int i, j;
if ( fork() ) /* must be parent */
{ printf("\t\t In Parent \n");
printf("\t\t pid = %d and ppid = %d \n\n", getpid(), getppid());
for (i=0; i<100; i=i+5)
{ for (j=0; j<100000; j++);
printf("\t\t\t In Parent %d \n", i);
}

wait(0); /* wait for child to terminate */
printf("In Parent: Now the child has terminated \n");
}
else
{ printf("\t In child \n");
printf("\t pid = %d and ppid = %d \n\n", getpid(), getppid() );
for (i=0; i<100; i=i+10)
{ for (j=0; j<100000; j++);
printf("\t In child %d \n", i);
} } }
```

The reader should carefully examine the structure of the code above. In particular, note how the return value of system call *fork()* is utilized. On perusing the code we note that, the code is written to *exec*ute in different parts of the program code for the child and the parent. The program makes use of true return value of *fork()* to print “In parent", i.e. if the parent process is presently *exec*uting. The dummy loop not only slows down the *exec*ution but also ensures that we obtain interleaved outputs with a manageable number of lines on the viewing screen.

Response of this program:

```
[bhatt@iiitbsun IPC]$./a.out
In child
pid = 22484 and ppid = 22483
In child 0
In child 10
In child 20
In Parent
pid = 22483 and ppid = 22456
In Parent 0
In Parent 5
In Parent 10
In Parent 15
In child 30

.......
.......
In child 90
In Parent 20
In Parent 25
......
......
In Parent: Now the child has terminated;
```

Let us study the response. From the response, we can determine when the parent process was *exec*uting and when the child process was *exec*uting. The final line shows the result of the *exec*ution of line following wait command in parent. It *exec*utes after the child has fallen through its code. Just as we used a wait command in the parent, we could have also used an exit command explicitly in the child to exit its *exec*ution at any stage. The command pair wait and exit are utilized to have inter-process communication. In particular, these are used to synchronize activities in processes. This program demonstrated how a process may be spawned. However, what one would wish to do is to spawn a process and have it *exec*ute a planned task. Towards this objective, we shall next populate the child code segment with a code for a specified task.

1.3 Assigning Task to a Newly Spawned Process

By now one thing should be obvious: if the child process is to *exec*ute some other code, then we should first identify that *exec*utable (the one we wish to see *exec*uted). For our example case, let us first generate such an *exec*utable. We compile a program entitled *get_int.c* with the command line cc *get_int.c -o int.o*. So, when *int.o exec*utes, it reads in an integer.

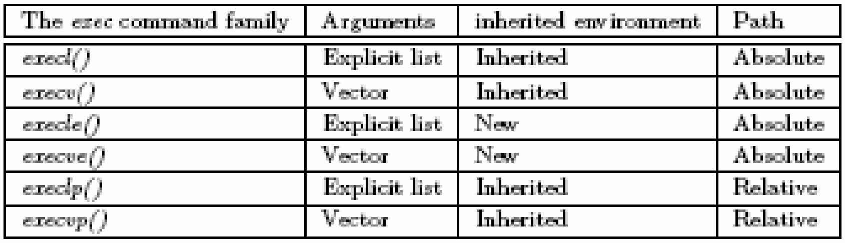
The program to get an integer :

```
#include <stdio.h>
#include <ctype.h>
int get_integer( n_p )
int *n_p;
{ int c;
int mul, sign;
int integer_part;
*n_p = 0;
mul = 10;
while( isspace( c = getchar() ) ); /* skipping white space */
if( !isdigit(c) && c != EOF && c != '+' && c != '-' )
{ /* ungetchar(c); */
printf("Found an invaild character in the integer description \n");
return 0;
}

if (c == '-') sign = -1.0;
if (c == '+') sign = 1.0;
if (c == '-' || c == '+' ) c = getchar();
for ( integer_part = 0; isdigit(c); c = getchar() )
{ integer_part = mul * integer_part + (c - '0');
};

*n_p = integer_part;
if ( sign == -1 ) *n_p = - *n_p;
if ( c == EOF ) return (*n_p);
}
main()
{ int no;
int get_integer();
printf("Input a number as signed or unsigned integer e.g. +5 or -6 or 23\n");
get_integer(&no);
printf("The no. that was input was %d \n", no);
}
```

Clearly, our second step is to have a process spawned and have it *exec*ute the program *int.o*. Unix offers a way of directing the *exec*ution from a specified code segment by using an *exec* command. In the program given below, we spawn a child process and populate its code segment with the program *int.o* obtained earlier. We shall entitle this program as *int_wait.c*.

Program *int_wait.c*

```
#include <stdio.h>
main()
{
if (fork() == 0)
{ /* In child process execute the selected command */
execlp("./int.o", "./int.o", 0);
printf("command not found \n"); /* execlp failed */
fflush(stdout);
```



The *exec* command family	Arguments	inherited environment	Path
execl()	Explicit list	Inherited	Absolute
execv()	Vector	Inherited	Absolute
execle()	Explicit list	New	Absolute
execve()	Vector	New	Absolute
execlp()	Explicit list	Inherited	Relative
execvp()	Vector	Inherited	Relative

Table 7.2: The *exec* command description.

```
exit(1);

}
else
{ printf("Waiting for the child to finish \n");
wait(0);
printf("Waiting over as child has finished \n");
}
}
```

To see the programs in action follow the steps:

1. cc *get_int.c* -o *int.o*

2. cc *int_wait.c*

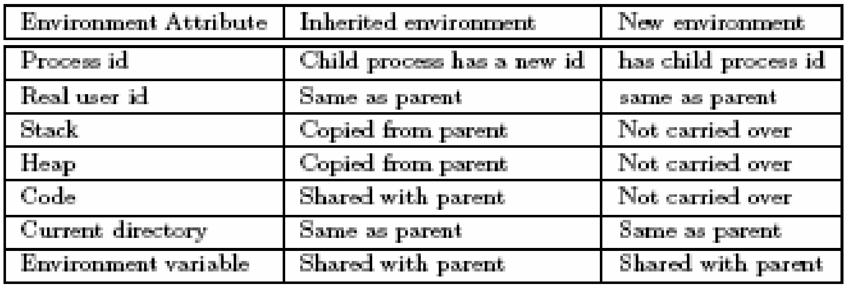
3. ./a.out

The main point to note here is that the forked child process gets populated by the code of program *int.o* with the parent *int_wait.c*. Also, we should note the arguments communicated in the *exec* command line.

Before we discuss some issues related to the new *exec*ution environment, a short discussion on *exec* command is in order. The *exec* family of commands comes in several flavors. We may choose an *exec* command to *exec*ute an identified *exec*utable defined using a relative or absolute path name. The *exec*() command may use some other arguments as well. Also, it may be *exec*uted with or without the inherited *exec*ution environment.

Most Unix systems support *exec* commands with the description in Table 7.2. The example above raises a few obvious questions. The first one is: Which are the properties the child retains after it is populated by a different code segment? In Table 7.3 we note that the process ID and user ID of the child process are carried over to the implanted process. However, the data and code segments obtain new information. Though, usually, a child process inherits open file descriptors from the parent, the implanted process may have some restrictions based on file access controls.

With this example we now have a way to first spawn and then populate a child process with the code of an arbitrary process. The implanted process still remains a child process but has its code independent of the parent. A process may spawn any number of child processes. However, much ingenuity lies in how we populate these processes and what form of communication we establish amongst these to solve a problem.



Environment Attribute	Inherited environment	New environment
Process id	Child process has a new id	has child process id
Real user id	Same as parent	same as parent
Stack	Copied from parent	Not carried over
Heap	Copied from parent	Not carried over
Code	Shared with parent	Not carried over
Current directory	Same as parent	Same as parent
Environment variable	Shared with parent	Shared with parent

Table 7.3: New environment description.

1.4 Establishing Inter-process Communication

In this section we shall study a few inter-process communication mechanisms. Each of these uses a different method to achieve communication amongst the processes. The first mechanism we study employs pipes. Pipes, as used in commands like *ls|more*, direct the output stream of one process to feed the input of another process. So for IPC, we need to create a pipe and identify the direction of feeding the pipe. Another way to communicate would be to use memory locations. We can have one process write into a memory location and expect the other process to read from it. In this case the memory location is a shared memory location. Finally, there is one more mechanism in which one may send a message to another process. The receiving process may interpret the message. Usually, the messages are used to communicate an event. We next study these mechanisms.

1.4.1 Pipes as a Mechanism for Inter-process Communication

Let us quickly recap the scheme which we used in section 1.3. The basic scheme has three parts: spawn a process; populate it and use the wait command for synchronization. Let us now examine what is involved in using pipes for establishing a communication between two processes. As a first step we need to identify two *exec*utables that need to communicate. As an example, consider a case where one process gets a character string input and communicates it to the other process which reverses strings. Then we have two processes which need to communicate. Next we define a pipe and connect it between the processes to facilitate communication. One process gets input strings and writes into the pipe. The other process, which reverses strings, gets its input (i.e. reads) from the pipe. Figure 7.2 explains how the pipes are used. As shown in the upper part of the figure, a pipe has an input end and an output end. One can write into a pipe from the input end and read from the output end. A pipe descriptor, therefore, has an array that stores two pointers. One pointer is for its input end and the other is for its output end. When a process defines a pipe it gets both the addresses, as shown in the middle part of Figure 7.2. Let us suppose array pp is used to store the descriptors. pp[0] stores the write end address and pp[1] stores the read end address. Suppose two processes, Process A and Process B, need to

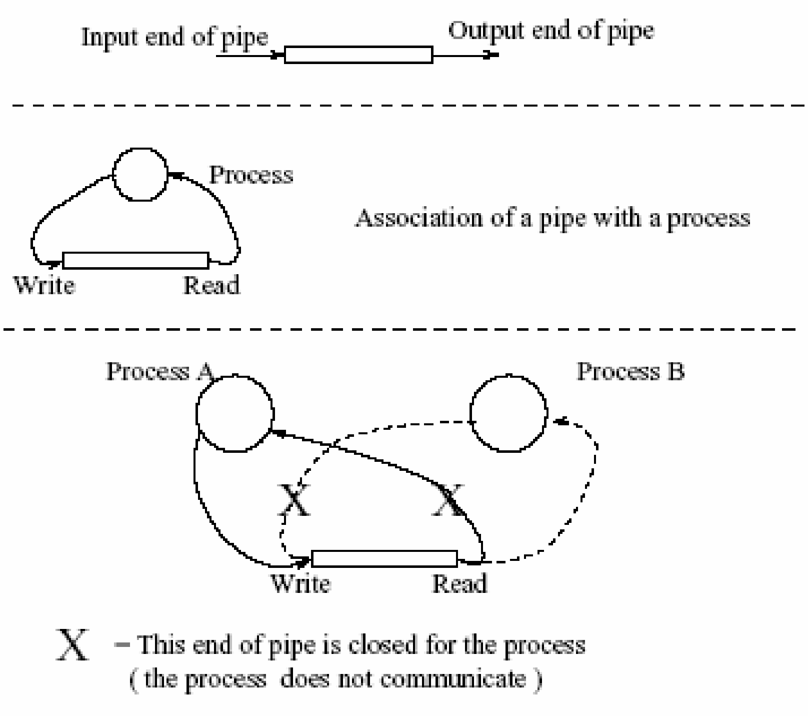

Input end of pipe
Output end of pipe
Process
Write
Read
Association of a pipe with a process
Process A
Process B
X
X
Write
Read
X
= This end of pipe is closed for the process
(the process does not communicate)

Figure 7.2: Pipe as an IPC Mechanism.

communicate, then it is imperative that the process which writes closes its read end of the pipe and the process which read closes its write end of the pipe. Essentially, for a communication from Process A to process B the following should happen. Process A should keep its write end open and close read end of the pipe. Similarly, Process B should keep its read end open and close its write end. This is what is shown in the lower part of Figure 7.2. Let us now describe how we may accomplish this.

1. First we have a parent process which declares a pipe in it.
2. Next we spawn two child processes. Both of these would get the pipe definition which we have defined in the parent. The child processes, as well as the parent, have both the write and read ends of the pipe open at this time.
3. Next, one child process, say Process A, closes its read end and the other child process, Process B, closes its write end.
4. The parent process closes both write and read ends.
5. Next, Process A is populated with code to get a string and Process B is populated to reverse a string.

With the above arrangement the output from Process A is piped as input to Process B. The programs given below precisely achieve this.

In reading the programs, the following interpretations have to be borne in mind:

1. The pipe is defined by the declaration *pipe*(p_des).
2. The *dup* command replaces the standard I/O channels by pipe descriptors.
3. The *execlp* command is used to populate the child process with the desired code.
4. The close command closes the appropriate ends of the pipe.
5. The *get_str* and *rev_str* processes are pre-compiled to yield the required executables.

The reader should be able to now assemble the programs correctly to see the operation of the programs given below:

pipe.c

```
#include <stdio.h>
#include <ctype.h>
main()
{ int p_des[2];
pipe( p_des ); /* The pipe descriptor */
printf("Input a string \n");
if ( fork () == 0 )
{
dup2(p_des[1], 1);
close(p_des[0]); /* process-A closing read end of the pipe */
execlp("./get_str", "get_str", 0);
/*** exit(1); ***/
}

else
if ( fork () == 0 )
{ dup2(p_des[0], 0);
close(p_des[1]); /* process-B closing write end of the pipe */
execlp("./rev_str", "rev_str", 0);
/*** exit(1); ****/
}

else
{ close(p_des[1]); /* parent closing both the ends of pipe */
close(p_des[0]);
wait(0);
wait(0);
}
fflush(stdout);
}
```

get_str.c

```
#include <stdio.h>
#include <ctype.h>
void get_str(str)
char str[];
{ char c;
int ic;
c = getchar();
ic = 0;
while ( ic < 10 && ( c != EOF && c != '\n' && c != '\t' ))
{ str[ic] = c;
c = getchar();
ic++;
}

str[ic] = '\0';
return;
}
```

rev_str.c

```
void rev_str(str1, str2)
char str1[];
char str2[];
{ char c;
int ic;
int rc;

ic = 0;
c = str1[0];
while( ic < 10 && (c != EOF && c != '\0' && c != '\n') )
{ ic++;
c = str1[ic];
}
str2[ic] = '\0';
rc = ic - 1;
ic = 0;
while (rc-ic > -1)
{ str2[rc-ic] = str1[ic];
ic++;
}
return;
}
```

It is important to note the following about pipes as an IPC mechanism:

1. Unix pipes are buffers managed from within the kernel.
2. Note that as a channel of communication, a pipe operates in one direction only.
3. Some plumbing (closing of ends) is required to use a pipe.
4. Pipes are useful when both the processes are schedulable and are resident on the same machine. So, pipes are not useful for processes across networks.
5. The read end of a pipe reads any way. It does not matter which process is connected to the write end of the pipe. Therefore, this is a very insecure mode of communication.
6. Pipes cannot support broadcast.

There is one other method of IPC using special files called “named pipes". We shall leave out its details. Interested readers should explore the suggested reading list of books. In particular, books by Stevenson, Chris Brown or Leach [28], [12], [24] are recommended.

1.4.2 Shared Files

One very commonly employed strategy for IPC is to share files. One process, identified as a writer process, writes into a file. Another process, identified as a reader process, reads from this file. The write process may continually make changes in a file and the other may read these changes as these happen. Unlike other IPC methods, this method does not require special system calls. It is, therefore, relatively easily portable. Of course, for creating processes we shall use the standard system calls *fork()* and *execlp()*. Besides these, there are no other system calls needed. However, we do need code for file creation, access and operations on files. A word of caution is in order. If the reader is faster than the writer, then this method shall have errors. Similarly, if a writer continues writing then the file may grow to unbounded lengths. Both these situations result in errors. This problem of a mismatch in the speed of reader and writer is called the reader writer problem. We earlier learned to resolve similar problems using mutual exclusion of resource sharing. In this case too we can program for mutually exclusive writes and reads.

Shared file pointers: Another way to handle files would be to share file pointers instead of files themselves. Sharing the file pointers with mutual exclusion could be easily done using semaphores.

The shared file pointer method of IPC operates in two steps. In the first step, one process positions a file pointer at a location in a file. In the second step, another process reads from this file from the communicated location. Note that if the reader attempts to read a file even before the writer has written something on a file, we shall have an error. So, in our example we will ensure that the reader process sleeps for a while (so that the writer has written some bytes). We shall use a semaphore simulation to achieve mutual exclusion of access to the file pointer, and hence, to the file.

This method can be used when the two processes are related. This is because the shared file pointer must be available to both. In our example, these two processes shall be a parent and its child. Clearly, if a file has been opened before the child process is spawned, then the file descriptors created by the parent are available to the child process as well. Note that when a process tries to create a file which some other process has already created, then an error is reported.

To understand the programs in the example, it is important to understand some instructions for file operations. We shall use *lseek*() system command. It is used to access a sequence of bytes from a certain offset in the file. The first byte in the file is considered to have an offset of 0. It has the syntax long *lseek*(int fd, long offset, int arg) with the following interpretation.

- With arg = 0, the second argument is treated as an offset from the first byte in file.
- With arg = 1, the current position of the file pointer is changed to sum of the current file pointer and the value of the second argument.

- With arg = 2, the current position of the file pointer is changed to the sum of the size of file and value of the second argument. The value of the second argument can be negative as long as the overall result of the sum is positive or zero.

The example here spans three programs, a main, a reader and a writer program. Let us look at the code for the main program.

```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation();
main(argc, argv)
int argc;
char *argv[];
{/* the program communicates from parent to child using a shared file pointer */
FILE *fp;
char message[MAXBYTES];
long i;
int mess_num, n_bytes, j, no_of_mess;
int sid, status;
if ( argc < 3 )
{ fputs("Bad argument count \n", stderr);
fputs("Usage: num_messages num_bytes \n", stderr);

exit(1);
}
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("no_of_mess : %6d and n_bytes : %6d \n", no_of_mess, n_bytes );
if(n_bytes > MAXBYTES)
{ fputs("Number of bytes exceeds maximum", stderr);
exit(1);

} /* open a file before creating a child process to share a file pointer*/
else if( ( fp = fopen("./temp_file", "w+" )) == NULL )
{ fputs("Cannot open temp_file for writing \n", stderr);
exit(1);
}
/* create processes and begin communication */
switch (fork ())
{ case -1: fputs("Error in fork ", stderr);
exit( 1 );

case 0: sleep(2);
if(execlp("./readfile", "./readfile", argv[1], argv[2], NULL) == -1)
fputs("Error in exec in child \n", stderr);
exit( 1 );
default: if(execlp("./writefile", "./writefile", argv[1], argv[2], NULL) == -1)
fputs("Error in exec in parent \n", stderr);
exit( 1 );
} /* end switch */
}
```

Now we describe the reader process.

```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation()
{ if (creat( "creation", 0444) == -1)
{ fputs("Error in create \n", stderr);
system("rm creation");
}

else fputs(" No error in creat \n", stderr);
}
main (argc, argv)
int argc;
char *argv[];

{ FILE *fp;
long i;
char message[MAXBYTES];
int mess_num, n_bytes, j, no_of_mess;
int sid, status;
void sem_simulation();
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("in read_child \n");
/* read messages from the shared file */
for ( i=0; i < no_of_mess; i++ )
{ sem_simulation();
fseek(fp, i*n_bytes*1L, 0);
while((fgets(message, n_bytes+1, fp)) == NULL ) ;
fseek(fp, i*n_bytes*1L, 0);
sem_simulation();
} /* end of for loop *

exit(0);
}
```

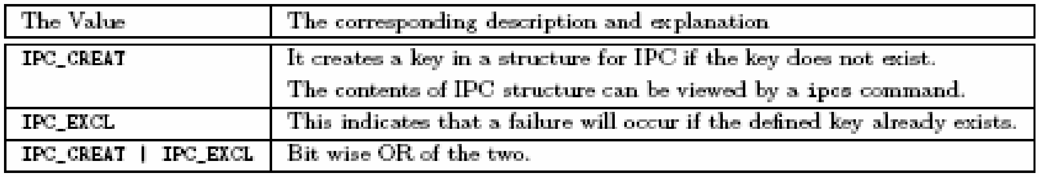
Now let us describe the writer process.

```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation()
{ if (creat( "creation", 0444) == -1)
{ fputs("Error in create \n", stderr);
system("rm creation");
}
else fputs(" No error in create \n", stderr);
}
main (argc, argv)
int argc;

char *argv[];
{ FILE *fp;
long i, j, status, message_num;
char message[MAXBYTES];
int n_bytes, no_of_mess;
void sem_simulation();
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("in parent with write option \n");
printf("no_of_mess : %6d n_bytes : %6d \n");
for ( i=0; i < no_of_mess; i++ )
{ /* Create a message with n_bytes */
message_num = i;
for ( j = message_num; j < n_bytes; j++ )

message[j] = 'd';
printf("%s \n", message);
/* Use semaphore to control synchronization, write to end of file */
sem_simulation();
fseek(fp, 0L, 2);
while( ( fputs(message, fp) ) == -1)
fputs("Cannot write message", stderr );
fseek(fp, 0L, 2);
sem_simulation();
}
wait(&status);
unlink("creation");
unlink("./temp_file");
fclose(fp);
}
```

The shared file pointer method is quite an elegant solution and is often a preferred solution where files need to be shared. However, many parallel algorithms require that “objects" be shared. The basic concept is to share memory. Our discussion shall now veer to IPC using shared memory communication.

1.4.3 Shared Memory Communication

Ordinarily, processes use memory areas within the scope of virtual memory space. However, memory management systems ensure that every process has a well-defined and distinct data and code area. For shared memory communication, one process would write into a certain commonly accessed area and another process would read subsequently from that area. One other point which we can debate is: do the processes have to be related? We have seen that a parent may share a data area or files with a child. Also, by using the *exec*() function call we may be able to populate a process with another code segment or data. Clearly, the shared memory method can allow access to a common data area even amongst the processes that are not related. However, in that case an area like a process stack may not be shareable. Also, it should be noted that it is important that the shared data integrity may get compromised when an arbitrary sequence of reads and writes occurs. To maintain data integrity, the access is planned carefully under a user program control. That then is the key to shared memory protocol.



The Value	The corresponding description and explanation
IPC_CREAT	It creates a key in a structure for IPC if the key does not exist. The contents of IPC structure can be viewed by a ipcs command.
IPC_EXCL	This indicates that a failure will occur if the defined key already exists.
IPC_CREAT | IPC_EXCL	Bit wise OR of the two.

Table 7.4: Creating a shared data mechanism.

The shared memory model has the following steps of execution.

1. First we have to set up a shared memory mechanism in the kernel.
2. Next an identified \safe area" is attached to each of the processes.
3. Use this attached shared data space in a consistent manner.
4. When finished, detach the shared data space from all processes to which it was attached.
5. Delete the information concerning the shared memory from the kernel.

Two important *.h* files in this context are: *shm.h* and *ipc.h* which are included in all the process definitions. The first step is to set up shared memory mechanism in kernel. The required data structure is obtained by using *shmget()* system call with the following syntax.

int shmget(key_t key, int size, int flag);

The parameter key_t is usually a long int. It is declared internally as key_t key. key_t is an alias defined in sys/types.h using a typedef structure. If this key is set to IPC_PRIVATE, then it always creates a shared memory region. The second parameter, size is the size of the sh-mem-region in bytes. The third parameter is a combination of usual file access permissions of r/w/e for o/g/w with the interpretation of non-zero constants as explained in Table 7.4. A successful call results in the creation of a shared memory data structure with a defined id. This data structure has the following information in it.

```
struct shmid_ds
{ struct ipc_perm shm_perm;
int shm_seg_segsz /* size of segments in bytes */
struct region *shm_reg; /* pointer to region struct */
char pad[4]; /* for swap compatibility */
ushort shm_lpid; /* pid of last shmop */
ushort shm_cpid; /* pid of creator */
ushort shm_nattch; /* used for shm_info */
ushort shm_cnattch; /* used for shm_info */
time_t shm_atime; /* last attach time */
time_t shm_dtime; /* last detach time */
time_t shm_ctime; /* last change time */
}
```

Once this is done we would have created a shared memory data space. The next step requires that we attach it to processes that would share it. This can be done using the system call *shmat()*. The system call *shmat()* has its syntax shown below.

```
char *shamt( int shmid, char *shmaddr, int shmflg );
```

The second argument should be set to zero as in (char *)0, if the kernel is to determine the attachment. The system uses three possible flags which are: SHM_RND, SHM_RDONLY and the combination SHM_RND | SHM_RDONLY. The SHM_RDONLY flag indicates the shared region is read only. Otherwise, it is both for read and write operations. The flag SHM_RND requires that the system enforces use of the byte address of the shared memory region to coincide with a double word boundary by rounding.

Now that we have a well-defined shared common area, reading and writing can be done in this shared memory region. However, the user must write a code to ensure locking of the shared region. For instance, we should be able to block a process attempting to write while a reader process is reading. This can be done by using a synchronization method such as semaphores. In most versions of Unix, semaphores are available to enforce mutual exclusion. At some stage a process may have finished using the shared memory region. In that case this region can be detached for that process. This is done by using the *shmdt()* system call. This system call detaches that process from future access. This information is kept within the kernel data-space. The system call *shmdt()* takes a single argument, the address of the shared memory region. The return value from the system call is rarely used except to check if an error has occurred (with -1 as the return value). The last step is to clean up the kernel's data space using the system call *shmctl()*. The system call *shmctl()* takes three parameters as input, a shared memory id, a set of flags, and a buffer that allows copying between the user and the kernel data space.

A considerable amount of information is pointed to by the third parameter. A call to *shmctl()* with the command parameter set to IPC_STAT gives the following information.

- User's id
- Creator's group id
- Operation permissions
- Key
- segment size
- Process id of creator *
- Current number of attached segments in the memory.
- Last time of attachment
- User's group id
- Creator's id
- Last time of detachment
- Last time of change
- Current no. of segments attached
- Process id of the last shared memory operation

Now let us examine the *shmget()* system call.
int shmget(key_t key, int region_size, int flags);

Here key is a user-defined integer, the size of the shared region to be attached is in bytes. The flags usually turn on the bits in IPC_CREAT. Depending upon whether there is key entry in the kernel's shared memory table, the *shmget()* call takes on one of the following two actions. If there is an entry, then *shmget()* returns an integer indicating the position of the entry. If there is no entry, then an entry is made in the kernel's shared memory table. Also, note that the size of the shared memory is specified by the user. It, however, should satisfy some system constraints which may be as follows.

```
struct shminfo
{ int shmmax, /* Maximum shared memory segment size 131072 for some */
shmmin, /* minimum shared memory segment size 1 for some */
shmni, /* No. of shared memory identifiers */
shmseg, /* Maximum attached segments per process */
shmall; /* Max. total shared memory system in pages */
};
```

The third parameter in *shmget()* corresponds to the flags which set access permissions as shown below:

```
400 read by user ...... Typically in shm.h file as constant SHM_R
200 write by user .......Typically in shm.h file as constant SHM_W
040 read by group
020 write by group
004 read by others
002 read by others ......All these are octal constants.
```

For example, let us take a case where we have read/write permissions by the user's group and no access by others. To be able to achieve this we use the following values.

SHM_R | SHM_W | 0040 | IPC_CREAT as a flag to a call to *shmget().*

Now consider the *shmat()* system call.

```
char *shmat( int shmid, char *address, int flags );
```

This system call returns a pointer to the shared memory region to be attached. It must be preceded by a call to *shmget*(). The first argument is a *shmid* (returned by *shmget*()). It is an integer. The second argument is an address. We can let the compiler decide where to attach the shared memory data space by giving the second argument as (char *) 0. The flags in arguments list are to communicate the permissions only as SHM_RND and SHM_RDONLY. The *shmdt()* system call syntax is as follows:

```
int shmdt(char * addr );
```

This system call is used to detach. It must follow a call *shmat()* with the same base address which is returned by *shmat()*. The last system call we need is *shmctl().* It has the following syntax.

```
int shmctl( int shmid, int command, struct shm_ds *buf_ptr );
```

The *shmctl()* call is used to change the ownership and permissions of the shared region. The first argument is the one earlier returned by *shmget()* and is an integer. The command argument has five possibilities:

- IPC_STAT : returns the status of the associated data structure for the shared memory pointed by buffer pointer.
- IPC_RMID : used to remove the shared memory id.
- SHM_LOCK : used to lock
- SHM_UNLOCK : used to unlock
- IPC_SET : used to set permissions.

When a region is used as a shared memory data space it must be from a list of free data space. Based on the above explanations, we can arrive at the code given below.

```
include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define MAXBYTES 4096 /* Maximum bytes per shared segment */
main(argc, argv)
int argc;
char *argv[];
{ /* Inter process communication using shared memory */
char message[MAXBYTES];

int i, message_num, j, no_of_mess, nbytes;
int key = getpid();
int semid;
int segid;
char *addr;
if (argc != 3) { printf("Usage : %s num_messages");
printf("num_of_bytes \n", argv[0]);
exit(1);
}

else
{ no_of_mess = atoi(argv[1]);
nbytes = atoi(argv[2]);
if (nbytes > MAXBYTES) nbytes = MAXBYTES;
if ( (semid=semget( (key_t)key, 1, 0666 | IPC_CREAT ))== -1)
{ printf("semget error \n");
exit(1);
}
/* Initialise the semaphore to 1 */
V(semid);
if ( (segid = shmget( (key_t) key, MAXBYTES, 0666 |
IPC_CREAT ) ) == -1 )
{ printf("shmget error \n");
exit(1);
}

/*if ( (addr = shmat(segid, (char * )0,0)) == (char *)-1) */
if ( (addr = shmat(segid, 0, 0)) == (char *) -1 )
{ printf("shmat error \n");
exit(1);
}
switch (fork())
{ case -1 : printf("Error in fork \n");
exit(1);

case 0 : /* Child process, receiving messages */
for (i=0; i < no_of_mess; i++)
if(receive(semid, message, sizeof(message)));
exit(0);
default : /* Parent process, sends messages */
for ( i=0; i < no_of_mess; i++)
{ for ( j=i; j < nbytes; j++)
message[j] = 'd';
if (!send(semid, message, sizeof(message)))
printf("Cannot send the message \n");
} /* end of for loop */
} /* end of switch */
} /* end of else part */
}

/* Semaphores */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
int sid;
cleanup(semid, segid, addr)
int semid, segid;
char *addr;
{ int status;
/* wait for the child process to die first */
/* removing semaphores */
wait(&status);
semctl(semid, 0, IPC_RMID, 0);
shmdt(addr);
```

```
shmctl(segid, 0, IPC_RMID, 0);

};

P(sid)
int sid;
{ /* Note the difference in this and previous structs */
struct sembuf *sb;
sb = (struct sembuf *) malloc(sizeof(struct sembuf *));
sb -> sem_num = 0;
sb -> sem_op = -1;
sb -> sem_flg = SEM_UNDO;
if( (semop(sid, sb, 1)) == -1) puts("semop error");
};

V(sid)
int sid;
{ struct sembuf *sb;
sb = (struct sembuf *) malloc(sizeof(struct sembuf *));
sb -> sem_num = 0;
sb -> sem_op = 1;
sb -> sem_flg = SEM_UNDO;
if( (semop(sid, sb, 1)) == -1) puts("semop error");
};

/* send message from addr to buf */
send(semid, addr, buf, nbytes)
int semid;
char *addr, *buf;
int nbytes;
{ P(semid);
memcpy(addr, buf, nbytes);
V(semid);
}
/* receive message from addr to buf */
receive(semid, addr, buf, nbytes)
int semid;
char *addr, *buf;

int nbytes;
{ P(semid);
memcpy(buf, addr, nbytes);
V(semid);
}
```

From the programs above, we notice that any process is capable of accessing the shared memory area once the key is known to that process. This is one clear advantage over any other method. Also, within the shared area the processes enjoy random access for the stored information. This is a major reason why shared memory access is considered efficient. In addition, shared memory can support many-to-many communication quite easily. We shall next explore message-based IPC.

1.4.4 Message-Based IPC

Messages are a very general form of communication. Messages can be used to send and receive formatted data streams between arbitrary processes. Messages may have types. This helps in message interpretation. The type may specify appropriate permissions for processes. Usually at the receiver end, messages are put in a queue. Messages may also be formatted in their structure. This again is determined by the application process.

Messages are also the choice for many parallel computers such as Intel's hyper-cube. The following four system calls achieve message transfers amongst processes.

- *msgget()* returns (and possibly creates) message descriptor(s) to designate a message queue for use in other systems calls.
- *msgctl()* has options to set and return parameters associated with a message descriptor. It also has an option to remove descriptors.
- *msgsnd()* sends a message using a message queue.
- *msgrcv()* receives a message using a message queue.

Let us now study some details of these system calls.

msgget() system call : The syntax of this call is as follows:

i*nt msgget(key_t key, int flag);*

The *msgget()* system call has one primary argument, the key, a second argument which is a flag. It returns an integer called a qid which is the id of a queue. The returned qid is an index to the kernel's message queue data-structure table. The call returns -1 if there is an

error. This call gets the resource, a message queue. The first argument key_t, is defined in sys/types.h file as being a long. The second argument uses the following flags:

- MSG_R : The process has read permission
- MSG_W : The process has write permission
- MSG_RWAIT : A reader is waiting to read a message from message queue
- MSG_WWAIT : A writer is waiting to write a message to message queue
- MSD_LOCKED : The msg queue is locked

- MSG_LOCKWAIT : The msg queue is waiting for a lock
- IPC_NOWAIT : Described earlier
- IPC_EXCL : ....

In most cases these options can be used in bit-ored manner. It is important to have the readers and writers of a message identify the relevant queue for message exchange. This is done by associating and using the correct qid or key. The key can be kept relatively private between processes by using a *makekey()* function (also used for data encryption).

For simple programs it is probably sufficient to use the process id of the creator process (assuming that other processes wishing to access the queue know it). Usually, kernel uses some algorithm to translate the key into qid. The access permissions for the IPC methods are stored in IPC permissions structure which is a simple table. Entries in kernel's message queue data structures are C structures. These resemble tables and have several fields to describe permissions, size of queue, and other information. The message queue data structure is as follows.

```
struct meqid_ds
{ struct ipc_perm meg_perm; /* permission structure */
struct msg *msg_first; /* pointer to first message */
struct msg *msg_last; /* ........... last ..........*/
ushort msg_cbytes; /* no. of bytes in queue */
ushort msg_qnum; /* no. of messages on queue */
ushort msg_qbytes; /* Max. no. of bytes on queue */
ushort msg_lspid; /* pid of last msgsnd */
ushort msg_lrpid; /* pid of the last msgrcv */
time_t msg_stime; /* last msgsnd time */
time_t msg_rtime; /* .....msgrcv................*/

time_t msg_ctime; /* last change time */
}
```

There is one message structure for each message that may be in the system.

```
struct msg
{ struct msg *msg_next; /* pointer to next message */
long msg_type; /* message type */
ushort msg_ts; /* message text size */
ushort msg_spot; /* internal address */
```

Note that several processes may send messages to the same message queue. The “type" of message is used to determine which process amongst the processes is the originator of the message received by some other process. This can be done by hard coding a particular number for type or using process-id of the sender as the msg_type. The *msgctl()* function call: This system call enables three basic actions. The most obvious one is to remove message queue data structure from the kernel. The second action allows a user to examine the contents of a message queue data structure by copying them into a buffer in user's data area. The third action allows a user to set the contents of a message queue data structure in the kernel by copying them from a buffer in the user's data area. The system call has the following syntax.

*int msgctl(int qid, int command, struct msqid_ds *ptr);*

This system call is used to control the resource (a message queue). The first argument is the qid which is assumed to exist before call to *msgctl().* Otherwise the system is in error state. Note that if *msgget()* and *msgctl()* are called by two different processes then there is a potential for a \race" condition to occur. The second argument command is an integer which must be one of the following constants (defined in the header file sys/msg.h).

- IPC STAT: Places the contents of the kernel structure indexed by the first argument, qid, into a data structure pointed to by the third argument, ptr. This enables the user to examine and change the contents of a copy of the kernel's data structure, as this is in user space.
- IPC SET: Places the contents of the data structures in user space pointed to by the third argument, ptr, into the kernel's data structure indexed by first argument qid, thus enabling a user to change the contents of the kernel's data structure. The only fields that a user can change are msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.
- IPC RMID : Removes the kernel data structure entry indexed by qid.

The msgsnd() and msgrcv() system calls have the following syntax.

*int msgsnd(int qid, struct msgbuf *msg_ptr, int message_size, int flag);*

*int msgrcv(int qid, struct msgbuf *msg_ptr, int message_size, int msgtype, int flag);*

Both of these calls operate on a message queue by sending and receiving messages respectively. The first three arguments are the same for both of these functions. The syntax of the buffer structure is as follows.

struct msgbuf{ long mtype; char mtext[1]; }

This captures the message type and text. The flags specify the actions to be taken if the queue is full, or if the total number of messages on all the message queues exceeds a prescribed limit. With the flags the following actions take place. If IPC_NOWAIT is set, no message is sent and the calling process returns without any error action. If IPC_NOWAIT is set to 0, then the calling process suspends until any of the following two events occur.

1. A message is removed from this or from other queue.
2. The queue is removed by another process. If the message data structure indexed by qid is removed when the flag argument is 0, an error occurs (*msgsnd()* returns -1).

The fourth arg to *msgrcv()* is a message type. It is a long integer. The type argument is used as follows.

- If the value is 0, the first message on the queue is received.
- If the value is positive, the queue is scanned till the first message of this type is received. The pointer is then set to the first message of the queue.
- If the value is -ve, the message queue is scanned to find the first message with a type whose value is less than, or equal to, this argument.

The flags in the *msgrcv()* are treated the same way as for *msgsnd().*

A successful *exec*ution of either *msgsnd()*, or *msgrcv()* always updates the appropriate entries in msgid_ds data structure. With the above explanation, let us examine the message passing program which follows.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
{ int status, pid, pid1;
if (( pid=fork())==0) execlp("./messender", "messender", argv[1], argv[2], 0);
if (( pid1=fork())==0) execlp("./mesrec", "mesrec", argv[1], 0);
wait(&status); /* wait for some child to terminate */
wait(&status); /* wait for some child to terminate */
}
```

Next we give the message sender program.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
/* This is the sender. It sends messages using IPC system V messages queues.*/
/* It takes two arguments : */
/* No. of messages and no. of bytes */
/* key_t MSGKEY = 100; */
/* struct msgformat {long mtype; int mpid; char mtext[256]} msg; */

{
key_t MSGKEY = 100;
struct msgformat { long mtype;
int mpid;
char mtext[256];
} msg;
int i ;
int msgid;
int loop, bytes;

extern cleanup();
loop = atoi(argv[1]);
bytes = atoi(argv[2]);
printf("In the sender child \n");
for ( i = 0; i < bytes; i++ ) msg.mtext[i] = 'm';
printf("the number of 'm' s is : %6d \n", i);
msgid = msgget(MSGKEY, 0660 | IPC_CREAT);
msg.mtype = 1;
msg.mpid = getpid();
/* Send number of messages specified by user argument */
for (i=0; i<loop; i++) msgsnd(msgid, &msg, bytes, 0);
printf("the number of times the messages sent out is : %6d \n", i);
/* Cleaning up; maximum number queues 32 */
for (i=0; i<32; i++) signal(i, cleanup);

}
cleanup()
{ int msgid;
msgctl(msgid, IPC_RMID, 0);
exit(0);
}
```

|Now we give the receiver program listing.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
/* The receiver of the two processes communicating message using */
/* IPC system V messages queues. */

/* It takes two arguments: No. of messages and no. of bytes */
/* key_t MSGKEY = 100; */
/* struct msgformat {long mtype; int mpid; char mtext[256]} msg; */
```

```
{
key_t MSGKEY = 100;
struct msgformat { long mtype;
int mpid;
char mtext[256];
} msg;
int i, pid, *pint;
int msgid;
int loop, bytes;
msgid = msgget(MSGKEY, 0777);
loop = atoi(argv[1]);
bytes = atoi(argv[2]);
for ( i = 0; i <= bytes; i++ )
{ printf("receiving a message \n");
msgrcv(msgid, &msg, 256, 2, 0);
} }
```

If there are multiple writer processes and a single reader process, then the code shall be somewhat along the following lines.

```
if ( mesg_type == 1) { search mesg_queue for type 1; process msg_type type 1 }
.
.
if ( mesg_type == n) { search mesg_queue for type n; process msg_type type n }
```

The number and size of messages available is limited by some constant in the IPC package.

In fact this can be set in the system V IPC package when it is installed. Typically the constants and structure are as follows.

```
MSGPOOL 8
MSGMNB 2048                                /* Max. no. of bytes on queue */
MSGMNI 50                                   /* No. of msg. queue identifiers */
MSGTQL 50                                  /* No. of system message headers */
MSGMAP 100                                  /* No. of entries in msg map */
MSGMAX ( MSGPOOL *1024 )                    /* Maximum message size */
MSGSSZ 8                                   /* Message segment size */
MSGSEG (( MSGPOOL *1024 ) / MSGSSZ )        /* No. of msg. segments */
```

Finally, we may note that the message queue information structure is as follows.

```
struct msginfo{int msgmap, msgmax, msgmnb, msgmni, msgssz, msgtql; ushort msgseg}
```

From the programs above, it should be obvious that the message-based IPC can also be used for merging multiple data streams (multiplexing). As messages carry senders' id it should also be possible to do de-multiplexing. The message type may also capture priorities. Prioritizing messages can be very useful in some application contexts. Also, note that the communicating parties need not be active at the same time. In our program descriptions we used signals. Note that signals too, are messages! Signals are important and so we shall discuss these in the next subsection.

1.4.5 Signals as IPC

Within the suite of IPC mechanisms, signals stand out for one very good reason. A signal, as a mechanism, is one clean way to communicate asynchronous events. In fact, we use signals more often than any other means of IPC. Every time we abort a program using ^c, a signal is generated to break. Similarly, if an unexpected value for a pointer is generated, we have core dump and a segmentation fault recognized. When we change a window size, a signal is generated. Note that in all these examples, an event happens within the process or the process receives it as an input. In general, a process may send a signal to another process. In all these situations the process receiving a signal needs to respond. We shall first enumerate typical sources of signal, and later examine the possible forms of responses that are generated. Below we list the sources for signal during a process *exec*ution:

1. From the terminal: Consider a process which has been launched from a terminal and is running. Now if we input the interrupt character, ^c, from the keyboard then we have a signal SIGINT initiated. Suppose, we have disconnect of the terminal line (this may happen when we may close the window for instance), then there is a signal SIGHUP to capture the hanging up of the line.
2. From window manager: This may be any of the mouse activity that may happen in the selected window. In case of change of size of the window the signal is SIGWINCH.
3. From other subsystems: This may be from memory or other subsystems. For instance, if a memory reference is out of the process's data or code space, then there shall be a signal SIGSEGV.
4. From kernel: The typical usage of time in processes can be used to set an alarm. The alarm signal is SIGALARM.

5. From the processes: It is not unusual to kill a child process. In fact, sometimes we may kill a job which may have entered an infinite loop. There may be other reasons to abort a process. The typical kill signal is SIGKILL. One of the uses is when a terminal hangs, the best thing to do is to log in from another terminal and kill the hanging process. One may also look upon the last case as a shell initiated signal. Note that a shell is itself a process.

Above we have noted various sources from where signals may be generated. Usually this helps to define the signal type. A process may expect certain types of signals and make a provision for handling these by defining a set of signal handlers. The signal handlers can offer a set of responses which may even include ignoring certain signals! So next, we shall study the different kind of signal responses which processes may generate.

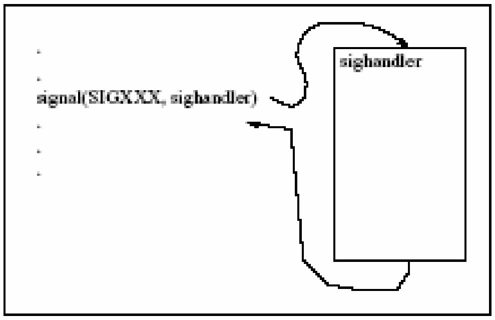


Figure 7.3: Processing signals.

In Figure 7.3 we see a program statement *signal*(SIGXXX, sighandler) to define how this process should respond to a signal. In this statement SIGXXX identifies the signal and sighandler identifies a signal service routine. In general, a process may respond to a given signal in one of the following ways.

1. Ignore it: A process may choose to ignore some kinds of signal. Since processes may receive signals from any source, it is quite possible that a process would authenticate the process before honoring the signal. In some cases then a process may simply ignore the signal and offer no response at all.
2. Respond to it: This is quite often the case in the distributed computing scenarios where processes communicate to further computations in steps. These signals may require some response. The response is encoded in the signal handler. For instance, a debugger and the process being debugged would require signal communication quite often. Another usage might be to advise a clean-up operation. For instance, we need to clean-up following the

shared memory mode of IPC. Users of Java would recognize that response for exception handling falls in the same category.

3. Reconfigure: This is required whenever system services are dynamically reconfigured. This happens often in fault-tolerant systems or networked systems. The following is a good example of dynamic configuration. Suppose we have several application servers (like WebSphere) provisioning services. A dispatcher system allocates the servers. During operations, some server may fail. This entails redeployment by the dispatcher. The failure needs to be recognized and dispatching reconfigured for future.
4. Turn on/off options: During debugging as well as profiling (as discussed in chapter on “Other Tools") we may turn some options \On" or \Off" and this may require some signals to be generated.
5. Timer information: In real-time systems, we may have several timers to keep a tab on periodic events. The ideas is to periodically generate required signals to set up services, set alarms or offer other time-based services.

In this chapter we examined the ways to establish communication amongst processes. Its a brief exposure. To comprehend the distributed computing field, it is important to look up the suggested reading list. Interested readers should explore PVM (parallel virtual machine) [30] and MPI (message passing interface) [31] as distributed computing environments.

Unit-4

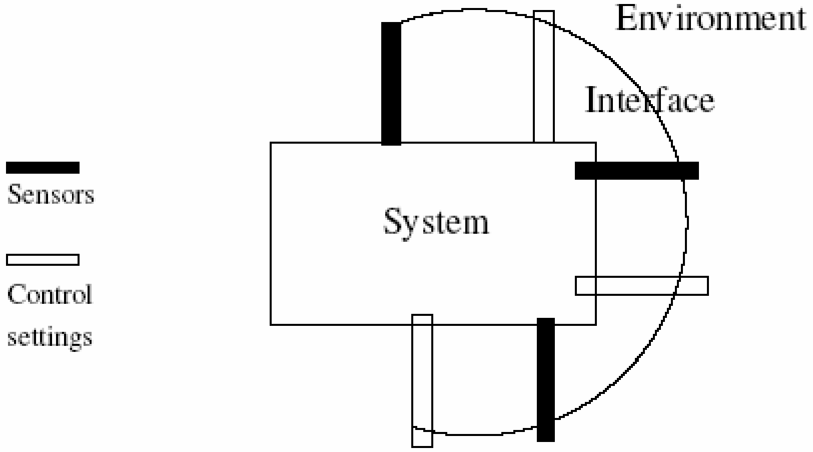
1.1 Introduction to Real-time Operating Systems and Microkernels

In some data processing applications system responses are meaningful, if these are within a certain stipulated time period. System responses that arrive later than the expected time are usually irrelevant or meaningless. In fact, sometimes, it's no better than being simply wrong. Therefore, the system response must be generated well within the stipulated time. This is true particularly of on-line stock trading, tele-ticketing and similar other transactions. These systems are generally recognized to be real-time systems. For interactive systems the responses ought to match human reaction times to be able to see the effects as well (as in on line banking or video games).

Real-time systems may be required in life critical applications such as patient monitoring systems. They may also be applied in safety critical systems such as reactor control systems in a power plant. Let us consider a safety critical application like anti-lock braking system (ABS), where control settings have to be determined in real-time. A passenger car driver needs to be able to control his automobile under adverse driving conditions.

In a car without ABS, the driver has to cleverly pump and release the brake pedal to prevent skids. Cars, with ABS control, regulate pumping cycle of brakes automatically. This is achieved by modifying the pressure applied on brake pedals by a driver in panic. A real-time control system gives a timely response. Clearly, what is a timely response is determined by the context of application. Usually, one reckons that a certain response is timely, if it allows for enough time to set the needed controller(s) appropriately, i.e. before it is too late. In safety critical, or life critical situations a delay may even result in a catastrophe. Operating systems are designed keeping in mind the context of use. As we have seen, the OS designers ensure high resource utilization and throughput in the general purpose computing context. However, for a system which both monitors and responds to events from its operative environment, the system responses are required to be timely. For such an OS, the minimalist kernel design is required. In fact, since all IO requires use of communications through kernel, it is important that kernel overheads are minimal. This has resulted in emergence of micro-kernels. Micro-kernels are minimal kernels which offer kernel services with minimum overheads. The kernels used in hard real-time systems 1 are often micro-kernels. In this chapter, we shall cover the relevant issues and strategies to design an OS which can service real-time requirements.

1.3 Characteristics of real-time systems



In a real–time system the events in the environment are detected by sensors and the responses to these events are generated in a timely manner. A RTOS ensures that control settings (in response to an event) are achieved in real–time.

Figure 8.1: Operative environment for RTOS.

A typical real-time operating environment is shown in Figure 8.1. In this figure we note that the computer system has an interface which is embedded within its environment. The operating system achieves the desired extent of regulation as follows:

1. Sense an event: The system monitors its operative environment using some sensors. These sensors keep a tab on some measurable entity. Depending upon the context of use this entity may be a measure of temperature, or a stock price fluctuation or fluid level in a reservoir. These measurements may be periodic. In that case the system would accept an input periodically. In case the measurement of inputs is taken at specified times of operation then the OS may schedule its input at these specified times or it may be interrupted to accept the input. The input may even be measured only when an unusual deviation in the value of the monitored entity occurs. In these cases the input would certainly result in an interrupt. Regardless of the input mode, the system would have an input following a sensor reading (which is an event).
2. Process the data: The next important task is to process the data which has been most recently acquired. The data processing may be aimed at checking the health of the system. Usually it is to determine if some action is needed.

3. Decide on an action: Usually, the processing steps involving arriving at some decisions on control settings. For instance, if the stock prices cross some threshold, then one has to decide to buy or sell or do nothing. As another example, the action may be to open a valve a little more to increase inflow in case reservoir level drops.
4. Take a corrective action: In case, the settings need to be altered, the new settings are determined and control actuators are initiated. Note that the actions in turn affect the environment. It is quite possible that as a consequence, a new set of events get triggered. Also, it is possible that the corrective step requires a drastic and an immediate step. For instance, if an alarm is to be raised, then all the other tasks have to be suspended or pre-empted and an alarm raised immediately. Real time systems quite often resort to pre-emption to prevent a catastrophe from happening.

The OS may be a bare-bone microkernel to ensure that input events are processed with minimum overhead. Usually, the sensor and monitoring instruments communicate with the rest of the system in interrupt mode. Device drivers are specifically tuned to service these inputs. In Section 1.4 we shall discuss the related design issues for micro-kernels and RTOS.

Why not use Unix or Windows? This is one very natural question to raise. Unix or Windows are operating systems that have been designed with no specific class of applications in mind. These are robust, (like all terrain vehicles), but not suitable for realtime operations (say Formula 1 cars). Their performance in real-time domain would be like that of an all terrain vehicle on a formula one race track. Note that the timeliness in response is crucial in real-time operations. General-purpose operating systems are designed to enhance throughput. Often it has considerable leeway in responding to events. Also, within a service type, the general-purpose OS cater to a very vast range of services. For example, just consider the print service. There is considerable leeway with regard to system response time. Additionally, the printer service may cater to a vast category of print devices which range from ink-jet to laser printing or from gray scale to color printing. In other words, the service rendering code is long. Additionally, it caters to a large selection in printer devices. This makes service rendering slow. Also, a few seconds of delay in printing matters very little, if at all. Real-time operative environments usually have a fixed domain of operations in which events have fairly predictable patterns, but do need monitoring and periodic checks. For instance, a vessel in a chemical process will witness fairly predictable form of rise in temperature or pressure, but needs to be monitored. This means that the scheduling strategies would be event centered or time centered. In a general-purpose computing environment the events arise from multiple, and not necessarily predictable, sources. In real-time systems, the events are fairly well known and may even have a pattern. However, there is a stipulated response time. Within this context, development of scheduling algorithms for real-time systems is a major area of research.

A natural question which may be raised is: Can one modify a general purpose OS to meet real-time requirements. Sometimes a general-purpose OS kernel is stripped down to provide for the basic IO services. This kernel is called microkernel. Microkernels do meet RTOS application specific service requirements. This is what is done in Windows CE and Embedded Linux.

Note we have made two important points above. One relates to timeliness of response and the other relates to event-centric operation. Scheduling has to be organized to ensure timeliness under event-centric operation. This may have to be done at the expense of loss of overall throughput!!

1.3.1 Classification of Real-time Systems

The classification of real-time systems is usually based on the severity of the consequences of failing to meet time constraints. This can be understood as follows. Suppose a system requires a response to an event in time period T. Now we ask: what happens if the response is not received within the stipulated time period? The failure to meet the time constraint may result in different degrees of severity of consequences. In a life-critical or safety critical application, the failure may result in a disaster such as loss of life. A case in point is the shuttle Columbia's accident in early February 2 2003. Recall Kalpana Chawla, an aeronautics engineering Ph. D. was on board. During its descent, about 16 minutes from landing, the spacecraft temperature rose to dangerous levels resulting in a catastrophic end of the mission. Clearly, the rise in temperature as a space draft enters earth's atmosphere is anticipated. Space crafts have RTOS regulating the controllers to respond to such situations from developing. And yet the cooling system(s) in this case did not offer timely mitigation. Both in terms of loss of human life and the cost of mission such a failure has the highest severity of consequences. Whereas in the case of an online stock trading, or a game show, it may mean a financial loss or a missed opportunity. In the case of a dropped packet in a video streaming application it would simply mean a glitch and a perhaps a temporary drop in the picture quality. The three examples of real-time system we have given here have different levels of severity in terms of timely response. The first one has life-threatening implication; the second case refers to a missed opportunity and finally, degraded picture quality in viewing.

Associated with these are the broadly accepted categories | hard, firm and soft real-time systems.

Architecture of Real-time Systems: The basic architecture of such systems is simple. As shown in Figure 8.1, some sensors provide input from the operative environment and a computation determines the required control. Finally, an appropriate actuator is activated. However, since the consequence of failure to respond to events can be catastrophic, it is important to build in the following two features in the system.

(a) It should be a fault tolerant design.

(b) The scheduling policy must provide for pre-emptive action.

For a fault tolerant design, the strategies may include majority voting out of the faulty sensors. Systems like satellite guidance system, usually have back-up (or a hot-stand-by) system to fall back upon. This is because the cost of failure of a mission is simply too high. Designers of Airbus A-320 had pegged the figure of failure probability at lower than 10_{-10} for one hour period in flight

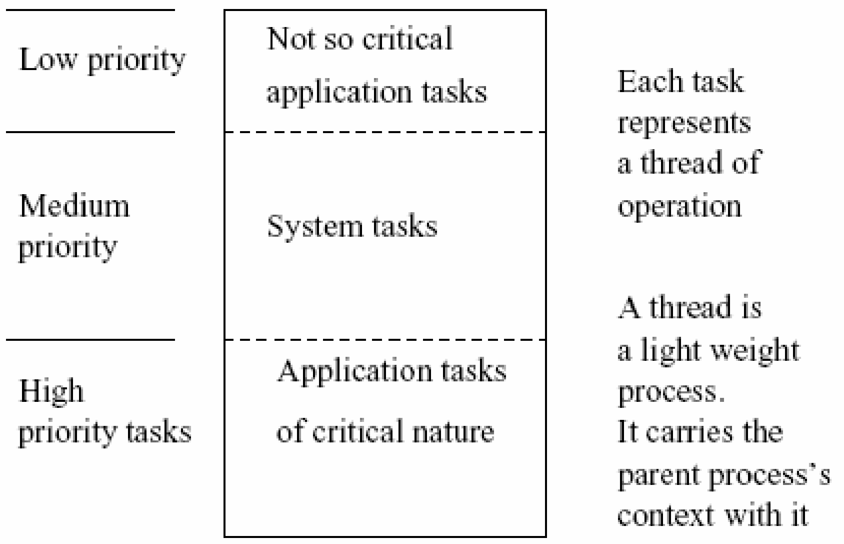

Low priority
Medium priority
High priority tasks
Not so critical application tasks
System tasks
Application tasks of critical nature
Each task represents a thread of operation
A thread is a light weight process.
It carries the parent process's context with it

Figure 8.2: Priority structure for RTOS tasks.

As for design of scheduling policy, one first identifies the critical functions and not so critical functions within an operation. The scheduling algorithm ensures that the critical functions obtain high priority interrupts to elicit immediate responses. In Figure 8.2, we depict the priority structure for such a design.

A very detailed discussion on design of real-time systems is beyond the scope of this book. Yet, it is worth mentioning here that RTOS designers have two basic design orientations to consider. One is to think in terms of event-triggered operations and the other is to think of time-triggered operations. These considerations also determine its scheduling policy. The report prepared by Panzierri and his colleagues compares architectures based on these two considerations. The observation is that time-triggered architectures obtain greater predictability but end up wasting more resource cycles of operation due to more frequent pre-emptions. On the other hand, event-triggered system architectures seem to score in terms of their ability to adapt to a variety of operating scenarios. Event-triggered systems are generally better suited for asynchronous input events. The time-triggered systems are better suited for systems with periodic inputs. For now, let us examine micro-kernels which are at the heart of RTOS, event-triggered or time-triggered.

1.4 Microkernels and RTOS

As stated earlier, micro-kernels are kernels with bare-bone, minimal essentials. To understand the notion of “bare-bone minimal essentials", we shall proceed bottom up. Also, we shall take an embedded system design viewpoint.

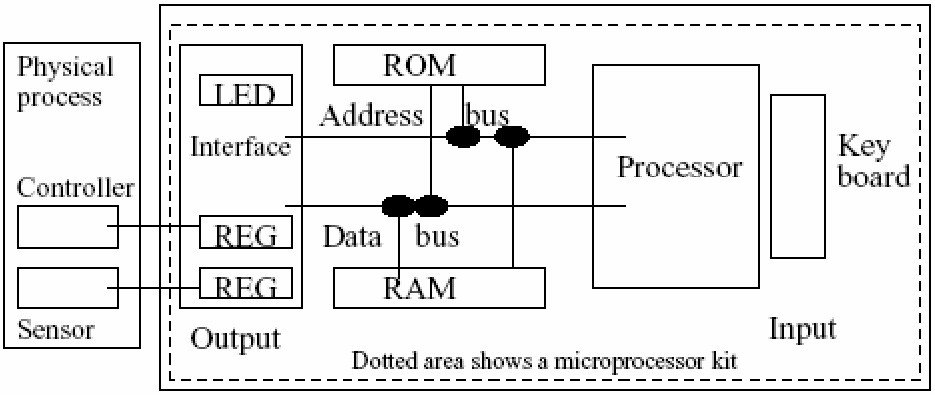


Figure 8.3: Embedded system architecture.

Let us first consider a microprocessor kit. The kit is shown in figure 8.3 within the dotted area. We can program the kit in machine language. The program can be directly stored in memory. On execution we observe output on LEDs. We also have some attached ROM. To simulate the operation of an embedded system input can be read from sensors and we can output to an interface to activate an actuator. We can use the timers and use these to periodically monitor a process. One can demonstrate the operation of an elevator control or a washing machine with these kits. We just write one program and may even do single steps through this program. Here there is no need to have an operating system. There is only one resident program.

Next, we move to an added level of complexity in interfaces. For an embedded system, input and output characterizations are very crucial. Many of the controls in embedded systems require a real-time clock. The need for real-time clocks arises from the requirement to periodically monitor (or regulate) process health. Also, abnormal state of any critical process variable needs to be detected. The timers, as also abnormal values of process state variables, generate interrupt. An example of process monitoring is shown in figure 8.4. As for the operational scenario, note that a serial controller is connected to two serial ports. Both the serial ports may be sending data. The system needs to regulate this traffic through the controller. For instance, in our example, regulating the operations of the serial controller is itself a task. In general, there may be more than one task each with its own priority level to initiate an interrupt, or there is a timer to interrupt. Essentially, one or more interrupts may happen. Interrupts require two actions. One to store away the context of the running process. The other is to switch to an interrupt service routine (ISR). The ROM may store ISRs. Before switching to an ISR, the context (the status of the present program) can be temporarily stored in RAM. All these requirements translate to management of multiple tasks with their own priorities. And that establishes the need for an embedded operating system.

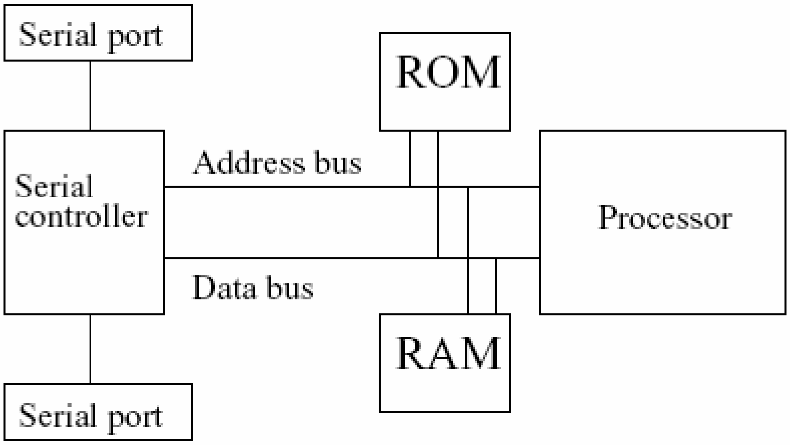


Figure 8.4: Embedded system example.

In addition, to fully meet control requirements, the present level of technology supports on-chip peripherals. Also, there may be more than one timer. Multiple timers enable monitoring multiple activities, each with a different period. There are also a number of ports to support inputs from multiple sensors and outputs to multiple controllers. All this because: a process in which this system is embedded usually has several periodic measurements to be made and several levels of priority of operations. Embedded systems may be even internet enabled. For instance, hand-held devices discussed in Section 1.4.1 are usually net enabled.

In Figure 8.5 we show a software view. The software view is that the device drivers are closely and directly tied to the peripherals. This ensures timely IO required by various tasks. The context of the applications define the tasks. Typically, the IO may be using polling or an interrupt based IO. The IO may also be memory mapped. If it is memory mapped then the memory space is adequately allocated to offer IO mappings. Briefly, then the embedded system OS designers shall pay attention to the device drivers and scheduling of tasks based on interrupt priority. The device driver functions in this context are the following.

- Do the initialization when started. May need to store an initial value in a register.
- Move the data from the device to the system. This is the most often performed task by the device driver.
- Bring the hardware to a safe state, if required. This may be needed when a recovery is required or the system needs to be reset.
- Respond to interrupt service routine. The interrupt service routine may need some status information. A typical embedded system OS is organized as a minimal system. essentially, it is a system which has a microkernel at the core which is duly supported by a library of system call functions. The microkernel together with this library is capable of the following.
- Identify and create a task.
- Resource allocation and reallocation amongst tasks.
- Delete a task.
- Identify task state like running, ready-to-run, blocked for IO etc.
- To support task operations (launch, block, read-port, run etc.), i.e. should facilitate low level message passing (or signals communication).
- Memory management functions (allocation and de-allocation to processes).

- Support preemptive scheduling policy.
- Should have support to handle priority inversion3.

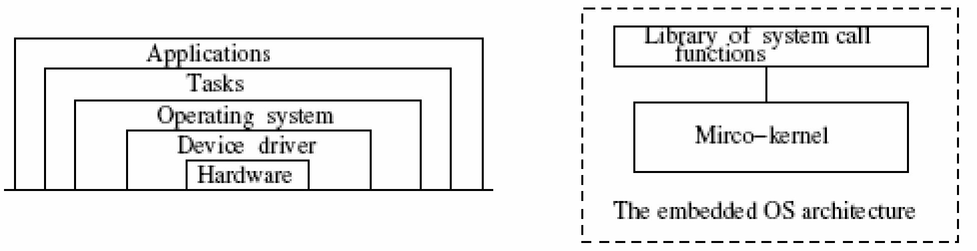


Figure 8.5: Software view of microkernel based OS.

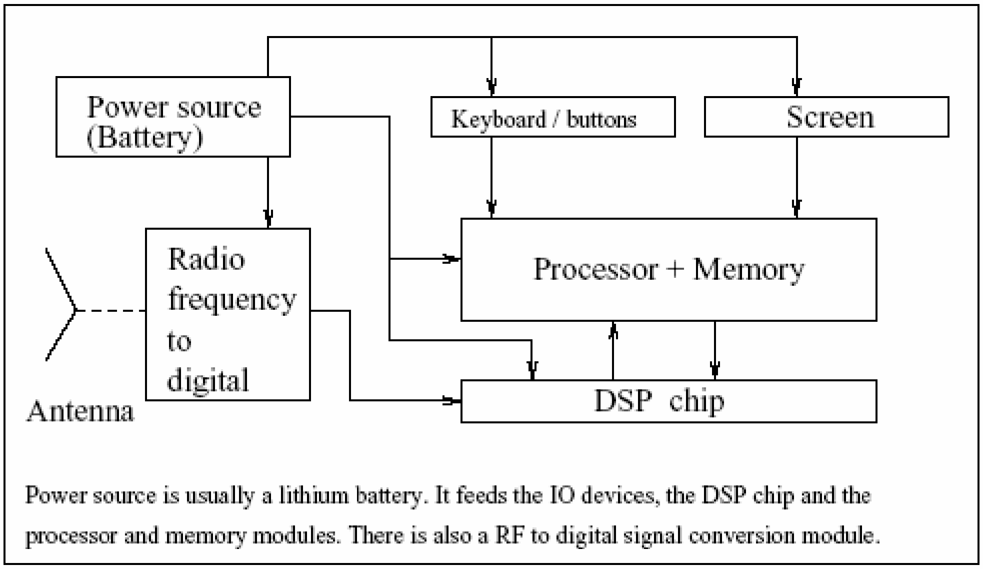
Let us elaborate on some of these points. The allocation and de-allocation of main memory in a microkernel requires that there is a main memory management system. Also, the fact that we can schedule the operation of tasks means that it is essential to have a loader as a part of a microkernel. Usually, the micro-kernels are designed with a system call \functions" library. These calls support, creation of a task, loading of a task, and communication via ports. Also, there may be a need to either suspend or kill tasks. When tasks are de-allocated, a resource reallocation may happen. This requires support for semaphores. Also, note that a support for critical section management is needed. With semaphores this can be provided for as well. In case the system operates in a distributed environment (tele-metered or internet environment), then a network support is also required. Here again, the support could be minimal so as to be able to communicate via a port. Usually in such systems, the ports are used as “mailboxes". Finally, hardware dependent features are supported via system calls.

When a task is created (or loaded), the task parameters in the system calls, include the size (in terms of main memory required), priority of the task, point of entry for task and a few other parameters to indicate the resources, ownership, or access rights. The microkernel needs to maintain some information about tasks like the state information of each task. This again is very minimal information like, running, runnable, blocked, etc. For periodic tasks we need to support the clock-based interrupt mechanism. We also have to support multiple interrupt levels. There are many advantages of a microkernel-based OS design. In particular, a microkernel affords portability. In fact, Carsten Ditze [10],argues that microkernel can be designed with minimal hardware dependence. The user services can be offered as set of library of system calls or utilities. In fact, Carsten advocates configuration management by suitably tailoring the library

functions to meet the requirements of a real-time system. In brief, there are two critical factors in the microkernel design. One concerns the way we may handle nested interrupts with priority.
The other concerns the way we may take care of scheduling. We studied interrupts in detail in the chapter on IO. Here, we focus on the consideration in the design of schedulers for real-time systems. An embedded OS veers around the device drivers and a microkernel with a library of system calls which supports real-time operations. One category of embedded systems are the hand-held devices. Next, we shall see the nature of operations of hand-held devices.

1.4.1 OS for Hand-held Devices

The first noticeable characteristic of hand-held devices is their size. Hand-held devices can be carried in person; this means that these device offer mobility. Mobile devices may be put in the category of phones, pagers, and personal digital assistants (PDAs). Each of these devices have evolved from different needs and in different ways. Mobile phones came about as the people became more mobile themselves. There was this felt need to extend the capabilities of land-line communication. Clearly, the way was to go over air and support common phone capabilities for voice. The mobile phone today supports text via SMS and even offers internet connectivity. The pagers are ideal for text transmission and PDAs offer many of the personal productivity tools. Now a days there are devices which capture one or more of these capabilities in various combinations. In Figure 8.6 we depict architecture of a typical hand-held device. Typically, the hand-held devices have the following hard-ware base.



Power source is usually a lithium battery. It feeds the IO devices, the DSP chip and the processor and memory modules. There is also a RF to digital signal conversion module.

Figure 8.6: A typical hand-held device architecture.

Microprocessor * Memory (persistent + volatile) * RF communication capability * IO units (keys + buttons / small screen (LCD)) * Power source (battery) Today's enabling technologies offer sophisticated add-ons. For instance, the DSP (digital signal processing) chips allow MP3 players and digital cameras to be attached with PDAs and mobiles. This means that there are embedded and real-time applications being developed for hand-held devices. The signal processing capabilities are easily several MIPS (million instructions per second) and beyond. The IO may also allow use of stylus and touch screen capabilities. Now let us look at some of the design concerns for OS on hand-held devices. One of the main considerations in hand-held devices is to be able to operate while conserving power. Even though the lithium batteries are rechargeable, these batteries drain in about 2 hours time. Another consideration is the flexibility in terms of IO.

These devices should be able to communicate using serial ports (or USB ports), infrared ports as well as modems. The OS should be able to service file transfer protocols. Also, the OS should have a small footprint, typically about 100K bytes with plug-in modules. Other design requirements include very low boot time and robustness. Another important facet of the operations is that hand-held devices hold a large amount of personal and enterprise information. This requires that the OS should have some minimal individual authentication before giving access to the device.

In some of the OSs, a memory management unit (MMU) is used to offer virtual memory operation. The MMU also determines if the data is in RAM. The usual architecture is microkernel supported by a library of functions just as we described in Section 1.4. An embedded Linux or similar capability OS is used in these devices. Microsoft too has some offerings around the Windows CE kernel.

The main consideration in scheduling for real-time systems is the associated predictability of response. To ensure the predictability of response, real-time systems resort to pre-emptive policies. This ensures that an event receives its due attention. So, one basic premise in real-time systems is that the scheduling must permit pre-emption. The obvious price is: throughput. The predictability requirement is regardless of the nature of input from the environment, which may be synchronous or asynchronous. Also, note that predictability does not require that the inputs be strictly periodic in nature (it does not rule out that case though). Predictability does tolerate some known extent of variability. The required adjustment to the variability is akin to the task of a wicketkeeper in the game of cricket. A bowler brings in certain variabilities in his bowling. The wicket-keeper is generally aware of the nature of variability the bowler beguiles. The wicket-keeper operates like a real-time system, where the input generator is the bowler, who has the freedom to choose his ball line, length, and flight. Clearly, one has to bear the worst case in mind. In real-time systems too, the predictable situations require to cater for the worst case schedulability of tasks. Let us first understand this concept. Schedulability of a task is influenced by all the higher priority tasks that are awaiting scheduling. We can explain this as follows. Suppose we identify our current task as tc and the other higher priority tasks as t1; :::; tn where ti identifies the i-th task having priority higher than tc. Now let us sum up the upper bounds for time of completion for all the higher priority tasks t1; :::; tn and to it add the time required for tc. If the total time is less than the period by which task tc must be completed, then we say that task tc meets the worst case schedulability consideration. Note that schedulability ensures predictability and offers an upper bound on acceptable time of completion for task tc. Above, we have emphasized pre-emption and predictability for real-time systems. We shall next examine some popular scheduling policies. The following three major scheduling policies are quite popular.

(a) Rate monotonic (or RM for short) scheduling policy.

(b) The earliest deadline first (or EDF for short) scheduling policy.

(c) The least laxity first (or LLF) scheduling policy.

We describe these policies in Sections 1.4.2, 1.4.3 and 1.4.4. A curious reader may wonder if the predictability (in terms of schedulability for timely response) could be guaranteed under all conditions. In fact, predictability does get affected when a lower priority task holds a mutually shared resource and blocks a higher priority task. This is termed as a case of priority inversion. The phenomenon of priority inversion may happen both under RM and EDF schedules. We shall discuss the priority inversion in section 1.4.5. We shall also describe strategies to overcome this problem.

1.4.2 Rate Monotonic Scheduling

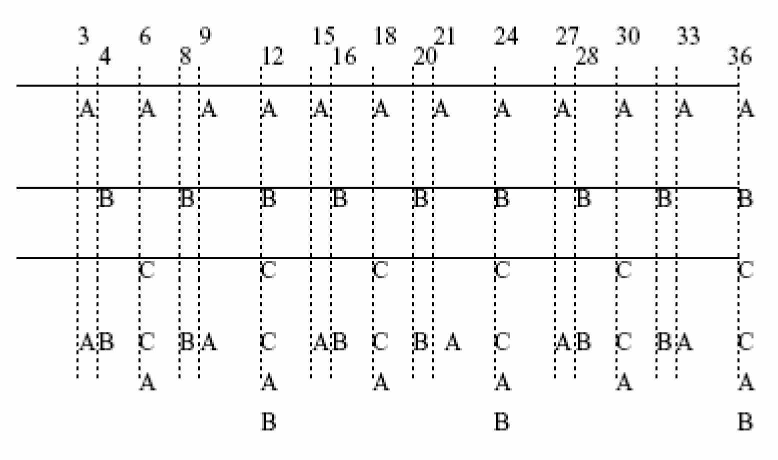


Figure 8.7: Rate monotonic scheduling for tasks.

Some real-time systems have tasks that require periodic monitoring and regulation. So the events are cyclic in nature. This cyclicity requires predictable event detection and consequent decision on control settings. For this category of real-time systems the popular scheduling strategy is rate monotonic scheduling. Let us now examine it in some detail. The rate monotonic scheduling stipulates that all the tasks are known apriori. Also, known is their relative importance. This means that we know their orders of priority. Tasks with highest priority have the shortest periodicity. The tasks may be independent of each other. Armed with this information, and the known times of completion for each task, we find the least common multiple lcm of the task completion times. Let us denote the lcm as Trm. Now a schedule is drawn for the entire time period Trm such that each task satisfies the schedulability condition. The schedule so generated

is the RM schedule. This schedule is then repeated with period Trm. As an example, consider that events A;B; and C happen with time periods 3, 4, and 6, and when an event occurs the system must respond to these. Then we need to draw up a schedule as shown in Figure 8.7. Note that at times 12, 24, and 36 all the three tasks need to be attended to while at time 21 only task
A needs to be attended. This particular schedule is drawn taking its predictability into account. To that extent the RM policy ensures predictable performance. In theory, the rate monotonic scheduling is known to be an optimal policy when priorities are statically defined tasks.

1.4.3 Earliest Deadline First Policy

The EDF scheduling handles the tasks with dynamic priorities. The priorities are determined by the proximity of the deadline for task completion. Lower priorities are assigned when deadlines are further away. The highest priority is accorded to the task with the earliest deadline. Clearly, the tasks can be put in a queue like data-structure with the entries in the ascending order of the deadlines for completion. In case the inputs are periodic in nature, schedulability analysis is possible. However, in general the EDF policy cannot guarantee optimal performance for the environment with periodic inputs. Some improvement can be seen when one incorporates some variations in EDF policy. For instance, one may account for the possibility of distinguishing the mandatory tasks from those that may be optional. Clearly, the mandatory tasks obtain schedules for execution by pre-emption whenever necessary. The EDF is known to utilize the processor
better than the RM policy option.

1.4.4 Earliest Least Laxity First Policy

This is a policy option in which we try to see the slack time that may be available to start
a task. This slack time is measured as follows:
slack time = dead line ¡ remaining processing time
The slack defines the laxity of the task. This policy has more overhead in general but has been found to be very useful for multi-media applications in multiprocessor environment.

1.4.5 Priority Inversion

Often priority inversion happens when a higher priority task gets blocked due to the fact that a mutually exclusive resource R is currently being held by a lower priority task. This may happen as follows. Suppose we have three tasks t1; t2, and t3 with priorities in the order of their index with task t1 having the highest priority. Now suppose there is a shared resource R which task t3 and task t1 share. Suppose t3 has obtained resource R and it is to now execute. Priority inversion occurs with the following plausible sequence of events.

1. Resource R is free and t3 seeks to execute. It gets resource R and t3 is executing.
2. Task t3 is executing. However, before it completes task t1 seeks to execute.
3. Task t3 is suspended and task t1 begins to execute till it needs resource R. It gets suspended as the mutually exclusive resource R is not available.
4. Task t3 is resumed. However, before it completes task t2 seeks to be scheduled.
5. Task t3 is suspended and task t2 begins executing.
6. Task t2 is completed. Task t1 still cannot begin executing as resource R is not available (held by task t3).
7. Task t3 resumes to finish its execution and releases resource R.
8. Blocked task t1 now runs to completion.

The main point to be noted in the above sequence is: even though the highest priority task t1 gets scheduled using a pre-emptive strategy, it completes later than the lower priority tasks t2 and t3!! Now that is priority inversion.

How to handle priority inversion: To ensure that priority inversion does not lead to missing deadlines, the following strategy is adopted [19]. In the steps described above, the task t1 blocks when it needs resource R which is with task t3. At that stage, we raise the priority of task t3, albeit temporarily to the level of task t1. This ensures that task t2 cannot get scheduled now. Task t3 is bound to complete and release the resource R. That would enable scheduling of the task t1 before task t2. This preserves the priority order and avoids the priority inversion. Consequently, the deadlines for task t1 can be adhered to with predictability.

Block-3

Unit-1

1.1 Introduction to OS and Security

Computers, with their ubiquitous presence, have ceased to be a wonder they once were. Their usage is pervasive. Information access and delivery from, and to, a remote location via internet is common. Today many societal services like railway time-table or election results are rendered through computers. The notion of electronic commerce has given fillip to provisioning commercial services as well. Most individuals use computers to store private information at home and critical professional information at work. They also use computers to access information from other computers anywhere on the net. In this kind of scenario, information is the key resource and needs to be protected.

The OS, being the system's resource regulator, must provide for security mechanisms. It must not only secure the information to protect the privacy but also prevent misuse of system resources. Unix designers had aimed to support large-scale program development and team work. The main plank of design was flexibility and support tools. The idea was to promote creation of large programs through cooperative team efforts. All this was long before 9/11. Security has become a bigger issue now. Much of Unix provisioning of services was with the premise that there are hardly, if any, abuses of system. So, Unix leaves much to be desired in respect of security. And yet, Unix has the flexibility to augment mechanisms that primarily protect users resources like files and programs. Unix incorporates security through two mechanisms, user authentication and access control. We shall elaborate on both these aspects and study what could be adequate security measures. We begin with some known security breaches. That helps to put security measures in proper perspective.

1.2 Security Breaches

We first need to comprehend the types of security breaches that may happen. Breaches may happen with malicious intent or may be initiated by users inadvertently, or accidentally. They may end up committing a security breach through a mis-typed command or ill understood interpretation of some command. In both these instances the OS must protect the interest of legitimate users of the system. Unix also does not rule out a malicious access with the intent to abuse the system. It is well known that former disgruntled employees often attempt access to systems to inflict damages or simply corrupt some critical information. Some malicious users' actions may result in one of the following three kinds of security breaches:

1. Disclosure of information.
2. Compromising integrity of data.
3. Denial of service to legitimate users of the system.

To launch an attack, an attacker may correctly guess a weak password of a legitimate user. He can then access the machine and all HW and SW resources made available to that user. Note that a password is an intended control (a means to authenticate a user) to permit legitimate access to system resources. Clearly, a malicious user may employ password racking methods with the explicit intent to bypass the intended controls. He may access classified information and may also misuse the system resources. An un authorized access ay be launched to steal precious processor cycles resulting in denial of service. Or, he may be able to acquire privileged access to modify critical files corrupting sensitive data. This would be an act of active misuse. Some activities like watching the traffic on a system or browsing without modifying files may be regarded as an act of passive misuse. Even this is a breach of security as it does lead to disclosure. It may result in some deterioration, albeit not noticeable, in the overall services as well.

1.2.1 Examples of Security Breaches

Here we shall discuss a few well known attacks that have happened and have been recorded. Study of these examples helps us to understand how security holes get created. Besides, it helps us to determine strategies to plug security holes as they manifest. Next we describe a few attack scenarios. Not all of these scenarios can be handled by OS control mechanisms. Nonetheless, it is very revealing to see how the attacks happen.

- **External Masquerading:** This is the case of unauthorized access. The access may be via a communication media tap, recording and playback. For instance, a login session may be played back to masquerade another user. The measures require a network-based security solution.
- **Pest Programs:** A malicious user may use a pest program to cause a subsequent harm. Its effect may manifest at some specified time or event. The Trojan horse and virus attacks fall in this category. The main difference between a Trojan horse and a virus is

that, a virus is a self reproducing program. Some virus writers have used the Terminate and Stay Resident (TSR) program facility in Micro-soft environments to launch such attacks. The pest programs require internal controls to counter. Generally, the time lag helps the attacker to cover the tracks. Typically, a virus propagation involves the following steps:

- **Remote copy:** In this step a program is copied to a remote machine.
- **Remote execute:** The copied program is instructed to execute. The step requires repeating the previous step on the other connected machine, thereby propagating the virus.
- **Bypassing internal controls:** This is achieved usually by cracking passwords, or using compiler generated attack to hog or deny resources.
- **Use a given facility for a different purpose:** This form of attack involves use of a given facility for a purpose other than it was intended for. For example, in Unix we can list files in any directory. This can be used to communicate secret information without being detected. Suppose `userB' is not permitted to communicate or access files of `userA'. When `userB' access files of `userA' he will always get a message permission denied. However, `userA' may name his files as atnine, tonight, wemeet. When `userB' lists the files in the directory of `userA' he gets the message “at nine tonight we meet", thereby defeating the access controls.
- **Active authority misuse:** This happens when an administrator (or an individual) abuses his user privileges. A user may misuse the resources advanced to him in good faith and trust. An administrator may falsify book keeping data or a user may manipulate accounts data or some unauthorized person may be granted an access to sensitive information.
- **Abuse through inaction:** An administrator may choose to be sloppy (as he may be disgruntled) in his duties and that can result in degraded services.
- **Indirect abuse:** This does not quite appear like an attack and yet it may be. For instance, one may work on machine `A' to crack a protection key on machine `B'. It may appear as a perfectly legal study on machine `A' while the intent is to break the machine `B' internal controls.

We next discuss the commonly used methods of attacks. It is recommended to try a few of these in off-line mode. With that no damage to the operating environment occurs nor is the operation of an organization affected.

- **The Password spoof program:** We consider the following Trojan horse and the effect it generates. It is written in a Unix like command language.

  ```
  B1='ORIGIN: NODE whdl MODULE 66 PORT 12'
  B2='DESTINATION:'
  FILE=$HOME/CRYPT/SPOOFS/TEST
  trap '' 1 2 3 5 15
  echo $B1
  sleep 1
  echo ''
  echo $B2
  read dest
  echo 'login:
  read login
  stty -echo
  echo 'password:
  read password
  stty echo
  echo ''

  echo $login $passwd >> spooffile
  echo 'login incorrect'
  exec login
  ```

  The idea is quite simple. The program on execution leaves a login prompt on the terminal. To an unsuspecting user it seems the terminal is available for use. A user would login and his login session with password shall be simply copied on to spooffile. The attacker can later retrieve the login name and password from the spooffile and now impersonate the user.

- **Password theft by clever reasoning:** In the early days passwords in Unix systems were stored in an encrypted form under /etc/password. The current practice of using a shadow file will be discussed later. So, in early days, the safety of password lay in the difficulty associated with decrypting just this file. So attackers used to resort to a clever way of detecting passwords. One such attack was through an attempt to match commonly used mnemonics, or use of convenient word patterns. Usually, these are words that are easy to type or recall. The attacker generated these and used the encrypting function to encrypt them. Once the encrypted pattern matched, the corresponding password was compromised.

- **Logic Bomb:** A logic bomb is usually a set-up like the login spoof described earlier. The attacker sets it up to go off when some conditions combine to happen. It may be long after the attacker (a disgruntled employee for instance) has quit the organization. This may leave no trail. Suppose we use an editor that allows setting of parameters to OS shell, the command interpreter. Now suppose one sets up a Unix command rm *.* and puts it in a file called EditMe and sends it over to the system administrator. If the system administrator opens the file and tries to edit the file, it may actually remove all the files unless he opens it in a secure environment.

  Also, if the administrator attempts opening this as a user, damage would be less, compared to when he opens it as a root.

- **Scheduled File Removal:** One of the facilities available on most OSs is scheduled execution of a program or a shell script. Under Unix this is done by using at command. A simple command like : *rm -f /usr* at 0400 saturday attack This can result in havoc. The program may be kept in a write protected directory and then executed at some specified time. The program recursively removes files without diagnostic messages from all users under *usr.*

- **Field Separator Attack:** The attack utilizes some OS features. The following steps describe the attack :

  1. The attacker redefines the field separator to include backslash character so that path names such as /coo/koo are indistinguishable from coo koo.
  2. The attacker knowing that some system program, say *sysprog,* uses administrative privilege to open a file called /coo/koo creates a program coo and places it in an accessible directory. The program is coded to transfer privileges from the system to the user via a copied OS shell.
  3. The attacker invokes *sysprog* which will try to open /coo/koo with the administrative privileges but will actually open the file coo since the field separator has been redefined. This will have the desired effect of transferring privileges to the user, just as the attacker intended.

  - **Insertion of Compiler Trojan Horse:** To launch an attack with a very widely istributed effect an attacker may choose a popular filtering program based Trojan horse. A compiler is a good candidate for such an attack. To understand an attack via a compiler Trojan horse, let us first describe how a compiler works:

```
Compile: get (line);
Translate (line);
```

A real compiler is usually more complex than the above description. Even this models a lexical analysis followed by the translating phases in a compiler. The objective of the Trojan horse would be to look for some patterns in the input programs and replace these with some trap door that will allow the attacker to attack the system at a later time. Thus the operation gets modified to:

```
Compile : get (line);
if line == "readpwd(p)" then translate (Trojan horse insertion)
else
        translate (line);
```

- **The Race Condition Attack:** A race condition occurs when two or more operations occur in an undefined manner. Specifically, the attacker attempts to change the state of the file system between two file system operations. Usually, the program expects these two operations to apply to the same file, or expects the information retrieved to be the same. If the file operations are not atomic, or do not reference the same file this cannot be guaranteed without proper care.

  In Solaris 2.x's ps utility had a security hole that was caused by a race condition. The utility would open a temporary file, and then use the *chown()* system call with the file's full path to change its ownership to root. This sequence of events was easily exploitable. All that an attacker now had to do was to first slow down the system, and find the file so created, delete it, and then slip in a new SUID world writable file. Once the new file was created with that mode and with the ownership changed by *chown* to root by the insecure process, the attacker simply copies a shell into the file. The attacker gets a root shell.

  The problem was that the second operation used the file name and not the file descriptor. If a call to *fchown*() would have been used on the file descriptor returned from the original *open*() operation, the security hole would have been avoided. File names are not unique. The file name /tmp/foo is really just an entry in the directory /tmp. Directories are special files. If an attacker can create, and delete files from a directory the program cannot trust file names taken from it. Or, to look at it in a more critical way, because the directory is modifiable by the attacker, a program cannot trust it as a source of valid input. Instead it should use file descriptors to perform its operations. One solution is to use the sticky bit (see Aside). This will prevent the attacker from removing the file, but not prevent the attacker from creating files in the directory. See below for a treatment of symbolic link attacks.

An Aside: Only directories can have sticky bit set. When a directory has the sticky bit turned on, anyone with the write permission can write (create a file) to the directory, but he cannot delete a file created by other users.

- **The Symlink Attack:** A security hole reported for SUN's license manager stemmed from the creation of a file without checking for symbolic links (or soft links). An *open*() call was made to either create the file if it did not exist, or open it if it did exist. The problem with a symbolic link is that an open call will follow it and not consider the link to constitute a created file. So if one had /tmp/foo. symlinked to /.rhosts or "/root/.rhosts), the latter file would be transparently opened. The license manager seemed to have used the O_CREAT flag with the open call making it create the file if it did not exist. To make matters worse, it created the file with world writable permissions. Since it ran as root, the .rhosts file could be created, written to, and root privileges attained.

1.3 Attack Prevention Methods

Attack prevention may be attempted at several levels. These include individual screening and physical controls in operations. Individual screening would require that users are screened to authenticate themselves and be responsible individuals. Physical controls involve use of physical access control. Finally, there are methods that may require configuration controls. We shall begin the discussion on the basic attack prevention with the defenses that are built in Unix. These measures are directed at user authentication and file access control.

1.3.1 User Authentication

First let us consider how a legitimate user establishes his identity to the system to access permitted resources. This is achieved typically by username/password pair. When the system finishes booting, the user is prompted for a username and then a password in succession. The password typed is not echoed to the screen for obvious reasons. Once the password is verified the user is given an interactive shell from where he can start issuing commands to the system. Clearly, choosing a clever password is important. Too simple a password would be an easy give away and too complex would be hard to remember. So how can we choose a nice password?

Choosing a good password: A malicious user usually aims at obtaining a complete control over the system. For this he must acquire the superuser or root status. Usually, he attacks vulnerable points like using badly installed software, bugs in some system software or human errors. There are several ways to hack a computer, but most ways require extensive knowledge. A relatively easier way is to log in as a normal user and search the system for bugs to become superuser. To do this, the attacker will have to have a valid user code and password combination to start with. Therefore, it is of utmost importance that all users on a system choose a password which is quite difficult to guess. The security of each individual user is closely related to the security of the whole system. Users often have no idea how a multi-user system works and do not realize that by choosing an easy to remember password, they indirectly make it possible for an attacker to manipulate the entire system. It is essential to educate the users well to avoid lackadaisical attitudes. For instance, if some one uses a certain facility for printing or reading some mails only, he may think that security is unimportant. The problem arises when someone assumes his identity. Therefore, the users should feel involved with the security of the system. It also means that it is important to notify the users of the security guidelines. Or at least make them understand why good passwords are essential.

Picking good passwords: We will look at some methods for choosing good passwords. A typical good password may consist up to eight characters. This means passwords like `members only' and `members and guests' may be mutually inter-changeable. A password should be hard to guess but easy to remember. If a password is not easy to remember then users will be tempted to write down their password on yellow stickers which makes it futile. So, it is recommended that a password should not only have upper or lowercase alphabets, but also has a few non-alphanumeric characters in it. The non-alphanumeric characters may be like (%,,*, =) etc. The use of control characters is possible, but not all control characters can be used, as that can create problems with some networking protocols. We next describe a few simple methods to generate good passwords.

- Concatenate two words that together consist of seven characters and that have no connection to each other. Concatenate them with a punctuation mark in the middle and convert some characters to uppercase like in 'abLe+pIG'.
- Use the first characters of the words of not too common a sentence. From the sentence “My pet writers are Wodehouse and Ustinov", as an example, we can create password “MpwaW+U!”. Note in this case we have an eight-character password with uppercase characters as well as punctuation marks.
- Alternatively, pick a consonant and one or two vowels resulting in a pronounceable (and therefore easy to remember) word like 'koDuPaNi'.

This username/password information is kept traditionally in the /etc/passwd file, commonly referred to simply as the password file. A typical entry in the password file is shown below:

user:x:504:504::/home/user:/bin/bash

There are nine colon separated fields in the above line. They respectively refer to the user name, password x (explained later), UID, GID, the GECOS field 1, home directory and users' default shell. In the early implementations of the Unix, the password information was kept in the passwd file in plain text. The passwd file has to be world readable as many programs require to authenticate themselves against this file. As the expected trust level enhanced, it became imperative to encrypt the password as well. So, the password field is stored in an encrypted format. Initially, the crypt function was used extensively to do this. As the speed of the machines increased the encrypted passwords were rendered useless by the brute force techniques. All a potential attacker needed to do is to get the passwd file and then do a dictionary match of the encrypted password. This has led to another innovation in the form of the shadow suite of programs. In modern systems compatible with the shadow suite the password information is now kept in the /etc/shadow file and the password field in the passwd file is filled with an x (as indicated above). The actual encrypted password is kept in the /etc/shadow file in the following format:

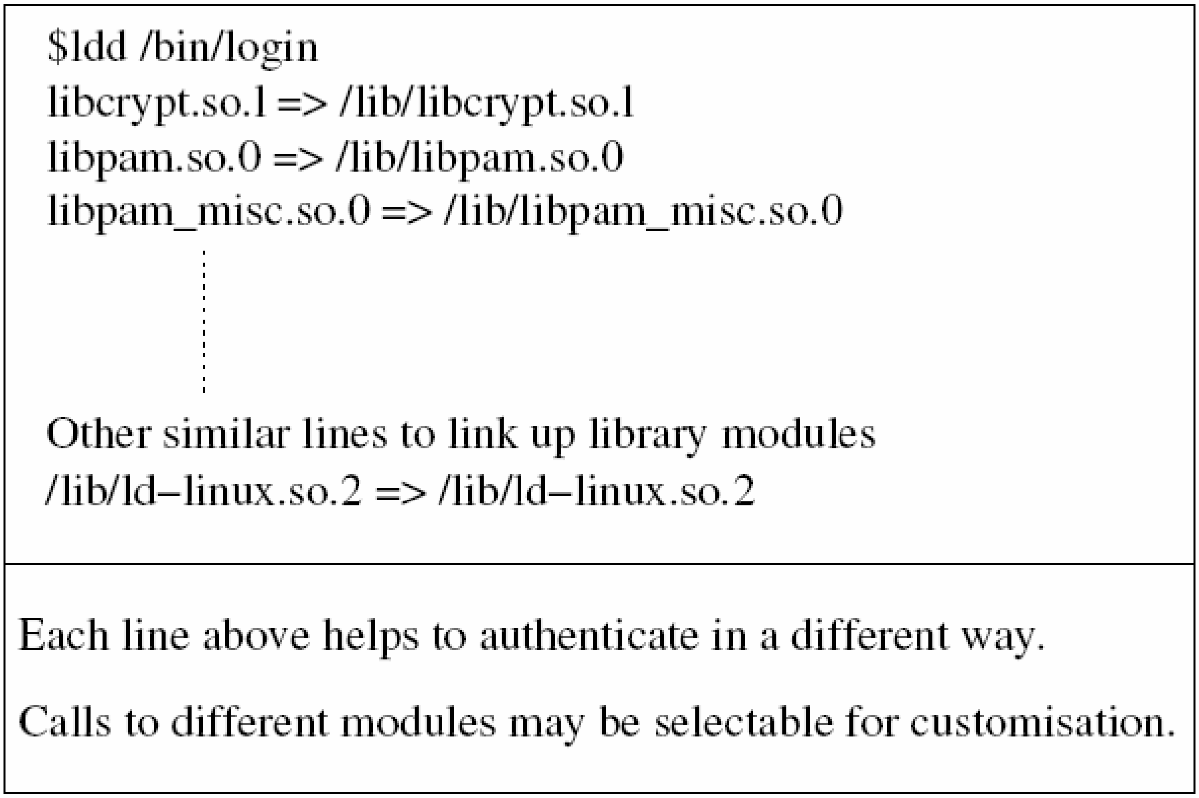
user:$1$UaV6PunD$vpZUg1REKpHrtJrVi12HP.:11781:0:99999:7:::

The second field here is the password in an md5 hash 2. The other fields relate to special features which the shadow suite offers. It offers facilities like aging of the passwords, enforcing the length of the passwords etc. The largest downside to using the shadow passwords is the difficulty of modifying all of the programs that require passwords from the appropriate file to use

/etc/shadow instead. Implementing other new security mechanisms presents the same difficulty. It would be ideal if all of these programs used a common framework for authentication and other security related measures, such as checking for weak passwords and printing the message of the day.

Pluggable authentication modules Red Hat and Debian Linux distributions ship with “Pluggable Authentication Modules" (PAM for short) and PAM-aware applications. PAM offers a flexible framework which may be customized as well. The basic PAMbased security model is shown in Figure 9.1. Essentially, the figure shows that one may have multiple levels of authentication, each invoked by a separate library module. PAM aware applications use these library modules to authenticate. Using PAM modules, the administrator can control exactly how authentication may proceed upon login. Such authentications go beyond the traditional /etc/passwd file checks. For instance, a certain application may require the pass-word as well as a form of bio-metric authentication. The basic strategy is to incorporate a file (usually called /etc/pam.d/login) which initiates a series of authentication checks for every login attempt. This file ensures that a certain authentication check sequence is observed. Technically, the library modules may be selectable. These selections may depend upon the severity of the authentication required. The administrator can customize the needed choices in the script. At the next level, we may even have an object based security model in which every object access would require authentication for access, as well as methods invocation.

For now we shall examine some typical security policies and how Unix translates security policies in terms of access control mechanisms.



```
$ldd /bin/login
libcrypt.so.1 => /lib/libcrypt.so.1
libpam.so.0 => /lib/libpam.so.0
libpam_misc.so.0 => /lib/libpam_misc.so.0

Other similar lines to link up library modules
/lib/ld–linux.so.2 => /lib/ld–linux.so.2
```

Each line above helps to authenticate in a different way.

Calls to different modules may be selectable for customisation.

Figure 9.1: Pluggable authentication.

1.3.2 Security Policy and Access Control

Most access control mechanisms emanate from a stated security policy. It is important to learn to design a security policy and offer suitable access mechanisms that can support security policies. Security policy models have evolved from many real-life operating scenarios. For instance, if we were to follow a regime of defense forces, we may resort to a hierarchy based policy. In such a policy, the access to resources shall be determined by associating ranks with users. This requires a security-related labeling on information to permit access. The access is regulated by examining the rank of the user in relation to the security label of the information being sought. For a more detailed discussion the reader may refer where there is a discussion on how to specify security policies as well.

If we were to model the security policies based on commercial and business practices or the financial services model, then data integrity would take a very high precedence. This like, the accounts and audit practices, preserves the integrity of data at all times. In practice, however, we may have to let the access be governed by ownership (who own the information) and role definitions of the users. For instance, in an organization, an individual user may own some information but some critical information may be owned by the institution. Also, its integrity should be impregnable. And yet the role of a system manager may require that he has access privileges which may allow him a free reign in running the system smoothly.

Almost all OSs provide for creating system logs of usage. These logs are extremely useful in the design of Intrusion Detection Systems (IDS). The idea is quite simple. All usages of resources are tracked by the OS and recorded. On analysis of the recorded logs it is possible to determine if there has been any misuse. The IDS helps to detect if a breach has occurred. Often this is after the event has taken place. To that extent the IDS provides a lot of input in designing security tools. With IDS in place one can trace how the attack happened. One can prevent attacks from happening in future. A full study and implementation of IDS is beyond the scope of this book. We would refer the reader to Amoroso's recent book on the subject.

Defenses in Unix: Defenses in Unix are built around the access control 3. Unix's access control is implemented through its file system. Each file (or directory) has a number of attributes, including a file name, permission bits, a UID and a GID. The UID of a file specifies its owner. In Chapter 2, we had explained that the permission bits are used to specify permissions to read (r), write (w), and execute (x). These permissions are associated with every file of every user, for the members of the user's group, and for all other users of that system. For instance, the permission string rwxr-x--x specifies that the owner may read, write and execute, the user's group members are allowed to read and execute it, while all the other users of the system may be permitted to only execute this file. A dash (`-') in the permission set indicates that the access rights are not permitted. Furthermore, each process in Unix has an effective and a real UID as well as an effective and a real GID associated with it. The real UID (and GID) are the primary identifications that Unix systems continually maintain based on the identifications assigned at the time of accounts creation. However, access rights and privileges evolve over time. The effective identifications precisely reflect that. A process's effective identification indicates the access privileges. Whenever a process attempts to access a file, the kernel will use the process's effective UID and GID to compare them with the UID and the GID associated with the file to decide whether or not to grant the request.

As we stated earlier, Unix logs the systems' usage. Unix kernel, and system processes, store pertinent information in the log files. The logs may be kept either locally, or centrally, on an network server. Sometimes logs are prepared for a fixed duration of time (like for 1 to 30 days) or archived. The logs may be analyzed on-line or off-line on a secured isolated system. An analysis on a secured isolated system has the advantage that it cannot be modified by an attacker (to erase his trace). Also, the analysis can be very detailed as this is the only purpose of such a system.

With the security concerns coming into focus, security standards have emerged. Usually the security standards recommend achieving minimal assured levels of security through some form of configuration management. Most OSs, Unix included, permit a degree of flexibility in operations by appropriately configuring the system resources. In addition, modern Unix systems support a fairly comprehensive type of auditing known as C2 audit. This is so named because it fulfils the audit requirements for the TCSEC C2 security level.

Networking concerns: Realistically speaking almost all machines are networked. In any case every machine has built-in network (NW) support. The default NW support is TCP/IP or its variant. This is very assuring from the point of compatibility. The range of NW services support includes remote terminal access and remote command execution using *rsh, rlogin* commands and remote file transfer using *ftp* command. The remote service soliciting commands are collectively known as the r commands. The NW File System (NFS) is designed to offer transparency to determine the location of the file. This is done by supporting mounting of a remote file as if it was on the local file system. In fact, NFS technically supports multiple hosts to share files over a local area network (LAN). The Network Information System (NIS), formally known as the Sun Yellow Pages, enables hosts to share systems and NW databases. The NW databases contain data concerning user account information, group membership, mail aliases etc. The NFS facilitates centralized administration of the file system. Basically, the r commands are not secure. There are many reasons why these are insecure operations. We delineate some of these below.

- The primary one being that Unix was designed to facilitate usage with a view to cooperating in flexible ways. The initial design did not visualize a climate of suspicion. So, they assumed that all hosts in the network are trusted to play by the rules, e.g. any request arising out of a TCP/IP port below 1024 is considered to be trusted.
- These commands require a simple address-based authentication, i.e. the source address of a request is used to decide whether or not to grant an access or offer a service.
- They send clear text passwords over the network.

Now a days there are other better alternatives to the r commands, namely *ssh, slogin* and *scp*, respectively, which use strong *ssl* public key infrastructure to encrypt their traffic. Before an NFS client can access files on a file system exported by an NFS server, it needs to mount the file system. If a mount operation succeeds, the server will respond with a file handle, which is later used in all accesses to that file system in order to verify that the request is coming from a legitimate client. Only clients that are trusted by the server are allowed to mount a file system. The primary problem with NFS is the weak authentication of the mount request. Usually the authentication is based on IP address of the client machine. Note that it is not difficult to fake an IP address!!. So, one may configure NIS to operate with an added security protocol as described below.

- Ensure minimally traditional Unix authentication based on machine identification and UID.
- Augment data encryption standard (DES) authentication.

The DES authentication provides quite strong security. The authentication based on machine identification or UID is used by default while using NFS. Yet another authentication method based on Kerberos is also supported by NIS. The servers as well as the clients are sensitive to attacks, but some are of the opinion that the real security problem with NIS lies in the client side. It is easy for an intruder to fake a reply from the NIS server. There are more secure replacements for the NIS as well (like LDAP), and other directory services.

Unix security mechanisms rely heavily on its access control mechanisms. We shall study the access control in more detail a little later. However, before we do that within the broad framework of network concerns we shall briefly indicate what roles the regulatory agencies play. This is because NWs are seen as a basic infrastructure.

Internet security concerns and role of security agencies: In USA, a federally funded Computer Emergency Response Team (CERT) continuously monitors the types of attacks that happen. On its site it offers a lot of advisory information. It even helps organizations whose systems may be under attack. Also, there is critical infrastructure protection board within whose mandate it is to protect internet from attack. The National Security Agency (NSA) acts as a watchdog body and influences such decisions as to what level of security products may be shipped out of USA. The NSA is also responsible to recommend acceptable security protocols and standards in USA. NSA is the major security research agency in USA. For instance, it was

NSA that made the recommendation on product export restriction beyond a certain level of DES security (in terms of number of bits).

In India too, we have a board that regulates IT infrastructure security. For instance, it has identified the nature of Public Key Infrastructure. Also, it has identified organizations that may offer security-related certification services. These services assure of authentication and support non-repudiation in financial and legal transactions. It has set standards for acceptable kind of digital signatures. For now let us return to the main focus of this section which is on access control. We begin with the perspective of file permissions.

File permissions: The file permissions model presents some practical difficulties. This is because Unix generally operates with none or all for group permissions. Now consider the following scenario:

There are three users with usernames Bhatt, Kulish, and Srimati and they belong to the group users. Is there anyway for Bhatt to give access to a file that he owns to Kulish alone. Unfortunately it is not possible unless Bhatt and Kulish belong to an identifiable group (and only these two must be members of that group) of which Srimati is not a member. To allow users to create their own groups and share files, there are programs like sudo which the administrator can use to give limited superuser privileges to ordinary users. But it is cumbersome to say the least. There is another option in the BSD family of Unix versions, where a user must belong to the Wheel group to run programs like *sudo* or *su*. This is where the Access Control Lists (ACLs) and the extended attributes come into picture. Since access control is a major means of securing in Unix we next discuss that. More on access control in Unix: Note that in Unix all information is finally in the form of a file. So everything in Unix is a file. All the devices are files (one notable exception being the network devices and that too for historical reasons). All data is kept in the form of files. The configuration for the servers running on the system is kept in files. Also, the authentication information itself is stored as files. So, the file system's security is the most important aspect in Unix security model. Unix provides access control of the resources using the two mechanisms:

(a) The file permissions, uid, gid.

(b) User-name and password authentication.

The file access permissions determine whether a user has access permissions to seek requested services. Username and password authentication is required to ensure that the user is who he claims to be. Now consider the following *rwx* permissions for user, group and others.

```
$ ls -l
drwxrwxr-x 3 user group 4096 Apr 12 08:03 directory
-rw-rw-r-- 1 user group 159 Apr 20 07:59 sample2e.aux
```

The first line above shows the file permissions associated with the file identified as a directory. It has read, write and execute permission for the user and his group and read and execute for others. The first letter `d' shows that it is a directory which is a file containing information about other files. In the second line the first character is empty which indicates that it is a regular file. Occasionally, one gets to see two other characters in that field. These are `s' and `l', where `s' indicates a socket and `l' indicates that the file is a link. There are two kinds of links in Unix. The hard link and the soft link (also known as symbolic links). A hard link is just an entry in the directory pointing to the same file on the hard disk. On the other hand, the symbolic link is another separate file pointing to the original file. The practical difference is that a hard link has to be on the same device as the original but the symbolic link can be on a different device. Also, if we remove a file, the hard link for the file will also be removed. In the case of a symbolic link, it will still exist pointing nowhere.

In Unix every legitimate user is given a user account which is associated with a user id (Unix only knows and understands user ids here to in referred as UIDs).The mapping of the users is maintained in the file /etc/passwd. The UID 0 is reserved. This user id is a special superuser id and is assigned to the user ROOT. The SU ROOT has unlimited privileges on the system. Only SU ROOT can create new user accounts on a system. All other UIDs and GIDs are basically equal.

A user may belong to one or more groups up to 16. A user may be enjoined to other groups or leave some groups as long as the number remains below the number permitted by the system. At anytime the user must belong to at least one group. Different flavors of Unix follow different conventions. Linux follows the convention of creating one group with the same name as the username whenever a new user id is created. BSDs follow the convention of having all the ordinary users belong to a group called users.

It is to be noted that the permissions are matched from left to right. As a consequence, the following may happen. Suppose a user owns a file and he does not have some permission. However, suppose the group (of which he also is a member) has the permission. In this situation because of the left to right matching, he still cannot have permission to operate on the file. This is more of a quirk as the user can always change the permissions whichever way he desires if he owns the file. The user of the system must be able to perform certain security critical functions on the system normally exclusive to the system administrator, without having access to the same security permissions. One way of giving users' a controlled access to a limited set of system privileges is for the system to allow the execution of a specified process by an ordinary user, with the same permissions as another user, i.e. system privileges. This specified process can then perform application level checks to insure that the process does not perform actions that the user was not intended to be able to perform. This of course places stringent requirements on the process in terms of correctness of execution, lest the user be able to circumvent the security checks, and perform arbitrary actions, with system privileges.

Two separate but similar mechanisms handle impersonation in Unix, the so called set UID, (SUID), and set-GID (SGID) mechanisms. Every executable file on a file system so configured, can be marked for SUID/SGID execution. Such a file is executed with the permissions of the owner/group of the file, instead of the current user. Typically, certain services that require superuser privileges are wrapped in a SUID superuser program, and the users of the system are given permission to execute this program. If the program can be subverted into performing some action that it was not originally intended to perform, serious breaches of security can result.

The above system works well in a surprising number of situations. But we will illustrate a few situations where it fails to protect or even facilitates the attacker. Most systems today also support some form of access control list (ACL) based schemes.

Access control lists: On Unix systems, file permissions define the file mode as well. The file mode contains nine bits that determine access permissions to the file plus three special bits. This mechanism allows to define access permissions for three classes of users: the file owner, the owning group, and the rest of the world. These permission bits are modified using the *chmod* utility. The main advantage of this mechanism is its simplicity. With a couple of bits, many permission scenarios can be modeled. However, there often is a need to specify relatively fine-grained access permissions.

Access Control Lists (ACLs) support more fine grained permissions. Arbitrary users and groups can be granted or denied access in addition to the three traditional classes of users. The three classes of users can be regarded as three entries of an Access Control List. Additional entries can be added that define the permissions which the specific users or groups are granted. An example of the use of ACLs: Let's assume a small company producing soaps for all usages. We shall call it Soaps4All. Soaps4All runs a Linux system as its main file server. The system administrator of Soaps4All is called Damu. One particular team of users, the Toileteers, deals with the development of new toilet accessories. They keep all their shared data in the sub-directory /home/toileteers/shared. Kalyan is the administrator of the Toileteers team. Other members are Ritu, Vivek, and Ulhas.

Username	Groups	Function
Damu	users	System administrator
Kalyan	toileteers, jumboT, perfumedT	administrator
ritu	toileteers, jumboT	
vivek	toileteers, perfumedT	
ulhas	toileteers, jumboT, perfumedT	

Inside the shared directory, all Toileteers shall have read access. Kalyan, being the Toileteers administrator, shall have full access to all the sub-directories as well as to files in those sub-directories. Everybody who is working on a project shall have full access to the project's sub-directory in /home/toileteers/shared.

Suppose two brand new soaps are under development at the moment. These are called Jumbo and Perfumed. Ritu is working on Jumbo. Vivek is working on Perfumed. Ulhas is working on both the projects. This is clearly reflected by the users' group membership in the table above.

We have the following directory structure:

```
$ ls -l
drwx------ 2 Kalyan toileteers 1024 Apr 12 12:47 Kalyan
drwx------ 2 ritu toileteers 1024 Apr 12 12:47 ritu
drwxr-x--- 2 Kalyan toileteers 1024 Apr 12 12:48 shared
drwx------ 2 ulhas toileteers 1024 Apr 12 13:23 ulhas
drwx------ 2 vivek toileteers 1024 Apr 12 12:48 vivek
```

```
/shared$ls -l
drwxrwx--- 2 Kalyan jumbo 1024 Sep 25 14:09 jumbo
drwxrwx--- 2 Kalyan perfumed 1024 Sep 25 14:09 perfumed
```

Now note the following:

- Ritu does not have a read access to /home/toileteers/shared/perfumed.
- Vivek does not have read access to /home/toileteers/shared/jumbo.
- Kalyan does not have write access to files which others create in any project subdirectory.

The first two problems could be solved by granting everyone read access to the /home/toileteers/shared/ directory tree using the others permission bits (making the directory tree world readable). Since nobody else but Toileteers have access to the /home/toileteers directory, this is safe. However, we would need to take great care of the other permissions of the /home/toileteers directory.

Adding anything to the toileteers directory tree later that is world readable is impossible. With ACLs, there is a better solution. The third problem has no clean solution within the traditional permission system.

The solution using ACLs: The /home/toileteers/shared/ sub-directories can be made readable for Toileteers, and fully accessible for the respective project group. For Kalyan's administrative rights, a separate ACL entry is needed. This is the command to grant read access to the Toileteers. This is in addition to the existing permissions of other users and groups:

```
$setfacl -m g:toileteers:rx *
$getfacl *

# file: jumbo
# owner: Kalyan
# group: jumbo
user::rwx
group::rwx
group:toileteers:r-x
mask:rwx
other:---
# file: perfumed
# owner: Kalyan
# group: perfumed
user::rwx
group::rwx
group:toileteers:r-x
```

```
mask:rwx
other:---
```

Incidentally, AFS(Andrew File System), and XFS (SGI's eXtended File System) support ACLs.

Unit-2

1.1 Introduction to Unix Primer

1.2 Motivation

1.3 Unix Environment

1.4 Unix File System

1.5 10.4 Some Useful Unix Commands

1.6 Unix Portability

1.1 Introduction to Unix Primer

From UNICS To Unix: A brief history: - Early on, in the 1960s and 1970s, every major computer manufacturer supplied operating system as a proprietary software. Such OSs were written specifically for their own machine. In particular, each machine had an instruction set and the operating system was generally written in its intermediate language (often assembly language). As a result, no two operating systems were alike in features. When a user moved to a new machine, he would be expected to learn the new operating system. No two machines could even exchange information, not to mention the notion of portability of software.

It was in this context, that “unics", an acronym for uniplexed information and computing system was developed by Dennis Richie at Bell Laboratories in USA. The idea was to offer an interpreted common (uniplexed) command language environment across platforms. Unics later became UNIX [16]. To implement this idea, Bell Laboratory team developed the idea of isolating a basic “kernel" and a “shell". Most OSs today follow UNIX design philosophy of providing a kernel and a shell. Modern Unix-based environments support an extensive suite of tools.

1.2 Motivation

Unix is a popular operating system. Unix philosophy has been to provide a rich set of generic tools and to support tool based application development. For instance, Unix provides generic string matching tools which are very useful in software development. These tools and utilities also aid in enhancing a user's productivity. Clearly, the main advantage of such an approach is to leave a lot of leeway for the users. Experience indicates that this encourages users to use tools in innovative ways to create new applications. Or they may just create a pleasant customised work environment. Now contrast this with a closed and/or packaged environment. That leaves little room, if any, for creative composition or enhancements. It offers little or no outlet to a user to customise the working environment. In Unix, users have access to the same tools which are also used by Unix as an OS. This gives one a peek into the working of the internals of Unix as well. In fact, by letting the user enrich the tool suite, or the OS utilities, Unix users expand their horizon. This is what makes Unix an open system.

Besides the tools and utilities orientation, there were two other major developments which have affected operational environments. First, the development of X-windows offered users a very helpful environment to develop graphical user interface (GUI) for newer applications. Secondly, Unix provided a strong backbone support in the development of computer communication networks by supporting client-server architecture. In fact, the TCP/IP suite of protocols (which forms the core of internet operations) was first developed on Unix. In this module we shall study the elements of the Unix operating systems.

1.3 Unix Environment

Unix, with X-windows support, provides a virtual terminal in every window. A Unix environment is basically a command driven environment[1]. Each window can be used to give commands. With this a user can run multiple applications, one in each of the windows. A user may open more windows if he needs them (though there is usually an upper limit to the number of windows that may be active at one time).

Basically Unix provides a user a friendly shell. It hides the system kernel beneath the shell. The shell interface accepts user commands. The user command is interpreted by the shell and then the shell seeks the desired service from the kernel on behalf of the user.

In Unix everything veers around the following two basic concepts:

- Files
- Processes

Users organise applications using files and processes. A program text is a file. A program in execution is a process.

An application may be spread over several files. In addition to program files, there may be other kinds of data files too. Files are generated with some end application in mind. For instance, a file may be a text file for document generation or it may be an image file supporting some medical information in a health-care system. A user may have many applications. Each application may require several files. Unix file system helps users in organizing their files. Next we shall study the file system in Unix.

1.4 Unix File System

Unix allows a user to group related files within a directory. Suppose we want to work with two different applications, one involving documents and the other involving images.

May be we group the document related files under one directory and image related files in another directory. Sometimes it helps to further subdivide files. Unix allows creation of sub-directories to help organise files hierarchically. Such a file organisation creates a tree of directories with files as leaves.

When you log into a Unix system you always do so in your login directory. You may create files as well as subdirectories starting from the login directory.

An example of file hierarchy: When I created course notes for web browsing, I first created a directory COURSES under my home directory. Under this directory I created directories called OS, SE, COMPILERS respectively for operating systems, software engineering and compiler courses. You may browse the course pages at the URL http://www.iiitb.ac.in/bhatt. If you browse these files, you will notice that each course has files organized around topics grouped as Modules. In fact, there is an organizational hierarchy. The COURSES directory has subdirectories like OS, SE and COMPILERS within each of which there are modules. At the bottom most level within the modules subdirectory there are individual page files. The links, which help you navigate, lead you to image files or postscript files of the course pages.

It is important to know how one may reach a particular file in such a file hierarchy. The home directory is usually denoted in Unix by tilde (~). Suppose we wish to reach a particular module in OS course in my organisation of files as described above. This would require that we traverse down from the home to COURSES to OS to the particular module. This is called traversing a path. Our path would be /COURSES/OS/moduleX.

Note how each step in the path hierarchy is separated by a forward slash. We say that directory COURSES has a subdirectory (or a child directory) OS. We may also say that COURSES is the parent of directory OS. In Unix, all paths emanate from a directory called *root*. The root directory has no parent. A user's login directory is usually a child of a directory named home which is a child of “/”(i.e. root). Under home user's home directories are created with the user's login name. My home directory is named bhatt.

The directory where you may be currently located is denoted by a period (.) symbol. The parent of current directory is denoted by two periods (..). These two are special symbols. These symbols are used when the user has to make a reference *relative* to the present position in the file system. An absolute path name would trace the file path starting with root as in */home/bhatt/COURSES/OS/module5*. Suppose we are in directory COURSES then the same path shall be denoted as ./OS/module5. Note that the file path name has no spaces. In fact, no spaces are permitted within file path names.

Unix commands for files: The general command structure for Unix has the following structure:

< UnixCommand >< Options >< arguments >

A user may choose one or more options and one or more arguments. It is important to be well versed with the commands that support the identification of files and navigation on the directory tree. Here is a brief list of relevant Unix commands[34].

• *ls:* Lists all the files within a directory.

• *cd:* By itself it brings you back to home directory.

• *cd pathname:* Takes you to the directory described by the *pathname.*

• *rm filename*:: Removes file *filename* from the current directory.

• *pwd:* Prints the name of the current working directory.

• *mkdir subdirname:* Creates a subdirectory under the current directory with the name *subdirname.*

• *rmdir subdirname:* Removes a subdirectory under the current directory with the name *subdirname.*

• *touch filename:* Creates a file in the current directory with the name *filename.*

This file, on creation has 0 characters and 0 lines.

For now you should use the above listed commands to create files and directories. You should basically learn to navigate the directory tree. Later, we shall learn more commands applicable to files. You should also consult the online manual for options available on *ls* and *rm* commands.

In general, files have three basic operations associated with them. These are *read, write,* and *execute*.

Unix supports four types of files.

- Ordinary files: These are usually text files. Programs written in specific programming languages may have specific extensions. For instance, programs written in C have a .c extension. Files prepared for TeX documentation have a .tex extension. These are usually the files users create most often using an editor.
- Directories: Subdirectories are treated as files by Unix.
- Binary files: These are executables.

- Special files: Special files provide input/output capability for Unix environment. Unix treats the device IO as file communication. In fact, these gives the look and feel of a file read or write whenever a user communicates with an IO device.

1.5 Some Useful Unix Commands

We give here a list of some commonly used Unix commands:

- *bc:* Gives you a basic calculator. Try simple arithmetic to get a feel for its operation.
- *cal:* Shows the calendar for the current month. A variation of *cal* will allow you to
- view calendar for any month or year.
- *clear:* Clears the screen.
- *cp filename1 filename2:* Creates a copy of file filename1 in filename2.
- *date:* Shows the current time and date.
- *echo sometext:* Echos back *sometext* on the terminal.
- *history:* Shows the previously given command history. This may be customised.
- *more(filename:* Shows the file filename one page at a time. *less* does the same
- except that it gives scrolling additionally.
- *cat filename:* Displays the file filename on the screen.
- *cat filename(s)* > *newfile:* Combines all the files in filename(s) and outputs to create a file newfile.
- *manAUnixCmd:* Shows the description of the command AUnixCmd from the online help manual. It describes the use of a command with all its possible options and arguments.
- *exit:* Exits the current shell.

1.6 Unix Portability

In Section 1.2 we described why Unix may be regarded as an open system. One of the main outcomes of such an open system, as Unix has been, is that there are many versions of Unix with mutual incompatibility. This led to problems in communication and interoperability. Fortunately, there are some standards now. Two of the better known standards are X/open and POSIX. In developing the POSIX, professional organizations like IEEE (URL: http://standards.ieee.org) have been involved. Now-a-days most developers ensure that their products are POSIX compliant. POSIX compliant software adheres to some interface protocols. This ensures interoperability. A very good URL which describes both X/Open and POSIX is http://www.starlink.rl.ac.uk/star/docs/sun121.htx/node9.html.

Finally: We discussed some Unix commands in this module. This description is far from complete. For instance, Unix allows the use of regular expressions in arguments. We have not discussed that in this module. This is because in the next module we shall be discussing tools for pattern matching which requires extensive use of regular expressions.

However, some exercises in this chapter may contain regular expressions. It is expected that the users of Unix are motivated to learn a few such details on the fly.

Unit-3

1.1 Search and Sort Tools

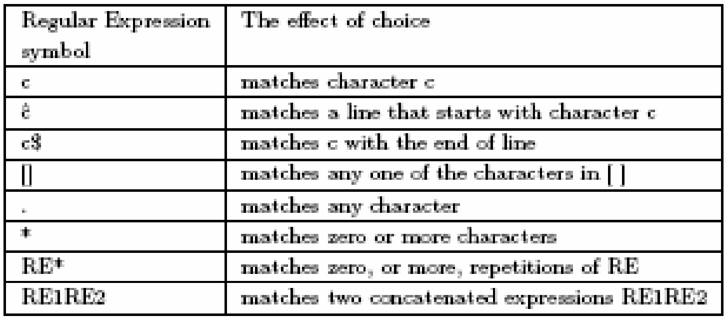
Unix philosophy is to provide a rich set of generic tools, each with a variety of options. These primitive Unix tools can be combined in imaginative ways (by using pipes) to enhance user productivity. The tool suite also facilitates to build either a user customized application or a more sophisticated and specialised tool.

We shall discuss many primitive tools that are useful in the context of text files. These tools are often called "filters" because these tools help in searching the presence or absence of some specified pattern(s) of text in text files. Tools that fall in this category include ls, grep, and find. For viewing the output from these tools one uses tools like *more, less, head, tail*. The sort tool helps to sort and tools like *wc* help to obtain statistics about files. In this chapter we shall dwell upon each of these tools briefly. We shall also illustrate some typical contexts of usage of these tools.

1.2 *grep, egrep* and *fgrep*

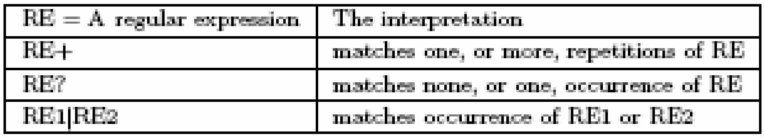
grep stands for general regular expression parser. *egrep* is an enhanced version of *grep*. It allows a greater range of regular expressions to be used in pattern matching. *fgrep* is for fast but fixed string matching. As a tool, *grep* usage is basic and it follows the syntax: *grep* options pattern files with the semantics that search the file(s) for lines with the pattern and options as command modifiers. The following example1 shows how we can list the lines with int declarations in a program called *add.c*. Note that we could use this trick to collate all the declarations from a set of files to make a common include file of definitions.

```
bhatt@SE-0 [~/UPE] >>grep int ./C/add.c
extern int a1; /* First operand */
extern int a2; /* Second operand */
extern int add();
printf("The addition gives %d \n", add());
```



Regular Expression symbol	The effect of choice
c	matches character c
ĉ	matches a line that starts with character c
c$	matches c with the end of line
[]	matches any one of the characters in [ ]
.	matches any character
*	matches zero or more characters
RE*	matches zero, or more, repetitions of RE
RE1RE2	matches two concatenated expressions RE1RE2

Table 11.1: Regular expression options.



RE = A regular expression	The interpretation
RE+	matches one, or more, repetitions of RE
RE?	matches none, or one, occurrence of RE
RE1|RE2	matches occurrence of RE1 or RE2

Table 11.2: Regular expression combinations.

Note: print has int in it !!

In other words, *grep* matches string literals. A little later we will see how we may use options to make partial patterns for intelligent searches. We could have used **.c* to list the lines in all the c programs in that directory. In such a usage it is better to use it as shown in the example below

```
grep int ./C/*.c | more
```

This shows the use of a pipe with another tool *more* which is a good screen viewing tool. more offers a one screen at a time view of a long file. As stated in the last chapter, there is a program called less which additionally permits scrolling.

Regular Expression Conventions: Table 11.1 shows many of the grep regular expression conventions. In Table 11.1, RE, RE1, and RE2 denote regular expressions. In practice we may combine Regular Expressions in arbitrary ways as shown in Table 11.2. *egrep* is an enhanced *grep* that allows additionally the above pattern matching capabilities. Note that an RE may be enclosed in parentheses. To practice the above we make a file called testfile with entries as shown. Next, we shall try matching patterns using various options. Below we show a session using our text file called testfile.

```
aaa
a1a1a1
456
10000001
This is a test file.
bhatt@SE-0 [F] >>grep '[0-9]' testfile
a1a1a1
```

```
456
10000001
bhatt@SE-0 [F] >>grep '^4' testfile
456
bhatt@SE-0 [F] >>grep '1$' testfile
a1a1a1

10000001
bhatt@SE-0 [F] >>grep '[A-Z]' testfile
This is a test file.
bhatt@SE-0 [F] >>grep '[0-4]' testfile
a1a1a1
456
10000001
bhatt@SE-0 [F] >>fgrep '000' testfile
10000001
bhatt@SE-0 [F] >>egrep '0..' testfile
10000001
```

\ The back slash is used to consider a special character literally. This is required when the character used is also a command option as in case of -, * etc.

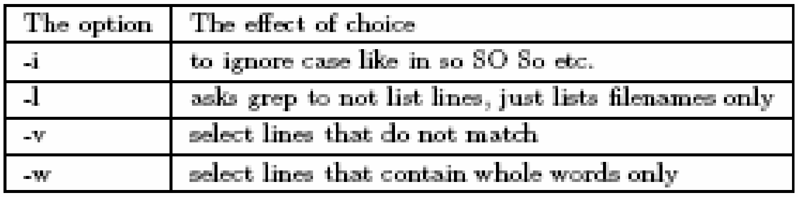
See the example below where we are matching a period symbol.

```
bhatt@SE-0 [F] >>grep '\.' Testfile
```

This is a test file.

We may use a character's characteristics as options in grep. The options available are shown in Table 11.3.

```
bhatt@SE-0 [F] >>grep -v 'a1' testfile
aaa
456
10000001
```



The option	The effect of choice
-i	to ignore case like in so SO So etc.
-l	asks grep to not list lines, just lists filenames only
-v	select lines that do not match
-w	select lines that contain whole words only

Table 11.3: Choosing match options.

```
This is a test file.
bhatt@SE-0 [F] >>grep 'aa' testfile
aaa
bhatt@SE-0 [F] >>grep -w 'aa' testfile
bhatt@SE-0 [F] >>grep -w 'aaa' testfile
aaa
```

```
bhatt@SE-0 [F] >>grep -l 'aa' testfile

testfile
```

Context of use: Suppose we wish to list all sub-directories in a certain directory.

```
ls -l | grep ^d
bhatt@SE-0 [M] >>ls -l | grep ^d
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 13:15 M1
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 12:37 M2
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 12:37 M3
drwxr-xr-x 2 bhatt bhatt 512 Oct 16 09:53 RAND
```

Suppose we wish to select a certain font and also wish to find out if it is available as a bold font with size 18. We may list these with the instruction shown below.

```
xlsfonts | grep bold\-18 | more
bhatt@SE-0 [M] >>xlsfonts | grep bold\-18 | more
lucidasans-bold-18
lucidasans-bold-18
lucidasanstypewriter-bold-18
lucidasanstypewriter-bold-18
```

Suppose we wish to find out at how many terminals a certain user is logged in at the moment. The following command will give us the required information:

```
who | grep username | wc -l > count
```

The *wc* with *-l* options gives the count of lines. Also, who|grep will output one line for every line matched with the given pattern (username) in it.

1.3 Using *find*

find is used when a user, or a system administrator, needs to determine the location of a certain file under a specified subtree in a file hierarchy. The syntax and use of *find* is:

```
find path expression action
```

where path identifies the subtree, expression helps to identify the file and the action specifies the action one wishes to take. Let us now see a few typical usages.

- List all the files in the current directory.

```
     bhatt@SE-0 [F] >>find . -print /* only path and action specified */
    .
    /ReadMe
    ./testfile
```

- Finding files which have been created after a certain other file was created.

```
bhatt@SE-0 [F] >>find . -newer testfile
.
/ReadMe
```

There is an option mtime to find modified files in a certain period over a number of days.

- List all the files which match the partial pattern test. One should use only shell meta-characters for partial matches.

```
bhatt@SE-0 [F] >>find . -name test*
./testfile
bhatt@SE-0 [F] >>find ../../ -name test*
../../COURSES/OS/PROCESS/testfile
../../UPE/F/testfile
```

- I have a file called linkedfile with a link to testfile. The find command can be used to find the links.

```
bhatt@SE-0 [F] >>find ./ -links 2
./
./testfile
./linkedfile
```

- Finding out the subdirectories under a directory.

```
bhatt@SE-0 [F] >>find ../M -type d
../M
../M/M1
../M/M2
../M/M3
../M/RAND
```

- Finding files owned by a certain user.

```
bhatt@SE-0 [F] >>find /home/georg/ASM-WB -user georg -type d
/home/georg/ASM-WB
/home/georg/ASM-WB/examples
/home/georg/ASM-WB/examples/Library
/home/georg/ASM-WB/examples/SimpleLang
/home/georg/ASM-WB/examples/InstructionSet
```

```
        /home/georg/ASM-WB/Projects
        /home/georg/ASM-WB/FirstTry
```

The file type options are: f for text file, c for character special file, b for blocked files and p for pipes.

Strings: Sometimes one may need to examine if there are ascii strings in a certain object or a binary file. This can be done using a string command with the syntax:

```
string binaryfileName | more
```

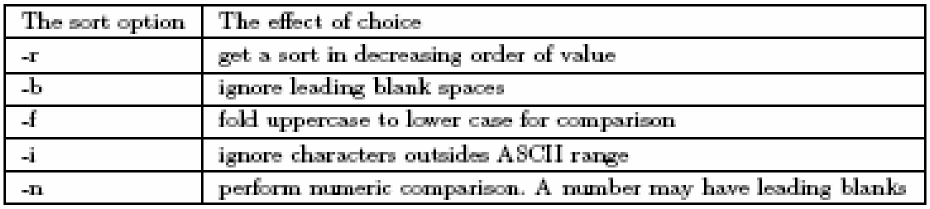
As an example see its use below2:

```
bhatt@SE-0 [F] >>strings ../M/RAND/main | more
The value of seed is %d at A
The value is %f at B
```

ctags and etags: These commands are useful in the context when one wishes to look up patterns like c function calls. You may look up man pages for its description if you are a power user of c.

1.3.1 Sort Tool

For sort tool Unix treats each line in a text file as data. In other words, it basically sorts lines of text in text file. It is possible to give an option to list lines in ascending or descending order. We shall demonstrate its usage through examples given below.



The sort option	The effect of choice
-r	get a sort in decreasing order of value
-b	ignore leading blank spaces
-f	fold uppercase to lower case for comparison
-i	ignore characters outsides ASCII range
-n	perform numeric comparison. A number may have leading blanks

Table 11.4: Sort options.

```
bhatt@SE-0 [F] >>sort
aaa
bbb
aba
^d terminates the input ....see the output below
aaa
aba
bbb
bhatt@SE-0 [F] >>sort testfile
10000001
```

```
456
This is a test file.
a1a1a1
aaa
```

(Use -r option for descending order).

```
bhatt@SE-0 [F] >>sort testfile -o outfile
bhatt@SE-0 [F] >>
```

One can see the outfile for sorted output. *sort* repeats all identical lines. It helps to use a filter *uniq* to get sorted output with unique lines. Let us now modify our testfile to have repetition of a few lines and then use *uniq* as shown below.

```
bhatt@SE-0 [F] >>sort testfile|uniq|more
10000001
456
This is a test file.
a1a1a1
aaa
```

In table 11.4 we list the options that are available with *sort. sort* can also be used to merge files. Next, we will split a file and then show the use of merge. Of course, the usage is in the context of merge-sort.

One often uses filtering commands like *sort, grep* etc. in conjunction with *wc, more, head* and *tail* commands available in Unix. System administrators use *who* in conjunction with *grep, sort, find* to track of terminal usage and also for lost or damaged files.

split: *split* command helps one to split a file into smaller sized segments. For instance, if we *split* ReadMe file with the following command:

```
split -l 20 ReadMe seg
```

Upon execution we get a set of files segaa, segab, ...etc. each with 20 lines in it. (Check the line count using *wc*). Now merge using sorted segaa with segab.

```
sort -m segaa segab > check
```

A clever way to merge all the split files is to use *cat* as shown below:

```
cat seg* > check
```

The file check should have 40 lines in it. Clearly, *split* and *merge* would be useful to support merge-sort and for assembling a set of smaller files that can be sent over a network using e-mail whenever there are restrictions on the size of attached files.

In the next module we shall learn about the AWK tool in Unix. Evolution of AWK is a very good illustration of how more powerful tools can be built. AWK evolves from the (seemingly modest) generic tool *grep*!!

1.4 AWK Tool in Unix

AWK was developed in 1978 at the famous Bell Laboratories by Aho, Weinberger and Kernighan [3]1 to process structured data files. In programming languages it is very common to have a definition of a record which may have one or more data fields. In this context, it is common to define a file as a collection of records. Records are structured data items arranged in accordance with some specification, basically as a pre-assigned sequence of fields. The data fields may be separated by a space or a tab. In a data processing environment it very common to have such record-based files. For instance, an organisation may maintain a personnel file. Each record may contain fields like employee name, gender, date of joining the organisation, designation, etc. Similarly, if we look at files created to maintain pay accounts, student files in universities, etc. all have structured records with a set of fields. AWK is ideal for the data processing of such structured set of records. AWK comes in many flavors [14]. There is gawk which is GNU AWK. Presently we will assume the availability of the standard AWK program which comes bundled with every flavor of the Unix OS. AWK is also available in the MS environment.

1.4.1 The Data to Process

As AWK is used to process a structured set of records, we shall use a small file called *awk.test* given below. It has a structured set of records. The data in this file lists employee name, employee's hourly wage, and the number of hours the employee has worked.

(File awk.test)

```
bhatt 4.00 0
ulhas 3.75 2
ritu 5.0 4
vivek 2.0 3
```

We will use this candidate data file for a variety of processing requirements. Suppose we need to compute the amount due to each employee and print it as a report. One could write a C language program to do the task. However, using a tool like AWK makes it simpler and perhaps smarter. Note that if we have a tool, then it is always a good idea to use it. This is because it takes less time to get the results. Also, the process is usually less error prone. Let us use the awk command with input file *awk.test* as shown below:

```
bhatt@falerno [CRUD] =>awk '$3 > 0 { print $1, $2 * $3 }' awk.test
```

```
ulhas 7.5
ritu 20
vivek 6
```

Note some features of the syntax above | the awk command, the quoted string following it and the data file name. We shall next discuss first a few simple syntax rules. More advanced features are explained through examples that are discussed in Section 1.5.

1.4.2 AWK Syntax

To run an AWK program we simply give an “awk” command with the following syntax:

```
awk [options] <awk_program> [input_file]
```

where the options may be like a file input instead of a quoted string. The following should be noted:

- Note that in the syntax awk 'awk_program' [input_files] , the option on input files may be empty. That suggests that awk would take whatever is typed immediately after the command is given.
- Also, note that fields in the data file are identified with a $ symbol prefix as in $1. In the example above we have a very small AWK program. It is the quoted string reproduced below:

  ```
  '$3 > 0 {print $1, $2 * $3}'
  ```

  The interpretation is to print the name corresponding to $1, and the wages due by taking a product of rate corresponding to $2 multiplied with the number of hours corresponding to $3. In this string the $ prefixed integers identify the fields we wish to use.
- In preparing the output: {print} or {print $0} prints the whole line of output. {print $1, $3} will print the selected fields.

In the initial example we had a one line awk program. Basically, we tried to match a pattern and check if that qualified the line for some processing or action. In general, we may have many patterns to match and actions to take on finding a matching pattern. In that case the awk program may have several lines of code. Typically such a program shall have the following structure:

```
pattern {action}
pattern {action}

pattern {action}
.
.
```

If we have many operations to perform we shall have many lines in the AWK program. It would be then imperative to put such a program in a file and AWKing it would require using a file input option as shown below. So if the awk program is very long and kept in a file, use the *-f* option as shown below:

```
awk -f 'awk_program_file_name' [input_files]
```

where the awk program file name contains the awk program.

1.5 Programming Examples

We shall now give a few illustrative examples. Along with the examples we shall also discuss many other features that make the task of processing easier.

- **Example 1**

Suppose we now need to find out if there was an employee who did no work. Clearly his hours work field should be equal to 0. We show the AWK
program to get that.

```
bhatt@falerno [CRUD] =>awk '$3 == 0 {print $1}' awk.test bhatt
```

The basic operation here was to scan a sequence of input lines searching for the lines that match any of the patterns in the program. Patterns like $3 > 0 match the 3rd field when the field has a value > 0 in it.

An Aside: Try a few errors and see the error detection on the one line awk programs.

➢ **Example 2**

In this example we shall show the use of some of the built-in variables which help in organizing our data processing needs. These variables acquire meaning in the context of the data file. NF is a built in variable which stores the number of fields and can be used in such context as fprint NF, $1, $NFg which prints the number of fields, the first and the last field.

Another built-in variable is NR, which takes the value of the number of lines read so far and can also be used in a print statement.

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print NR, NF, $1, $NF }' awk.test
```

```
3 3 ulhas 2
4 3 ritu 4
5 3 vivek 3
```

- **Example 3**

The formatted data in files is usually devoid of any redundancy. However, one needs to generate verbose output. This requires that we get the values and interspread the desired strings and generate a verbose and meaningful output. In this example we will demonstrate such a usage.

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {print "person ", NR, $1, "be paid",

$2*$3,

"dollarperson 3 ulhas be paid 7.5 dollars

person 4 ritu be paid 20 dollars

person 5 vivek be paid 6 dollars
```

One can use printf to format the output like in the C programs.

```
bhatt@falerno [CRUD] =>awk '$3 > 0 {printf("%-8s be paid $%6.2f dollars
“n", $1,
$2*$3ulhas be paid $ 7.50 dollars
ritu be paid $ 20.00 dollars
vivek be paid $ 6.00 dollars
```

An Aside: One could sort the output by <awk_program> | sort i.e. by a pipe to sort.

- **Example 4**

In the examples below we basically explore many selection possibilities. In general the selection of lines may be by comparison involving computation. As an example, we may use $2 > 3.0 to mean if the rate of payment is greater than 3.0.

We may check for if the total due is > 5, as $2*$3 > 5:0, which is an example of comparison by computation.

One may also use a selection by text content (essentially comparison in my opinion). This is done by enclosing the test as /bhatt/ to identify $1 being string “bhatt" as in $1 == /bhatt/.
Tests on patterns may involve relational or logical operators as $ >=; ||

Awk is excellent for data validation. Checks like the following may be useful.

. NF != 3 ... no. of fields not equal to 3

. $2 < 2.0 .. wage rate below min. stipulated

. $2 > 10.0 . ...exceeding max. .....

. $3 < 0 ...no. of hrs worked -ve etc.

It should be remarked that data validation checks are a very important part of data processing activity. Often an organization may employ or outsource data preparation. An online data processing may result in disasters if the data is not validated. For instance, with a wrong hourly wage field we may end up creating a pay cheque which may be wrong. One needs to ensure that the data is in expected range lest an organization ends up paying at a rate below the minimum legal wage or pay extra-ordinarily high amounts to a low paid worker!

- **Example 5**

  In these examples we demonstrate how we may prepare additional pads to give the formatted data a look of a report under preparation. For instance, we do not have headings for the tabulated output. One can generate meaningful headers and trailers for a tabulated output. Usually, an AWK program may have a BEGIN key word to identify some pre-processing that can help prepare headers before processing the data file. Similarly, an AWK program may be used to generate a trailer with END key word. The next example illustrates such a usage. For our example the header can be generated by putting BEGIN {print "Name Rate Hours"} as preamble to the AWK program as shown below.

```
bhatt@falerno [CRUD] =>awk 'BEGIN{ print"name rate hours"; print""} "
{print}' awk.test
```

```
name          rate                hours
bhatt         4.00                 0
ulhas          3.75                2
ritu          5.0                 4
vivek         2.0                 3
```

Note that print "" prints a blank line and the next print reproduces the input. In general, BEGIN matches before the first line of input and END after the last line of input. The ; is used to separate the actions. Let us now look at a similar program with -f option.

file awk.prg is

```
BEGIN {print "NAME RATE HOURS"; print ""} { print $1," ",$2," ",$3,"..."}
bhatt@falerno [CRUD] =>!a
```

```
awk -f awk.prg awk.test
NAME            RATE       HOURS
bhatt           4.00       0 ...
ulhas           3.75       2 ...
ritu            5.0        4 ...
vivek           2.0        3 ...
```

- **Example 6**

Now we shall attempt some computing within awk. To perform computations we may sometimes need to employ user-defined variables. In this example “pay" shall be used as a user defined variable. The program accumulates the total amount to be paid in “pay". So the printing is done after the last line in the data file has been processed, i.e. in the END segment of awk program. In NR we obtain all the records processed (so the number of employees can be determined). We are able to do the computations like “pay" as a total as well as compute the average salary as the last step.

```
BEGIN {print "NAME RATE HOURS"; print ""}
{ pay = pay + $2*$3 }
END {print NR "employees"
print "total amount paid is : ", pay
print "with the average being :", pay/NR}
     bhatt@falerno [CRUD] =>!a
```

```
awk -f prg2.awk awk.test

4 employees

total amount paid is : 33.5

with the average being : 8.375
```

A better looking output could be produced by using *printf* statement as in *c*. Here is another program with its output. In this program, note the computation of “maximum" values and also the concatenation of names in “emplist". These are user-defined data-structures. Note also the use of “last" to store the last record processed, i.e. $0 gets the record and we keep storing it in last as we go along.

```
BEGIN {print "NAME RATE HOURS"; print ""}
{pay = pay + $2*$3}
$2 > maxrate {maxrate = $2; maxemp = $1}
{emplist = emplist $1 " "}
{last = $0}
END {print NR " employees"
print "total amount paid is : ", pay
print "with the average being :", pay/NR
print "highest paid rate is for " maxemp, " @ of : ", maxrate
print emplist
print ""
print "the last employee record is : ", last}
```

output is

```
bhatt@falerno [CRUD] =>!a
awk -f prg3.awk test.data
4 employees
total amount paid is : 33.5
with the average being : 8.375
highest paid rate is for ritu @ of : 5.0
bhatt ulhas ritu vivek
the last employee record is : vivek 2.0 3
```

- **Example 7**

There are some builtin functions that can be useful. For instance, the function “length" helps one to compute the length of the argument field as the number of characters in that field. See the program and the corresponding output below:

```
{ nc = nc + length($1) + length($2) + length($3) + 4 }
{ nw = nw + NF }
END {print nc " characters and "; print ""
print nw " words and "; print ""
print NR, " lines in this file "}
bhatt@falerno [CRUD] =>!a
awk -f prg4.awk test.data
```

```
53 characters and
12 words and
4 lines in this file
```

- **Example 8**

  AWK supports many control flow statements to facilitate programming. We will first use the if-else construct. Note the absence of "then" and how the statements are grouped for the case when the if condition evaluates to true. Also, in the program note the protection against division by 0.

  ```
  BEGIN {print "NAME RATE HOURS"; print ""}
  $2 > 6 {n = n+1; pay = pay + $2*$3}
  $2 > maxrate {maxrate = $2; maxemp = $1}
  {emplist = emplist $1 " "}
  {last = $0}
  END {print NR " employees in the company "
  ```

```
    if ( n > 0 ) {print n, "employees in this bracket of salary. "
    print "with an average salary of ", pay/n, "dollars"
    } else print " no employee in this bracket of salary. "
    print "highest paid rate is for " maxemp, " @ of : ", maxrate
    print emplist
        print ""}
```

This gives the result shown below:

```
bhatt@falerno [CRUD] =>!a

awk -f prg5.awk data.awk

4 employees in the company

no employee in this bracket of salary.

highest paid rate is for ritu @ of : 5.0

bhatt ulhas ritu vivek
```

Next we shall use a “while" loop2. In this example, we simply compute the compound interest that accrues each year for a five year period.

```
#compound interest computation
#input : amount rate years
#output: compounded value at the end of each year
{ i = 1; x = $1;
while (i <= $3)
{ x = x + (x*$2)
printf("“t%d”t%8.2f”n",i, x)
i = i + 1
}
}
```

The result is shown below:

```
bhatt@falerno [CRUD] =>!a

awk -f prg6.awk

1000 0.06 5

1 1060.00

2 1123.60

3 1191.02

4 1262.48

5 1338.23
```

AWK also supports a “for" statement as in

```
for (i = 1; i <= $3; i = i + 1)
```

which will have the same effect. AWK supports arrays too, as the program below demonstrates.

# reverse - print the input in reverse order ...

```
BEGIN {print "NAME RATE HOURS"; print ""}
{line_ar [NR] = $0} # remembers the input line in array line_ar
END {# prepare to print in reverse order as input is over now
for (i = NR; i >= 1; i = i-1)
print line_ar[i]
}
```

The result is shown below.

```
bhatt@falerno [CRUD] =>awk -f prg7.awk data.awk

        NAME           RATE             HOURS
        vivek            2.0              3
        ritu             5.0              4
        ulhas            3.75             2
        bhatt            4.00             0
```

1.5.1 Some One-liners

Next we mention a few one-liners that are now folklore in the AWK programming community. It helps to remember some of these at the time of writing programs in AWK.

1. Print the total no. of input lines: END {print NR}.

2. Print the 10th input line: NR = 10.

3. Print the last field of each line: “{print “$NF”}.

4. Print the last field of the last input line:

```
{field = $NF}

END {print field}
```

5. Print every input line with more than 4 fields: NF > 4.

6. Print every input line i which the last field is more than 4: $NF > 4.

7. Print the total number of fields in all input lines.

```
{nf = nf + NF}

END {print nf}
```

8. Print the total number of lines containing the string “bhatt".

9. Print the largest first field and the line that contains it.

```
        $1 > max {max = $1; maxline = $0}
        END {print max, maxline}
```

10. Print every line that has at least one field: NF > 0.

11. Print every line with > 80 characters: length($0) > 80.

12. Print the number of fields followed by the line itself.

```
{print NF, $0}
```

13. Print the first two fields in opposite order: {print $2, $1}.
14. Exchange the first two fields of every line and then print the line:

```
{temp = $1; $1 = $2, $2 = temp, print}
```

15. Print every line with the first field replaced by the line number:

```
{$1 = NR; print}
```

16. Print every line after erasing the second field:

```
{$2 = ""; print}
```

17. Print in reverse order the fields of every line:

```
{for (i = NF; i > 0; i = i-1) printf("%s ", $i)
printf("“n")}
```

18. Print the sums of fields of every line:

```
{sum = 0
for (i = 1; i <= NF; i = i+1) sum = sum + $i
print sum}
```

19. Add up all the fields in all the lines and print the sum:

```
{for (i = 1; i <= NF; i = i+1) sum = sum + $i}
END {print sum}
```

20. Print every line after replacing each field by its absolute value:

```
{for (i = 1; i <= NF; i = i+1) if ($i < 0) $i = -$i
print}
```

1.6 AWK Grammar

At this stage it may be worth our while to recapitulate some of the grammar rules. In particular we shall summarize the patterns which commonly describe the AWK grammar.

The reader should note how right across all the tools and utilities, Unix maintains the very same regular expression conventions for programming.

1. BEGIN{statements}: These statements are executed once before any input is processed.
2. END{statements}: These statements are executed once all the lines in the data input file have been read.
3. expr.{statements}: These statements are executed at each input line where the expr is true.
4. /regular expr/{statements}: These statements are executed at each input line that contains a string matched by regular expression.
5. compound pattern{statements}: A compound pattern combines patterns with && (AND), || (OR) and ! (NOT) and parentheses; the statements are executed at each input line where the compound pattern is true.
6. pattern1, pattern2 {statements}: A range pattern matches each input line from a line matched

by “pattern1" to the next line matched by “pattern2", inclusive; the statements are executed at each matching line.

7. “BEGIN" and “END" do not combine with any other pattern. “BEGIN" and “END" also always require an action. Note “BEGIN" and “END" technically do not match any input line. With multiple “BEGIN" and “END" the action happen in the order of their appearance.
8. A range pattern cannot be part of any other pattern.
9. “FS" is a built-in variable for field separator.

Note that expressions like $3/$2 > 0.5= match when they evaluate to true. Also “The" < “Then" and “Bonn" > “Berlin". Now, let us look at some string matching considerations. In general terms, the following rules apply.

1. /regexpr/ matches an input line if the line contains the specified substring. As an example : /India/ matches “ India " (with space on both the sides), just as it detects presence of India in “Indian".
2. expr ~ /regexpr/ matches, if the value of the expr contains a substring matched by regexpr. As an example, $4 ~ /India/ matches all input lines where the fourth field contains “India" as a substring.
3. expr !~/regexpr/ same as above except that the condition of match is opposite. As an example, $4 !~/India/ matches when the fourth field does not have a substring “India".

The following is the summary of the Regular Expression matching rules.

^C : matches a C at the beginning of a string

C$: matches a C at the end of a string

^C$: matches a string consisting of the single character C

^.$: matches single character strings

^...$: matches exactly three character strings

... : matches any three consecutive characters

“.$: matches a period at the end of a string

* : zero or more occurrences

? : zero or one occurrence

+ : one or more occurrence

The regular expression meta characters are:

1. \ ^ $. [ ] | ( ) * + ?

2. A basic RE is one of the following:

A non meta character such as A that matches itself

An escape sequence that matches a special symbol: “t matches a tab.

A quoted meta-character such as “* that matches meta-ch literally

^, which matches beginning of a string

$, which matches end of a string

., which matches any single character

A character class such as [ABC] matches any of the A or B or C

Character abbreviations [A-Za-z] matches any single character

A complemented character class such as [^0-9] matches non digit characters

3. These operators combine REs into larger ones:

alteration : A|B matches A or B

concatenation : AB matches A immediately followed by B

closure : A* matches zero or more A's

positive closure : A+ matches one or more A's

zero or one : A? matches null string or one A

parentheses : (r) matches same string as r does.

The operator precedence being |, concatenation, (*, +, ?) in increasing value. i.e. *, +, ? bind stronger than |

The escape sequences are:

“b : backspace; “f : form feed; “n : new-line; “c literally c (““ for “)

“r : carriage return; “t : tab; “ddd octal value ddd

Some useful patterns are:

/^[0-9]+$/ : matches any input that consists of only digits

/^(“+|-)?[0-9]+”.?[0-9]/ : matches a number with optional sign and optional fractional part

/^[A-Za-z][A-Za-z0-9]*$/ : matches a letter followed by any letters or digits, an awk variable

/^[A-Za-z]$|^[A-Za-z][0-9]$/ : matches a letter or a letter followed by any letters or digits (a variable in Basic)

Now we will see the use of FILENAME (a built-in variable) and the use of range operators in the RE.

1.6.1 More Examples

For the next set of examples we shall consider a file which has a set of records describing the performance of some of the well-known cricketers of the recent past. This data is easy to obtain from any of the cricket sites like khel.com or cricinfo.com or the International Cricket Councils' website. The data we shall be dealing with is described in Table 12.1. In Table 12.1, we have information such as the name of the player, his country affliation, matches played, runs scored, wickets taken, etc. We should be able to scan the cricket. data file and match lines to yield desired results. For instance, if we were to look for players with more than 10,000 runs scored, we shall expect to see SGavaskar and ABorder. Similarly, for anyone with more than 400 wickets we should expect to see Kapildev and RHadlee3.

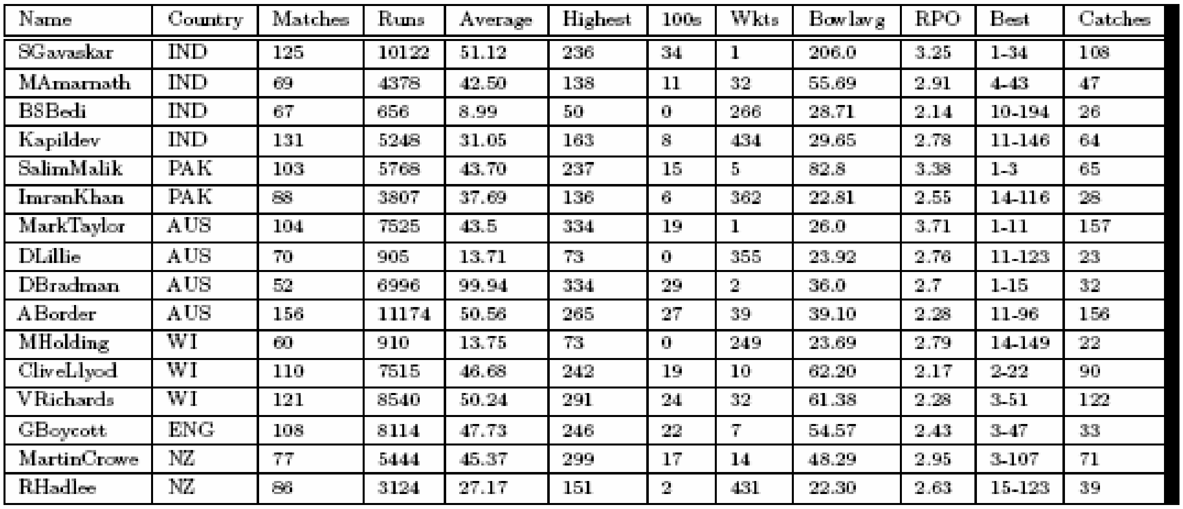
So let us begin with our example programs.

1. Example – 1

# In this program we identify Indian cricketers and mark them with ***.

# We try to find cricketers with most runs, most wickets and most catches

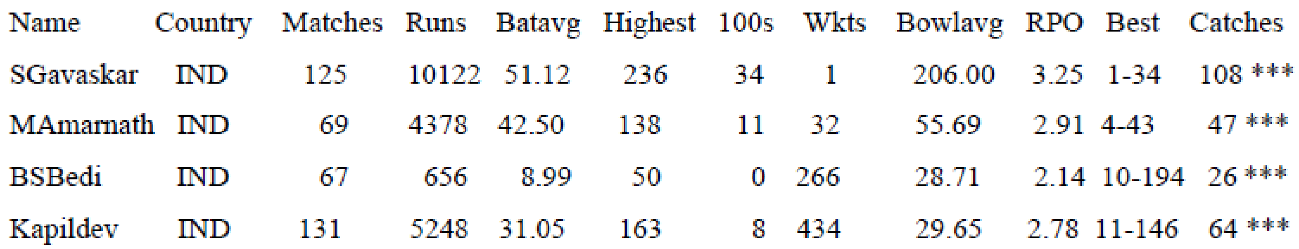
```
BEGIN {FS = "“t" # make the tab as field separator
printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s”n”n",
        "Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
```



Name	Country	Matches	Runs	Average	Highest	100s	Wkts	Bowlavg	RPO	Best	Catches
SGavaskar	IND	125	10122	51.12	236	34	1	206.0	3.25	1-34	108
MAmarnath	IND	69	4378	42.50	138	11	32	55.69	2.91	4-43	47
BSBedi	IND	67	656	8.99	50	0	266	28.71	2.14	10-194	26
Kapildev	IND	131	5248	31.05	163	8	434	29.65	2.78	11-146	64
SalimMalik	PAK	103	5768	43.70	237	15	5	82.8	3.38	1-3	65
ImranKhan	PAK	88	3807	37.69	136	6	362	22.81	2.55	14-116	28
MarkTaylor	AUS	104	7525	43.5	334	19	1	26.0	3.71	1-11	157
DLillie	AUS	70	905	13.71	73	0	355	23.92	2.76	11-123	23
DBradman	AUS	52	6996	99.94	334	29	2	36.0	2.7	1-15	32
ABorder	AUS	156	11174	50.56	265	27	39	39.10	2.28	11-96	156
MHolding	WI	60	910	13.75	73	0	249	23.69	2.79	14-149	22
CliveLlyod	WI	110	7515	46.68	242	19	10	62.20	2.17	2-22	90
VRichards	WI	121	8540	50.24	291	24	32	61.38	2.28	3-51	122
GBoycott	ENG	108	8114	47.73	246	22	7	54.57	2.43	3-47	33
MartinCrowe	NZ	77	5444	45.37	299	17	14	48.29	2.95	3-107	71
RHadlee	NZ	86	3124	27.17	151	2	431	22.30	2.63	15-123	39

Table 12.1: The cricket data file.

```
"Bowlavg","Rpo","Best","Catches")}
$2 ~/IND/ { printf("%12s %5s %7s %6s %6s %7s %4s %8s %8s %4s %7s %7s %3s”n",
$1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,"***")}
$4 > runs {runs = $4;name1 = $1}
$8 > wickets {wickets = $8;name2 = $1}
$12 > catches {catches = $12;name3 = $1}
END
{printf("“n %15s is the highest scorer with %6s runs",name1,runs)
printf("“n %15s is the highest wicket taker with %8s wickets",name2,wickets)
printf("“n %15s is the highest catch taker with %7s catches”n",name3,catches)
}
bhatt@falerno [AWK] =>!a
awk -f cprg.1 cricket.data
```



```
Name        Country  Matches  Runs   Batavg  Highest  100s  Wkts  Bowlavg  RPO   Best    Catches
SGavaskar   IND        125    10122  51.12     236     34     1    206.00  3.25  1-34    108 ***
MAmarnath   IND         69     4378  42.50     138     11    32     55.69  2.91  4-43     47 ***
BSBedi      IND         67      656   8.99      50      0   266     28.71  2.14  10-194   26 ***
Kapildev    IND        131     5248  31.05     163      8   434     29.65  2.78  11-146   64 ***
```

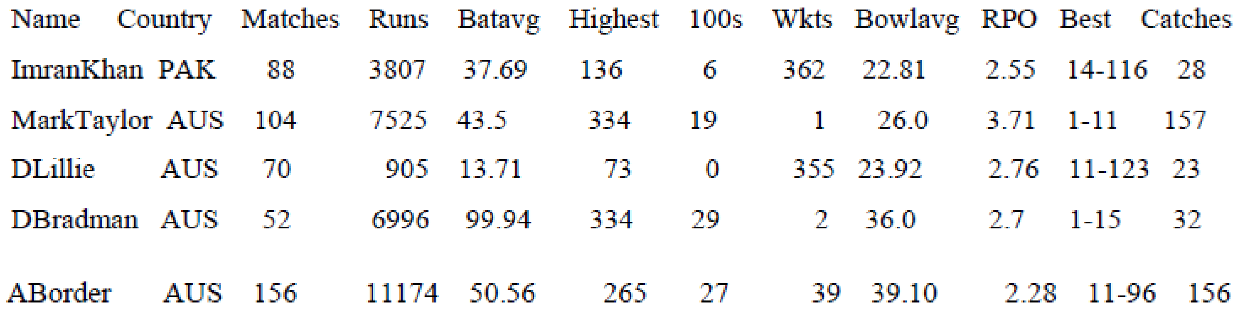
```
AR.Border is the highest scorer with 11174 runs
Kapildev is the highest wicket taker with 434 wickets
MTaylor is the highest catch taker with 157 catches
```

2. Example 2 In this example we use the built-in variable FILENAME and also match a few patterns.

```
# In this example we use FILENAME built in variable and print data from
# first three lines of cricket.data file. In addition we print data from
# ImranKhan to ABorder
BEGIN {FS = "“t" # make the tab as the field separator
printf("%25s “n","First three players")
NR == 1, NR == 5 {print FILENAME ": " $0}
{printf("%12s %5s “n", $1,$2)}
/ImranKhan/, /ABorder/ {num = num + 1; line[num] = $0}
```

```
END
{printf("Player list from Imran Khan to Allen Border “n",
printf("%12s %5s %7s %4s %6s %7s %4s %8s %8s %3s %7s %7s”n”n",
"Name","Country","Matches","Runs","Batavg","Highest","100s","Wkts",
"Bowlavg","Rpo","Best","Catches")}
for(i=1; i <= num; i = i+1)
print(line[i] ) }
bhatt@falerno [AWK] =>!a
awk -f cprg.2 cricket.data
```

```
First three players
cricket.data SGavaskar IND
cricket.data MAmarnath IND
cricket.data BSBedi IND
Player list from Imran Khan to Allen Border
```

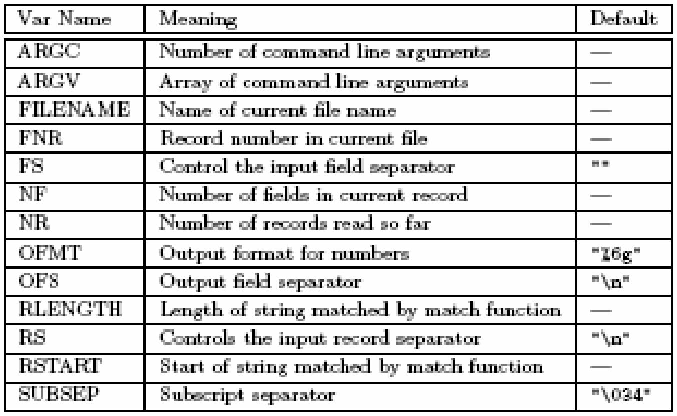


```
Name      Country  Matches  Runs   Batavg  Highest  100s  Wkts  Bowlavg  RPO   Best    Catches
ImranKhan  PAK      88      3807   37.69    136      6    362   22.81    2.55  14-116   28
MarkTaylor  AUS    104      7525   43.5     334     19      1    26.0    3.71  1-11     157
DLillie     AUS     70       905   13.71     73      0    355  23.92     2.76  11-123   23
DBradman    AUS     52      6996   99.94    334     29      2   36.0     2.7   1-15     32

ABorder     AUS    156     11174   50.56    265     27     39   39.10     2.28  11-96   156
```

At this time it may be worth our while to look at some of the list of the built-in variables that are available in AWK. (See Table 12.2).

1.6.2 More on AWK Grammar

Continuing our discussion with some aspects of the AWK grammar, we shall describe the nature of statements and expressions permissible within AWK. The statements



Var Name	Meaning	Default
ARGC	Number of command line arguments	—
ARGV	Array of command line arguments	—
FILENAME	Name of current file name	—
FNR	Record number in current file	—
FS	Control the input field separator	""
NF	Number of fields in current record	—
NR	Number of records read so far	—
OFMT	Output format for numbers	"%6g"
OFS	Output field separator	"\n"
RLENGTH	Length of string matched by match function	—
RS	Controls the input record separator	"\n"
RSTART	Start of string matched by match function	—
SUBSEP	Subscript separator	"\034"

Table 12.2: Built-in variables in AWK.

are essentially invoke actions. In this description, “expression" may be with constants, variables, assignments, or function calls. Essentially, these statements are program actions as described below:

1. print expression-list
2. printf(format, expression-list)
3. if (expression) statement
4. while (expression) statement
5. for (expression; expression; expression) statement
6. for (variable in array) expression note : “in" is a key word
7. do statement while (expression)
8. break: immediately leave the innermost enclosing while loop
9. continue: start next iteration of the innermost enclosing while loop

10 next: start the next iteration of the main input loop

11 exit: go immediately to the END action

12 exit expression: same with return expression as status of program

13 statements

; : is an empty statement

We should indicate the following here:

1. Primary Expressions are: numeric and string constants, variables, fields, function calls, and array elements

2. The following operators help to compose expressions:

(a) assignment operators =;+ =;¡ =; ¤ =; = =; % =;=

(b) conditional operator ?:

(c) logical operators || && !

(d) matching operators ~ and !~

(e) relational operators <;<=;==; ! =;>=

(f) concatenation operator (see the string operator below)

(g) arithmetic +, -, *, /, % ^ unary + and -

(h) incrementing and decrementing operators ++ and -- (pre- as well as post-fix)

(i) parentheses and grouping as usual.

For constructing the expressions the following built-in functions are available:

atan2(y,x)	arctan of y/x in the range of -pi to +pi
cos(x)	cosine of x; x in radians
exp(x)	
int(x)	truncated towards 0 when x > 0
log(x)	natural (base e) log of x
rand(x)	random no. r, where 0<= r < 1
sin(x)	sine of x; x in radians
sqrt(x)	square root of x
srand(x)	x is the new seed for rand()

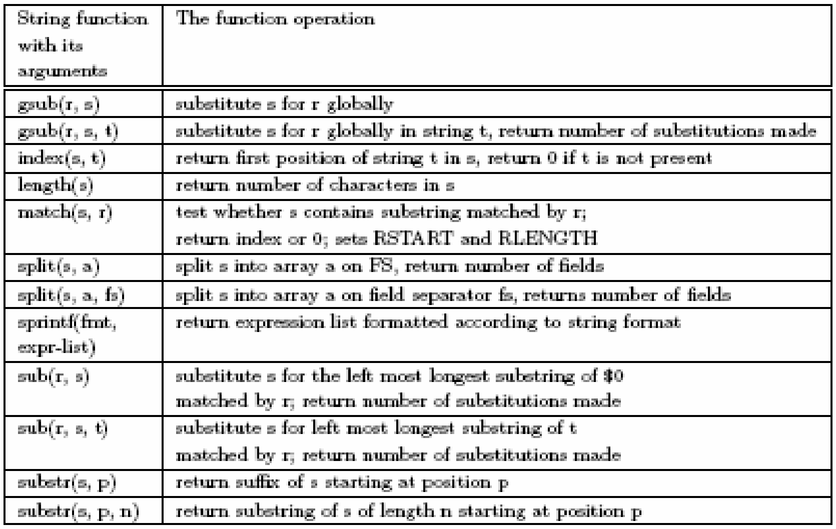
To compute base 10 log use log(x)/log(10)

randint = int(n* rand()) + 1 sets randint to a random no. between 1 and n inclusive

2. **Example 3** We next look at string operations. Let us first construct a small recognizer. The string concatenation operation is rather implicit. String expressions are created by writing constants, vars, fields, array elements, function values and others placed next to each other. The program {print NR ": " $0} concatenates as expected ": " to each line of output. In the example below, we shall use some of these facilities which we have discussed.

```
# this program is a small illustration of building a recognizer
BEGIN {
sign = "[+-]?"
decimal = "[0-9]+[.]?[0-9]*"
fraction = "[.][0-9]+"
exponent ="([eE]" sign "[0-9]+)?"
number ="^" sign "(" decimal "|" fraction ")" exponent "$"
}
$0 ~ number {print}
```

Note that in this example if /pattern/ were to be used then meta-characters would be recognized with an escape sequence, i.e. using “. for . and so on. Run this program using gawk with the data given below:



String function with its arguments	The function operation
gsub(r, s)	substitute s for r globally
gsub(r, s, t)	substitute s for r globally in string t, return number of substitutions made
index(s, t)	return first position of string t in s, return 0 if t is not present
length(s)	return number of characters in s
match(s, r)	test whether s contains substring matched by r; return index or 0; sets RSTART and RLENGTH
split(s, a)	split s into array a on FS, return number of fields
split(s, a, fs)	split s into array a on field separator fs, returns number of fields
sprintf(fmt, expr-list)	return expression list formatted according to string format
sub(r, s)	substitute s for the left most longest substring of $0 matched by r; return number of substitutions made
sub(r, s, t)	substitute s for left most longest substring of t matched by r; return number of substitutions made
substr(s, p)	return suffix of s starting at position p
substr(s, p, n)	return substring of s of length n starting at position p

Table 12.3: Various string function in AWK.

```
1.2e5
129.0
abc
129.0
```

You should see the following as output:

```
bhatt@falerno [AWK] =>gawk -f flt_num_rec.awk flt_num.data
1.2e5
129.0
129.0
```

4. Example 4 Now we shall use some string functions that are available in AWK. We shall match partial strings and also substitute strings in output (like the substitute command in vi editor). AWK supports many string oriented operations. These are listed in Table 12.3.

Let us now suppose we have the following AWK program line:

```
x = sprintf("%10s %6d",$1, $2)
```

This program line will return x in the specified format. Similarly, observe the behaviour of the program segment given below:

```
{x = index("ImranKhan", "Khan"); print "x is :", x}
bhatt@falerno [AWK] =>!a
awk -f dummy.rec
x is : 6
```

In the response above, note that the index on the string begins with 1. Also, if we use *gsub* command it will act like vi substitution command as shown below:

```
{ gsub(/KDev/, "kapildev"); print}.
```

Let us examine the program segment below.

```
BEGIN {OFS = "“t"}
{$1 = substr($1, 1, 4); x = length($0); print $0, x}
{s = s substr($1, 1, 4) " "}
END {print s}
```

Clearly, the output lines would be like the one shown below. We have shown only one line of output.

```
SGav   IND    125    10122    51.12     236  34    1   206.00   3.25 1-34     108              54
....
....
```

We next indicate how we may count the number of centuries scored by Indian players and players from Pakistan.

```
/IND/ {century["India"] += $7 }
/PAK/ {century["Pakistan"] += $7 }
/AUS/ {catches["Australia"] += $12; k = k+1; Aus[k] = $0}
END {print "The Indians have scored ", century["India"], "centuries"
print "The Pakistanis have scored ", century["Pakistan"], "centuries"
print "The Australians have taken ", catches["Australia"], "catches"}
```

The response is:

```
The Indians have scored 53 centuries
The Pakistanis have scored 21 centuries
```

The Australians have taken 368 catches

5. Example 5 Now we shall demonstrate the use of Unix pipe within the AWK program.

This program obtains output and then pipes it to give us a sorted output.

```
# This program demonstrates the use of pipe.
BEGIN{FS = "“t"}
{wickets[$2] += $8}
END {for (c in wickets)
printf("%10s”t%5d”t%10s”n", c, wickets[c], "wickets") | "sort -t'“t' +1rn" }
bhatt@falerno [AWK] =>awk -f recog5.awk cricket.data
IND                 733                 wickets
NZ                   445                wickets
AUS                 397                 wickets
PAK                  367                wickets
WI                  291                 wickets
ENG                 7                   wickets
```

Suppose we have a program line as follows:

```
egrep "BRAZIL" cricket.data | awk 'program'
```

The obvious response is reproduced below :4

Normally, a file or a pipe is created and opened only during the run of a program. If the file, or pipe, is explicitly closed and then reused, it will be reopened. The statement close(expression) closes a file, or pipe, denoted by expression. The string value of expression must be the same as the string used to create the file, or pipe, in the first place.

Close is essential if we write and read on a file, or pipe, alter in the same program. There is always a system defined limit on the number of pipes, or files that a program may open.

One good use of pipes is in organizing input. There are several ways of providing the input data with the most common arrangement being:

```
awk 'program' data
```

AWK reads standard input if no file names are given; thus a second common arrangement is to have another program pipe its output into AWK. For example, *egrep* selects input lines containing a specified regular expression, but does this much faster than AWK does. So, we can type in a command egrep 'IND' countries.data | awk 'program' to get the desired input

```
bhatt@falerno [AWK] =>egrep 'BRAZIL' cricketer.data | awk -f recog3.awk
egrep: NO such file or directory
```

6. Example 6 Now we shall show the use of command line arguments. An AWK command line may have any of the several forms below:

```
awk 'program' f1 f2 ...
awk -f programfile f1 f2 ...
awk -Fsep 'program' f1 f2 ...
awk -Fsep programfile f1 f2 ...
```

If a file name has the form var=text (note no spaces), however, it is treated as an assignment of text to var, performed at the time when that argument would be otherwise a file. This type of assignment allows vars to be changed before and after a file is read.

The command line arguments are available to AWK program in a built-in array called ARGV. The value of ARGC is one more than the number of arguments. With the command line awk -f progfile a v=1 bi, ARGC is 4 and the array ARGV has the following values : ARGV[0] is awk, ARGV[1] is a ARGV[2] is v=1 and finally, ARGV is b. ARGC is one more than the number of arguments as awk is counted as the zeroth argument. Here is another sample program with its response shown:

```
#echo - print command line arguments
BEGIN{ for( i = 1; i < ARGC; i++ )
print("%s , “n", ARGV[i] )}
outputs
```

```
bhatt@falerno [AWK] =>!g
gawk -f cmd_line1.awk cricket.data
cricket.data ,
```

6. **Example - 7** Our final example shows the use of shell scripts. 5 Suppose we wish to have a shell program in file sh1.awk. We shall have to proceed as follows.

- step 1: make the file sh1.awk as gawk '{print $1}' $* .
- step 2: chmod sh1.swk to make it executable.

  ```
  bhatt@falerno [AWK] => chmod +x sh1.awk
  ```

- step 3: Now execute it under the shell command

  ```
  bhatt@falerno [AWK] => sh sh1.awk cricket.data file1.data file2.data
  ```

- step 4: See the result.

```
    bhatt@falerno [AWK] =>sh sh1.awk cricket.data
    SGavaskar
    .......
    .......
MartinCrowe
RHadlee
```

Here is an interesting program that swaps the fields:

```
#field swap bring field 4 to 2; 2 to 3 and 3 to 4
#usage : sh2.awk 1 4 2 3 cricket.data to get the effect
gawk '
BEGIN {for (i = 1; ARGV[i] ~ /^[0-9]+$/; i++) {# collect numbers
fld[++nf] = ARGV[i]
#print " the arg is :", fld[nf]
ARGV[i] = ""}
#print "exited the loop with the value of i : ", i
if (i >= ARGC) #no file names so force stdin
ARGV[ARGC++] = "-"
}
# {print "testing if here"}
{for (i = 1; i <= nf; i++)
#print
printf("%8s", $fld[i])
}
{ print "" }' $*
```

```
bhatt@falerno [AWK] =>!s
sh sh2.awk 1 2 12 3 4 5 6 7 8 9 10 11 cricket.data
SGavaskar IND 108 125 10122 51.12 236 34 1 206.00 3.25 1-34
......
......
ABorder AUS 156 156 11174 50.56 265 27 39 39.10 2.28 11-96
```

In the examples above we have described a very powerful tool. It is hoped that with these examples the reader should feel comfortable with the Unix tools suite.

Unit-4

1.1 Shell Scripts in UNIX

A Shell, as we remarked Block-1 in unit-1 offers a user an interface with the OS kernel. A user obtains OS services through an OS shell. When a user logs in, he has a login shell. The login shell offers the first interface that the user interacts with either as a terminal console, or through a window interface which emulates a terminal. A user needs this command interface to use an OS, and the login shell immediately provides that (as soon as user logs in). Come to think of it, a shell is essentially a process which is a command interpreter!! In this module we shall explore ways in which a shell supports enhancement in a user's productivity.

1.2 Facilities Offered by Unix Shells

In Figure 13.1, we show how a user interacts with any Unix shell. Note that a shell distinguishes between the commands and a request to use a tool. A tool may have its own operational environment. In that case shell hands in the control to another environment. As an example, if a user wishes to use the editor tool vi, then we notice that it has its own states like edit and text mode, etc. In case we have a built-in command then it has a well understood interpretation across all the shells. If it is a command which is particular to a specific shell, then it needs interpretation in the context of a specific shell by interpreting its semantics.

Every Unix shell offers two key facilities. In addition to an interactive command interpreter, it also offers a very useful programming environment. The shell programming environment accepts programs written in shell programming language. In later sections in this chapter we shall learn about the shell programming language and how to make use of it for enhancing productivity. A user obtains maximum gains when he learns not only to automate a sequence of commands but also to make choices within command sequences and invoke repeated executions.

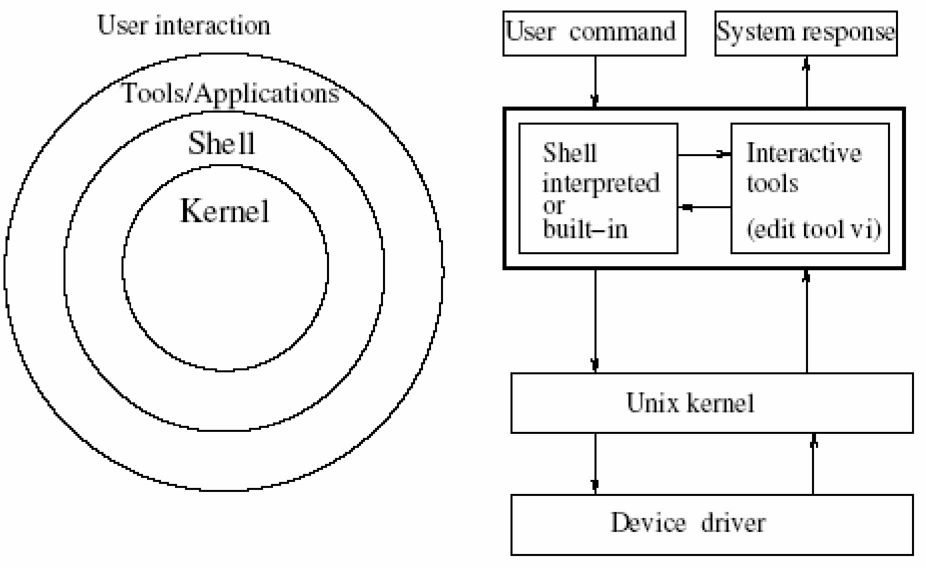


Figure 13.1: A Unix system shell.

1.2.1 The Shell Families

One of the oldest shells is the Bourne shell. In most Unix machines it is available either in its original form or as BASH which expands to Bourne Again Shell. An improvement over the Bourne shell was the Korn shell. To properly enmesh with *c* language programming environment, a new family of shells came along. The first in this family is csh, the c shell. csh has its more recent version as tchs.

The Bourne shell is the primary shell. It has all the primary capabilities. The subsequent shells provided some extensions and some conveniences. Notwithstanding the differences, all shells follow a four-step operational pattern and that is:

1. Read a command line.
2. Parse and interpret it.
3. Invoke the execution of the command line.
4. Go to step 1.

When there are no more command lines, the shell simply awaits one. As a programming environment, a shell programming language allows a user to write a program, often called *a shell script.* Once a script is presented to a shell, it goes back to its four-step operational pattern and takes commands from the script (exactly as it would have done from the terminal). So what is the advantage of the script over the terminal usage? The advantages manifest in the form of automation. One does not have to repeatedly type the commands. One can make a batch of a set of such commands which need to be repeatedly performed and process the batch command script. We can even automate the decision steps to choose amongst alternative paths in command sequence.

In later sections, we shall study the syntax and associated semantics of shell programming language. We shall use Bourne shell as that is almost always available regardless of which of the Unix family OSs is being used.

1.2.2 Subshells

Since we have mentioned so many shells from different families, it is natural to be curious about the shell currently in use. The following Unix command tells us which is our current shell environment:

echo $SHELL

We shall examine the above command for some of the concepts associated with shell programming. The first part is the command *echo* and the second part is the argument $SHELL. The latter, in this case, is an *environmental variable*. First the command “echo". This command asks the shell environment to literally echo something based on the text that follows. We shall make use of this command very frequently in shell scripts either to prompt to ourselves some intermediate message, or show a value of interest.

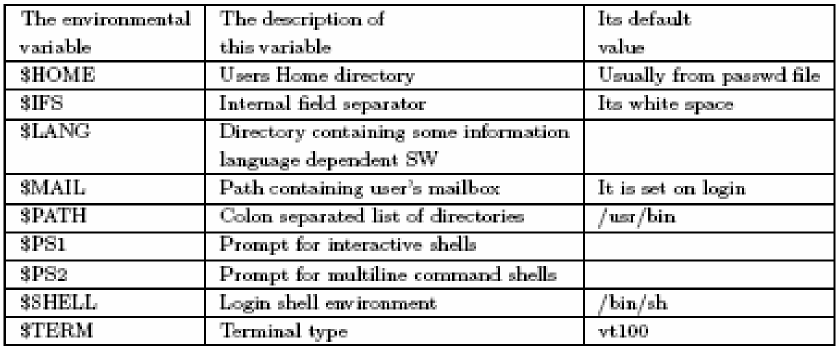
Every OS has some environmental variables. To see the values associated with various environmental variables just give the Unix command set or env.

- set /* shows values of all the environmental variables in use */
- env /* shows values of all the environmental variables in use */

In the response, we should get the value of $SHELL as the name of shell currently in use. This should be the same as the value which our echo command prompted. Next, the second part of the echo command. The $ in $SHELL yields a value. Try giving the echo command without a $ in front of SHELL and you will see that echo promptly responds with SHELL. That then explains the role of the leading $ on variable names. One may have user defined variables, as we shall see later, whose values too can be echoed using a leading $ symbol.

In Table 13.1 we show some typical settings for Bourne shell environmental variables. One may open a new shell within an existing shell. In the new shell, one may define variables or organize a control flow for a sequence of shell commands to be executed.

The variables defined in sub-shells scope to their nested level only. The nesting shell variables may be reassigned within the nested shell for local usage.



The environmental variable	The description of this variable	Its default value
$HOME	Users Home directory	Usually from passwd file
$IFS	Internal field separator	Its white space
$LANG	Directory containing some information language dependent SW	
$MAIL	Path containing user's mailbox	It is set on login
$PATH	Colon separated list of directories	/usr/bin
$PS1	Prompt for interactive shells	
$PS2	Prompt for multiline command shells	
$SHELL	Login shell environment	/bin/sh
$TERM	Terminal type	vt100

Table 13.1: A partial list of environmental variables.

Also, one can use a particular shell of his choice. For instance, suppose we wish to use scripts that are specific to Korn shell. We could enter a Korn shell from the current shell by giving the command ksh. To check the location of a specific shell use:

which shell-name

where *which* is a Unix command and *shell-name* is the name of the shell whose location you wished to know. To use scripts we need to create a file and use it within a new shell environment.

1.3 The Shell Programming Environment

Shell files may be created as text files and used in two different ways. One way is to create a text file with the first line which looks like the following:

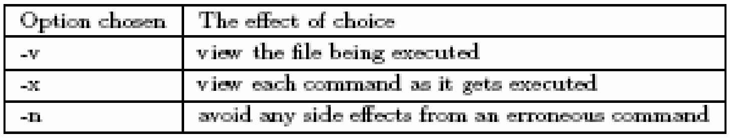
#! /bin/sh

The rest of the file would be the actual shell script. The first line determines which shell shall be used. Also, this text file's mode must be changed to executable by using +x in *chmod command.*

Another way is to create a shell file and use a -f option with the shell command like *ksh -f file_name*, with obvious interpretation. The named shell then uses the file identified as the argument of the script file.

1.4 Some Example Scripts

In this section we shall see some shell scripts with accompanying explanations. It is hoped that these examples would provide some understanding of shell programming. In addition it should help a learner to start using shell scripts to be able to use the system



Option chosen	The effect of choice
-v	view the file being executed
-x	view each command as it gets executed
-n	avoid any side effects from an erroneous command

Table 13.2: The options with their effects.

efficiently in some repetitive programming situations. All examples use the second method discussed above for text files creation and usage as shell scripts. To execute these the pattern of command shall be as follows:

```
sh [options] <file_name> arg1 arg2 ....
```

The options may be [vxn]. The effect of the choice is as shown in Table 13.2.

1.4.1 Example Shell Scripts

We shall first see how a user-defined variable is assigned a value. The assignment may be done within a script or may be accepted as an inline command argument. It is also possible to check out if a variable has, or does not have a definition. It is also possible to generate a suitable message in case a shell variable is not defined.

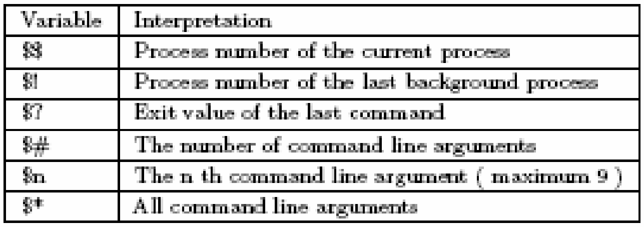
file sh_0.file: We shall begin with some of the simplest shell commands like *echo* and also introduce how a variable may be defined in the context of current shell.

```
# file sh_0.file
echo shellfile is running /* just echos the text following echo */
defineavar=avar /* defines a variable called defineavar */
echo $defineavar /* echos the value of defineavar */
echo "making defineavar readonly now" /* Note the text is quoted */
readonly defineavar
echo "an attempt to reassign defineavar would not succeed"
```

```
defineavar=newvar
```

file sh_1.file One of the interesting things which one can do is to supply parameters to a shell script. This is similar to the command line arguments given to a program written in a high level programming language like c.

```
# file sh_1.file
# For this program give an input like "This is a test case" i.e. 5 parameters
echo we first get the file name
echo $0 /* $0 yields this script file name */
```



Variable	Interpretation
$$	Process number of the current process
$!	Process number of the last background process
$?	Exit value of the last command
$#	The number of command line arguments
$n	The n th command line argument (maximum 9)
$*	All command line arguments

Table 13.3: A partial list of special variables.

```
echo we now get the number of parameters that were input
echo $# /* yields the number of arguments */
echo now we now get the first three parameters that were input
echo $1 $2 $3 /* The first three arguments */
shift /* shift cmd could have arguments */
echo now we now shift and see three parameters that were input
echo $1 $2 $3 /* Three arguments after one shift */
```

A partial list of symbols with their respective meanings is shown in Table 13.3.

```
# file sh_1a.file
# this is a file with only one echo command
echo this line is actually a very long command as its arguments run \
on and on and on Note that both this and the line above and below \
are part of the command Also note how back slash folds commands
```

In most shells, multiple commands may be separated on the same line by a semi-colon (;). In case the command needs to be folded this may be done by simply putting a back slash and carrying on as shown.

file sh_2.file: If a variable is not defined, no value is returned for it. However, one can choose to return an error message and check out if a certain variable has indeed been defined. A user may even generate a suitable message as shown in scripts sh 2a and sh 2b.

```
# file sh_2.file
# This is to find out if a certain parameter has been defined.
echo param is not defined so we should get a null value for param
echo ${param}
echo param was not defined earlier so we got no message.
echo Suppose we now use "?" option. Then we shall get an error message
echo ${param?error}
-----------------------------------------------------------------------
# file sh_2a.file
# This is to find out if a certain parameter has been defined.
echo param is not defined so we should get a null value for param
echo ${param}
# echo param is not defined with "?" option we get the error message
# echo ${param?error}
echo param is not defined with "-" option we get the quoted message
echo ${param-'user generated quoted message'}
-----------------------------------------------------------------------
# file sh_2b.file
# This is to find out if a certain parameter has been defined.
echo param is not defined so we should get a null value for param
echo ${param}
# echo param is not defined with "?" option we get the error message
# echo ${param?error}
echo param is not defined with "=" option we get the quoted message
echo ${param='user generated quoted message'}
```

file sh_3.file: Now we shall see a few scripts that use a text with three kinds of quotes - the double quotes, the single forward quote and the single backward quote. As the examples in files sh 3, sh 3a and sh 3b show, the double quotes evaluates a string within it as it is. The back quotes result in substituting the value of shell variables and we may use variables with a $ sign pre-fixed if we wish to have the string substituted by its value. We show the use of back quotes in this script in a variety of contexts.

```
# file sh_3.file
echo the next line shows command substitution within back quotes
echo I am `whoami` /* every thingin back quotes is evaluated */
echo I am 'whoami' /* nothing in back quotes is evaluated */
echo "I am using $SHELL" /* Variables evaluated in double quotes */
echo today is `date` /* one may mix quotes and messages */
echo there are `who | wc -l` users at the moment /* using a pipe */
echo var a is now assigned the result of echo backquoted whoami
a=`whoami`
echo we shall output its value next
echo $a
echo also let us reassign a with the value for environment var HOME
a=`echo $HOME`
echo $a
echo a double dollar is a special variable that stores process id of the shell
echo $$
echo the shell vars can be used to generate arguments for Unix commands
echo like files in the current directory are
cur_dir=.
ls $cur_dir
echo list the files under directory A
ls $cur_dir/A
----------------------------------------------------------------------
# file sh_3a.file
# In this file we learn to use quotes. There are three types of quotes
# First use of a single quote within which no substitution takes place
a=5
```

```
echo 'Within single quotes value is not substituted i.e $a has a value of $a'
# now we look at the double quote
echo "Within double quotes value is substituted so dollar a has a value of $a"
echo Finally we look at the case of back quotes where everything is evaluated
echo `$a`
echo `a`
echo Now we show how a single character may be quoted using reverse slash
echo back quoted a is \`a and dollar a is \$a
echo quotes are useful in assigning variables values that have spaces
b='my name'
echo value of b is = $b
--------------------------------------------------------------------
# file sh_3b.file
# In this file we shall study the set command. Set lets you
# view shell variable values
echo ---------out put of set -------------
set
echo use printenv to output variables in the environment
echo ---------output of printenv -------------
printenv
--------------------------------------------------------------------
```

file sh_4.file

One of the interesting functions available for use in shell scripts is the \eval" function. The name of the function is a give away. It means \to evaluate". We simply evaluate the arguments. As a function it was first used in functional programming languages. It can be used in a nested manner as well, as we shall demonstrate in file sh_4.file.

```
# file sh_4.file
# this file shows the use of eval function in the shell
b=5
a=\$b
echo a is $a
echo the value of b is $b
eval echo the value of a evaluated from the expression it generates i.e. $a
c=echo
eval $c I am fine
```

```
d=\$c
echo the value of d is $d
eval eval $d I am fine
```

----

file sh_5.file: In the next two files we demonstrate the use of a detached process and also how to invoke a sleep state on a process.

```
# file sh_5.file
# This file shows how we may group a process into a detached process
# by enclosing it in parentheses.
# Also it shows use of sleep command
echo basically we shall sleep for 5 seconds after launching
echo a detached process and then give the date
(sleep 5; date)
```

----

file sh_6.file

```
# file sh_6.file
# Typically << accepts the file till the word that follows
# in the file. In this case the input is taken till
# the word end appears in the file.
#
# This file has the command as well as data in it.
# Run it : as an example : sh_6.file 17 to see him 2217 as output.
# $1 gets the file argument.
grep $1<<end /* grep is the pattern matching command in Unix */
me 2216
him 2217
others 2218
end
```

----

file sh_7.file: Now we first show the use of an if command. We shall check the existence of a .ps file and decide to print it if it exists or else leave a message that the file is not in this directory. The basic pattern of the “if” command is just like in the programming languages. It is:

```
if condition
then

command_pattern_for_true
else
command_pattern_for_false
fi
```

We shall now use this kind of pattern in the program shown below:

```
# file sh_7.file
if ls my_file.ps
then lpr -Pbarolo-dup my_file.ps /* prints on printer barolo on both sides */
else echo "no such file in this directory"
fi
```

Clearly a more general construct is the case and it is used in the next script.

```
# file sh_7a.file
# This file demonstrates use of case
# In particular note the default option and usage of selection
# Note the pattern matching using the regular expression choices.
case $1 in
[0-9]) echo "OK valid input : a digit ";;
[a-z]|[A-Z]) echo "OK valid input : a letter ";;
*) echo "please note only a single digit or a letter is valid as input";;
esac
```

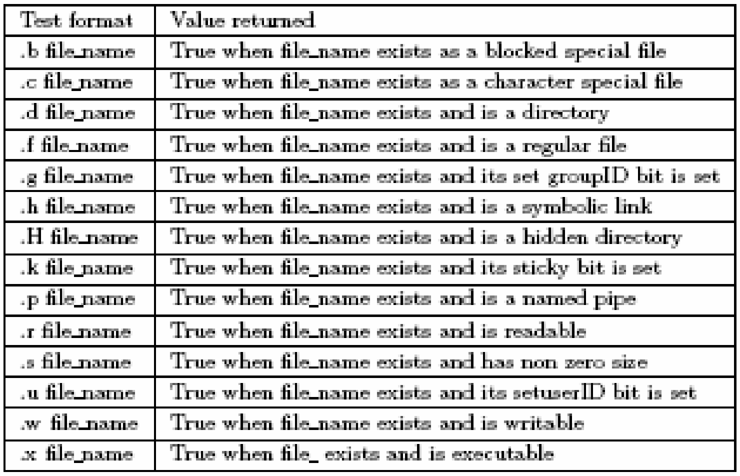
----

file sh_8.file: We shall now look at an iterative structure. Again it is similar to what we use in a programming language. It is:

```
for some_var in the_list
do
the_given_command_set
done /* a do is terminated by done */
```

We shall show a use of the pattern in the example below:

```
# file sh_8.file
# In this file we illustrate use of for command
# It may be a good idea to remove some file called
```



Test format	Value returned
.b file_name	True when file_name exists as a blocked special file
.c file_name	True when file_name exists as a character special file
.d file_name	True when file_name exists and is a directory
.f file_name	True when file_name exists and is a regular file
.g file_name	True when file_name exists and its set groupID bit is set
.h file_name	True when file_name exists and is a symbolic link
.H file_name	True when file_name exists and is a hidden directory
.k file_name	True when file_name exists and its sticky bit is set
.p file_name	True when file_name exists and is a named pipe
.r file_name	True when file_name exists and is readable
.s file_name	True when file_name exists and has non zero size
.u file_name	True when file_name exists and its setuserID bit is set
.w file_name	True when file_name exists and is writable
.x file_name	True when file_ exists and is executable

Table 13.4: A list of test options.

```
# dummy in the current directory as a first step.
#echo removing dummy
rm dummy
for i in `ls`; do echo $i >> dummy; done
grep test dummy
```

File tests may be much more complex compared to the ones shown in the previous example. There we checked for only the existence of the file. In Table 13.4 we show some test options.

In the context of use of test, one may perform tests on strings as well. The table below lists some of the possibilities.

Test Operation	Returned value
------------	--------------
str1 = str2	True when strs are equivalent
str != str2	True when not equivalent
-1 str	True when str has length 0
-n str	True when str has nonzero length
string	True when NOT the null string.

In addition to for there are while and until iterators which also have their do and done.

These two patterns are shown next.

```
while condition
do
command_set

done
```

Alternatively, we may use until with obvious semantics.

```
until condition
do
          command_set
done
```

A simple script using until may be like the one given below.

```
count=2
until [ $count -le 0 ]
do
          lpr -Pbarolo-dup file$count /* prints a file with suffix = count */
          count=`expr $count - 1`
done
```

Note that one may nest these commands, i.e. there may be a until within a while or if or case.

file sh_9.file: Now we shall demonstrate the use of expr command. This command offers an opportunity to use integer arithmetic as shown below.

```
b=3
echo value of b is = $b
echo we shall use as the value of b to get the values for a
echo on adding two we get
a=`expr $b + 2`
echo $a

----------------------------------------------------------------------
```

file sh 9a.file We shall combine the use of test along with expr. The values of test may be *true or false* and these may be combined to form relational expressions which finally yield a logical value.

```
# file sh_9a.file
# this file illustrates the use of expr and test commands
b=3
echo on adding two we get

a=`expr $b + 2`
echo $a
echo on multiplying two we get
a=`expr $b \* 2` /* Note the back slash preceding star */
# We shall see the reason for using back slash before star in the next example
echo $a
test $a -gt 100
$?
test $a -lt 100
$?
test $a -eq 6
$?
test $a = 6
$?
test $a -le 6
$?
test $a -ge 6

$?
```

```
test $a = 5
$?
if (test $a = 5)
then echo "found equal to 5"
else echo "found not equal to 5"
fi
test $a = 6
if (test $a = 6)
then echo "the previous test was successful"
fi
----------------------------------------------------------------------
```

file sh 10.file Now we shall use some regular expressions commonly used with file names.

```
# file sh_10.file
# in this program we identify directories in the current directory
echo "listing all the directories first"
for i in *
do
if test -d $i
then echo "$i is a directory"
fi
done
echo "Now listing the files"
for i in *
do
if test -f $i
then
echo "$i is a file"
fi
done
echo "finally the shell files are"
ls | grep sh_
----------------------------------------------------------------------
```

file sh 11.file

```
# file sh_11.file
# In this file we learn about the trap command. We will first
# create many files with different names. Later we will remove
# some of these by explicitly trapping
touch rmf1
touch keep1
touch rmf2
touch rmf3
touch keep2
touch rmf4
touch keep3
echo "The files now are"
ls rmf*
ls keep*
trap `rm rm*; exit` 1 2 3 9 15
echo "The files now are"
ls rmf*
ls keep*
```

----

file sh_12:.file Now we assume the presence of files of telephone numbers. Also, we demonstrate how Unix utilities can be used within the shell scripts.

```
# file sh_12.file
# In this file we invoke a sort command and see its effect on a file
# Also note how we have used input and output on the same line of cmd.
sort < telNos > stelNos
# We can also use a translate cmd to get translation from lower to upper case
tr a-z A-Z < telNos > ctelNos
```

In this unit we saw the manner in which a user may use Unix shell and facilities offered by it. As we had earlier remarked, much of Unix is basically a shell and a kernel.

1.5 Unix Kernel Architecture

The kernel runs the show, i.e. it manages all the operations in a Unix flavored environment. The kernel architecture must support the primary Unix requirements. These requirements fall in two categories namely, functions for process management and functions for file management (files include device files). Process management entails allocation of resources including CPU, memory, and offers services that processes may need. The file management in itself involves handling all the files required by processes, communication with device drives and regulating transmission of data to and from peripherals. The kernel operation gives the user processes a feel of synchronous operation, hiding all underlying asynchronism in peripheral and hardware operations (like the time slicing by clock). In summary, we can say that the kernel handles the following operations:

1. It is responsible for scheduling running of user and other processes.
2. It is responsible for allocating memory.
3. It is responsible for managing the swapping between memory and disk.
4. It is responsible for moving data to and from the peripherals.
5. it receives service requests from the processes and honors them.

All these services are provided by the kernel through a call to a system utility. As a result, kernel by itself is rather a small program that just maintains enough data structures to pass arguments, receive the results from a call and then pass them on to the calling process. Most of the data structure is tables. The chore of management involves keeping the tables updated. Implementing such a software architecture in actual lines of code would be very small. The order of code for kernel is only 10000 lines of C and 1000 lines of assembly code.

Kernel also aids in carrying out system generation which ensures that Unix is aware of all the peripherals and resources in its environment. For instance, when a new disk is attached, right from its formatting to mounting it within the file system is a part of system generation.

1.6 User Mode and Kernel Mode

At any one time we have one process engaging the CPU. This may be a user process or a system routine (like ls, chmod) that is providing a service. A CPU always engages a process which is \runnable". It is the task of the scheduler to choose amongst the runnable processes and give it the control of CPU to execute its instructions. Upon being scheduled to run, the process is marked now to be in \running" state (from the previous runnable state).

Suppose we trace the operation of a user process. At some point in time it may be executing user instructions. This is the user mode of operation. Suppose, later in the sequence, it seeks to get some data from a peripheral. In that event it would need to make a system call and switch to kernel mode.

The following three situations result in switching to kernel mode from user mode of operation:

1. The scheduler allocates a user process a slice of time (about 0.1 second) and then system clock interrupts. This entails storage of the currently running process status and selecting another runnable process to execute. This switching is done in kernel mode. A point that ought to be noted is: on being switched the current process's priority is re-evaluated (usually lowered). The Unix priorities are ordered in decreasing order as follows:

    - HW errors
    - Clock interrupt
    - Disk I/O
    - Keyboard
    - SW traps and interrupts

2. Services are provided by kernel by switching to the kernel mode. So if a user program needs a service (such as print service, or access to another file for some data) the operation switches to the kernel mode. If the user is seeking a peripheral transfer like reading a data from keyboard, the scheduler puts the currently running process to “sleep" mode.
3. Suppose a user process had sought a data and the peripheral is now ready to provide the data, then the process interrupts. The hardware interrupts are handled by switching to the kernel mode. In other words, the kernel acts as the via-media between all the processes and the hardware as depicted in Figure 14.1.

The following are typical system calls in Unix:
Intent of a process The C function call

- Open a file open
- Close a file close

- Perform I/O read/write
- Send a signal kill (actually there are several signals)
- Create a pipe pipe
- Create a socket socket
- Duplicate a process fork
- Overlay a process exec
- Terminate a process exit

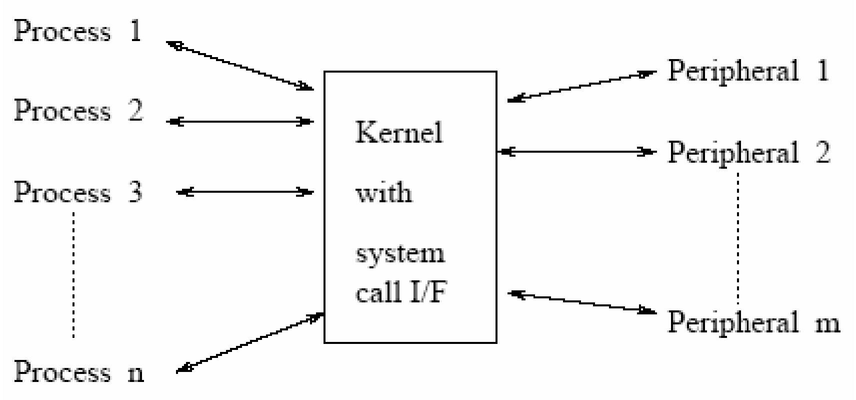


Figure 14.1: The kernel interface.

1.7 System Calls

The kernel lies between the underlying hardware and other high level processes. So basically there are two interfaces: one between the hardware and kernel and the other between the kernel and other high level processes. System calls provide the latter interface.

System call interacts with the kernel employing a syscall vector. In this vector each system call vector has a fixed position. Note that each version of Unix may individually differ in the way they organize the syscall vector. The fork() is a system call. It would be using a format as shown below.

syscall fork-number

Clearly, the fork-number is the position for fork in the syscall vector. All system calls are executed in the kernel mode. Typically, Unix kernels execute the following secure seven steps on a system call:

1. Arguments (if present) for the system call are determined.
2. Arguments (if present) for the system call are pushed in a stack.
3. The state of calling process is saved in a user structure.

4. The process switches to kernel mode.
5. The syscall vector is used as an interface to the kernel routine.
6. The kernel initiates the services routine and a return value is obtained from the kernel service routine.
7. The return value is converted to a c version (usually an integer or a long integer). The value is returned to process which initiated the call. The system also logs the userid of the process that initiated that call.

1.7.1 An Example of a System Call

Let us trace the sequence when a system call to open a file occurs.

- User process executes a syscall open a file.
- User process links to a c runtime library for open and sets up the needed parameters in registers.
- A SW trap is executed now and the operation switches to the kernel mode.
  - The kernel looks up the syscall vector to call "open"
  - The kernel tables are modified to open the file.
  - Return to the point of call with exit status.
- Return to the user process with value and status.
- The user process may resume now with modified status on file or abort on error with exit status.

1.8 Process States in Unix

Unix has the following process state transitions:

- idle ----> runnable -----> running.
- running ----> sleep (usually when a process seeks an event like I/O, it sleeps awaiting event completion).
- running ----> suspended (suspended on a signal).
- running ----> Zombie (process has terminate but has yet to return to its exit code to parent. In

unix every process reports its exit status to the parent.)

- sleeping ---> runnable
- suspended---> runnable

Note that it is the sleep operation which gives the user process an illusion of synchronous operation. The Unix notion of suspended on a signal gives a very efficient mechanism for process to respond to awaited signals in inter-process communications.

1.9 Kernel Operations

The Unix kernel is a main memory resident \process". It has an entry in the process table and has its own data area in the primary memory. Kernel, like any other process, can also use devices on the systems and run processes. Kernel differs from a user process in three major ways.

1. The first major key difference between the kernel process and other processes lies in the fact that kernel also maintains the needed data-structures on behalf of Unix. Kernel maintains most of this data-structure in the main memory itself. The OS based paging or segmentation cannot swap these data structures in or out of the main memory.
2. Another way the kernel differs from the user processes is that it can access the scheduler. Kernel also maintains a disk cache, basically a buffer, which is synchronized ever so often (usually every 30 seconds) to maintain disk file consistency. During this period all the other processes except kernel are suspended.
3. Finally, kernel can also issue signals to kill any process (like a process parent can send a signal to child to abort). Also, no other process can abort kernel.

A fundamental data structure in main memory is page table which maps pages in virtual address space to the main memory. Typically, a page table entry may have the following information.

1. The page mapping as a page frame number, i.e. which disk area it mirrors.
2. The date page was created.
3. Page protection bit for read/write protections.
4. Bits to indicate if the page was modified following the last read.
5. The current page address validity (vis-a-vis the disk).

Usually the page table area information is stored in files such as immu.h or vmmac.h. Processes operating in \user mode" cannot access the page table. At best they can obtain a copy of the page table. They cannot write into page table. This can be done only when they are operating in kernel mode.

User processes use the following areas :

- Code area: Contains the executable code.
- Data area: Contains the static data used by the process.
- Stack area: Usually contains temporary storages needed by the process.
- User area : Stores the housekeeping data.
- Page tables : Used for memory management and accessed by kernel.

The memory is often divided into four quadrants as shown in Figure 14.2. The vertical line shows division between the user and the kernel space. The horizontal line shows the swappable and memory resident division. Some Unix versions hold a higher level data structure in the form of region table entries. A region table stores the following information.

- Pointers to i-nodes of files in the region.
- The type of region (the four kinds of files in Unix).
- Region size.
- Pointers to page tables that store the region.
- Bit indicating if the region is locked.
- The process numbers currently accessing the region.

Generally, the above information is stored in region.h files. The relevant flags that help manage the region are:

1. RT_UNUSED : Region not being used
2. RT_PRIVATE: Only private access permitted, i.e. non-shared region
3. RT_STEXT: Shared text region
4. RT_SHMEM: Shared memory region

The types of regions that can be attached to a process with their definitions are as follows:

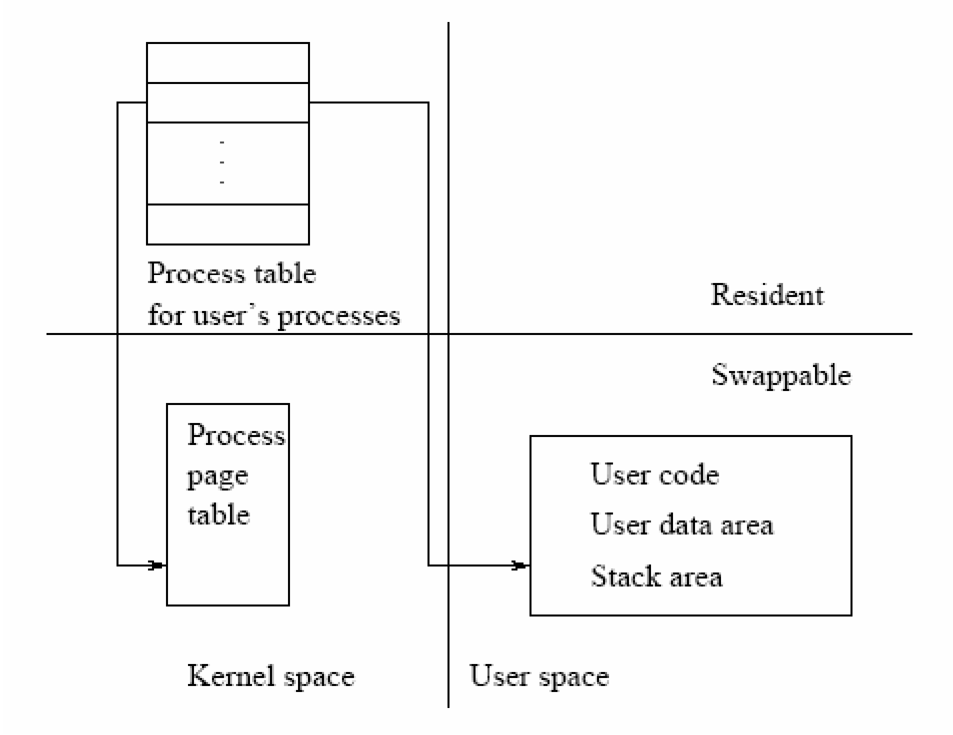


Figure 14.2: User and kernel space.

1. PT_TEXT: Text region
2. PT_DATA: Data region
3. PT_STACK: Stack region
4. PT_SHMEM: Shared memory region
5. PT_DMM: Double mapped memory
6. PT_LIBTEXT: Shared library text region
7. PT_LIBDAT: Shared library data region

In a region-based system, lower-level functions that are needed to be able to use the regions are as follows:

1. *allocreg(): To allocate a region
2. freereg(): To free a region
3. *attachreg(): Attach region to the process
4. *detachreg(): Detach region from the process
5. *dupreg(): Duplicate region in a fork
6. growreg(): Increase the size of region
7. findreg(): Find from virtual address

8. chprot(): Change protection for the region

9. reginit(): Initialise the region table

The functions defined above are available to kernel only. These are useful as the virtual memory space of the user processes needs regions for text, data and stack. The above function calls cannot be made in user mode. (If they were available in user mode they would be system calls. These are not system calls.)

1.10 The Scheduler

Most Unix schedulers follow the rules given below for scheduling:

1. Usually a scheduler reevaluates the process priorities at 1 second interval. The system maintains queues for each priority level.

2. Every tenth of a second the scheduler selects the topmost process in the runnable queue with the highest priority.
3. If a process is runnable at the end of its allocated time, it joins the tail of the queue maintained for its priority level.

4. If a process is put to sleep awaiting an event, then the scheduler allocates the processor to another process.
5. If a process awaiting an event returns from a system call within its allocated time interval but there is a runnable process with a higher priority then the process is interrupted and higher priority, process is allocated the CPU.
6. Periodic clock interrupts occur at the rate of 100 interruptions per second. The clock is updated for a tick and process priority of a running process is decremented after a count of 4 ticks. The priority is calculated as follows:

   priority = (CPU quantum used recently)/(a constant) + (base priority) + (the nice setting).

   Usually the priority diminishes as the CPU quantum rises during the window of time allocated. As a consequence compute intensive processes are penalised and processes with I/O intensive activity enjoy higher priorities.

In brief:

- At every clock tick: Add 1 to clock, recalculate process priority if it is the 4th clock tick for this process.
- At every 1/10th of the second: Select process from the highest priority queue of runnable processes.
- Run till:

  1. End of the allocated time slot or
  2. It sleeps or
  3. It returns from a system call and a higher priority process is ready to run.

- If it is still ready to run: It is in runnable state, so place it at the end of process queue with the same priority level.
- At the end of 1 second: Recalculate the priorities of all processes.

Having seen the Unix Kernels' structure and operation, let us look at Linux.

1.11 Linux Kernel

The Linux environment operates with a framework having four major parts. These sections are: the hardware controllers, the kernel, the OS services and user applications. The kernel as such has a scheduler. For specialised functions, the scheduler makes a call to an appropriate utility. For instance, for context switching there is an assembly program function which the scheduler calls. Since Linux is available as open source we can get the details of the kernel internals function. For instance, the source description of the scheduler is typically located at /usr/src/linux/sched.c. This program regulates the access of processes to the processor. The memory management module source is at /usr/src/linux/mm. This module supports virtual memory operations.

The VFS or virtual file system takes into a variety of file formats. The VFS is a kind of virtual file interface for communication with the devices. The program at /usr/src/linux/net provides the basic communication interface for the net. Finally, the programs at /usr/src/linux/ipc define the range of inter-process communication capabilities. The net and VFS require device drivers usually found at /usr/src/linux/drivers. The architecture x dependent code, essentially, assembly code is found at /usr/src/linux/arch and /usr/src/linux/include. Between them these define and offer all the configuration information which one may need. For instance, one can locate architecture specific files for processors like i386 and others. Lastly, as mentioned earlier, the Linux kernel uses some utilities written in assembly language instructions to do the process switching.

1.11.1 Linux Sources and URLs

Linux, as we stated earlier, is an open source. The following URLs offer a great deal of information about Linux.

1. www.linux.org The most sought after site for Linux. It gives all the user driven and developed information on Linux.
2. sunshine.unc.edu/mdw/linux.html The home page of linux documentation.
3. www.cl.cam.ac.uk/users/iwj10/linux-faq A very good site on frequently asked questions on Linux.
4. www.XFree86.org The X windows in Linux comes from XFree.
5. www.arm.uk.org/rmk/armlinux.html A non-PC architecture Linux home page.
6. www.maclinux-m68k.org An Apple computer, Motorola home page for Linux.
7. The following three URLS are for linux administration, open source gnu c compiler and Linux kernel. linux.dev.admin linux.dev.gcc linux.dev.kernel

Now that we have studied Unix kernel, we shall reinforce the tools track. The next module shall deal with make tool and its use in large projects.

Block-4

Unit-1

1.1 Make Tool In UNIX

In Unix environment, Make is both a productivity enhancement utility as well as a program management tool. It is particularly useful in managing the development of largesized programs. A program may be considered to be large sized when it uses a very large number of functions or it may involve a large number of developers. It is more often the case that different teams of programmers are engaged in developing different functions or sub-parts of a large programming project. In most large-sized projects, one often requires that some common definitions hold across all the functions under development. This is more often the case and is true regardless of the number of persons involved in the development (of large programming project). Clearly, all the teams must use and interpret all common definitions consistently.

In this unit we shall discuss the facilities which make tool makes available. Make tool not only facilitates consistent usage of definitions in the large program development context, but also helps to avoid wasteful re-compilations. Thus it enhances the productivity of individuals as well as that of teams. As we shall, see it is also useful in the context of software installations.

1.2 When to Use Make

Let us examine a commonly occurring scenario in a c program development effort involving a large team of programmers. They all may be sharing a common (.h) file which may have some commonly used data definitions. In case each member of such a team has his own copy of (.h) file, then it would be very difficult to ensure consistency.

Not every member may compile his program with a consistent and current data definitions. The problem becomes even more acute when, within a set of files, there are frequent updates.

Let us illustrate how problems may arise in yet another scenario. For instance, suppose some project implementation involves generating a large user-defined library for later use and subsequent integration. Such a library may evolve over time. Some groups may offer updated or new library modules and definitions for use by the rest of the project group pretty regularly. Consistent use of a new and evolving library is imperative in such a project. If different team members use inconsistent definitions or interpretations, then this can potentially have disastrous consequences.

Note that both these scenarios correspond to large-sized applications development environment. In such cases it is imperative that we maintain consistency of all the definitions which may be spread over several files. Also, such development scenarios often require very frequent re-compilations.

*Mak*e or *make* helps in the management of programs that spread over several files. The need arises in the context of programming projects using any high level programming language or even a book writing project in TEX. Mainly changes in the definitions like (.h) files or modification of (.c) files and libraries of user-defined functions require to be linked to generate a new executable. Large programs and projects require very frequent re-compilations. Make can be also be used in a Unix context where tasks can be expressed as Unix shell commands to account for certain forms of dependencies amongst files. In these situation the use of make is advised.

Make is also useful in installation of new software. In almost all new installations, one needs to relate with the present software configuration and from it, derive the interpretations. This helps to determine what definitions the new software should assume.

Towards the end of this chapter we will briefly describe how installations software use a mastermakefile. They essentially use several minimakefiles which helps in generating the appropriate installation configuration. Both Windows and Unix offer their makefile versions. Most of the discussion would apply to both these environments 1.

1.3 How Make Works

Make avoids unnecessary compiling. It checks recency of an object code relative to its source. Suppose a certain source file has been modified but is not re-compiled. Make would find that the compiled file is now older than the source. So, the dependency of the object on the source will ensure re-compilation of source to generate new object file.

To ensure a consistency, make uses a rule-based inferencing system to determine the actions needed. It checks on object code dependencies on sources by comparing its recency with regards to time of their generation or modification. Actions in command lines ensure consistency using rules of dependence. Any time a (.c) or a (.h) file is modified, all the dependent objects are recreated.

Now we can state how make works: make ensures that a file, which is dependent on its input files, is consistent with the latest definitions prevailing in the input files. This also ensures that in the ensuing compilation, linking can be fully automated by Make. Thus, it helps in avoiding typing out long error prone command sequences. Even the clean-up process following re-compilation can be automated as we will see later.

Makefile Structure: The basic structure of make file is a sequence of targets, dependencies, and commands as shown below:

```
--------------------------------------------------------------
| <TARGET> : SET of DEPENDENCIES /* Inline comments */|
| <TAB> command /* Note that command follows a tab */ |
| <TAB> command /* There may be many command lines */ |
| . |
| . |
| <TAB> command /* There may be many command lines */ |
--------------------------------------------------------------
```

The box above has one instance of a target with its dependencies paired with a possible set of commands in sequence. *A makefile is a file of such paired sequences.*

The box above defines one rule in a makefile. A makefile may have several such rules. Each rule identifies a target and its dependencies. Note that every single rule defines a direct dependency. However, it is possible that there are nested dependencies, i.e. “a" depends on “b" and “b" depends on “c". Dependencies are transitive and the dependency of “a" on “c" is indirect (or we may say implied). Should any of the dependencies undergo a modification, the reconstruction of the target is imperative. This is achieved automatically upon execution of makefiles.

Nesting of dependencies happens when a certain dependency is a sub-target. The makefile is then a description of a tree of sub-targets where the leaves are the elementary files like (.c) or (.h) files or libraries which may be getting updated ever so often. We shall use a one-level tree description for our first example. In subsequent examples, we shall deal with multi-level target trees.

As a simple case, consider the following “helloworld" program. We will demonstrate the use of a make file through this simple example.

Step1: Create a program file helloWorld.c as shown below:

```
#include<stdio.h>
#include<ctype.h>
main
{
printf("HelloWorld \n");
}
```

Step2: Prepare a file called “Makefile" as follows:

```
# This may be a comment
hello: helloWorld.c
cc -o hello helloWorld.c
# this is all in this make file
```

Step3: Now give a command as follows:

```
make
```

Step4: Execute helloWorld to see the result.

To see the effect of make, first repeat the command make and note how make responds to indicate that files are up to date needing no re-compilation. Now modify the program.

Let us change the statement in printf to:

```
printf("helloWorld here I come ! \n")
```

Now execute make again.

One can choose a name different from Makefile. However, in that case use:

```
make -f given_file_name.
```

To force make to re-compile a certain file one can simply update its time by a Unix touch command as given below:

touch <filename> /* this updates its recent modification time */

Make command has the following other useful options:

- -n option: With this option, make goes through all the commands in makefile without executing any of them. Such an option is useful to just check out the makefile itself.
- -p option: With this option, make prints all the macros and targets as it progresses.
- -s option: With this option, make is silent. Essentially, it is the opposite of –p option.
- -i option : With this option make ignores any errors or exceptions which are thrown up. This is indeed very helpful in a development environment. During development, sometimes one needs to focus on a certain part of a program. One may have to do this with several modules in place. In these situations, one often wishes to ignore some obvious exceptions. This ensures that one is not bogged down in reaching the point of test which is in focus at the moment.

Another Example: This time around we shall consider a two-level target definition. Suppose our executable \fin" depends upon two object files (a.o) and (b.o) as shown in Figure 15.1. We may further have (a.o) depend upon (a.cc) and (x.h) and (y.h). Similarly, (b.o) may depend upon (b.cc, z.h) and (y.h). In that case our dependencies would be as shown in Figure 15.1. These dependencies dictate the following:

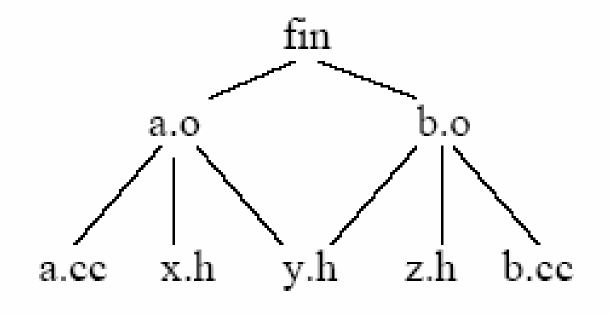
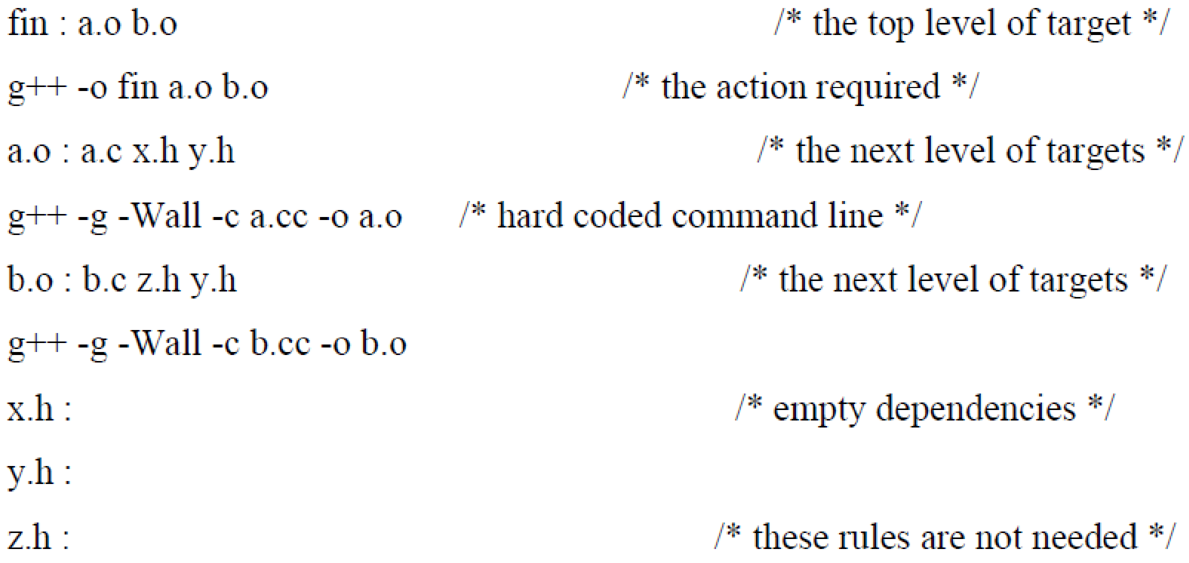


Figure 15.1: Dependencies amongst files.

- A change in x.h will result in re-compiling to get a new a.o and fin.
- A change in y.h will result in re-compiling to get a new a.o, b.o, and fin.
- A change in z.h will result in re-compiling to get a new b.o and fin.
- A change in a.cc will result in re-compiling to get a new a.o and fin.
- A change in b.cc will result in re-compiling to get a new b.o and fin.

Assuming we are using a gnu g++ compiler, we shall then have a make file with the following rules:



```
fin : a.o b.o                                   /* the top level of target */
g++ -o fin a.o b.o                      /* the action required */
a.o : a.c x.h y.h                               /* the next level of targets */
g++ -g -Wall -c a.cc -o a.o    /* hard coded command line */
b.o : b.c z.h y.h                               /* the next level of targets */
g++ -g -Wall -c b.cc -o b.o
x.h :                                           /* empty dependencies */
y.h :
z.h :                                           /* these rules are not needed */
```

The bottom three lines in the example have empty dependencies. These are also referred to as pseudo targets. This make file clearly brings out the two levels of dependencies.

Also, all the command lines in this example use hard-coded commands. Hard-coded commands are specific commands which are relevant only in that programming environment. For instance, all commands here are relevant in the g++ context only. To make *make* files more portable, we shall have to use generic symbols. Each such symbol then can be assigned specific values by using macros as we shall see a little later.

Linking with Libraries: In Unix and c program development environments it is a common practice to let users develop there own libraries. This helps in creating a customized computing environment. In fact, come to think of it, the X series of graphics package is nothing but a set of libraries developed by a team of programmers. These are now available to support our windows and graphics packages.

In this example we shall think of our final executable to be fin which depends upon a.o which in turn depends upon x.h, y.h and as in the last example. The library we shall linkup with is defined as thelib and is generated by a program thelib.cc using definitions from thelib.h. We shall make use of comments to elaborate upon and explain the purpose of various make lines.

```
#
# fin depends on a.o and a library libthelib.a
#
```

```
fin : a.o libthelib.a
g++ -o fin a.o -L -lthelib
#
# a.o depends on three files a.c, x.h and y.h
# The Wall option is a very thorough checking option
# available under g++
a.o : a.cc x.h y.h
g++ -g -Wall -c a.cc
#
# Now the empty action lines.

#
x.h :
y.h :
thelib.h :

#
# Now the rules to rebuild the library libthelib.a
#
libthelib.a thelib.o
ar rv libthelib.a thelib.o
thelib.o: thelib.cc thelib.h
g++ -g -Wall thelib.cc
# end of make file
```

If we run this make file we should expect to see the following sequence of actions

1. g++ -c a.cc
2. cc -c thelib.cc
3. ar rv libthelib.a thelib.o
4. a - thelib.o
5. g++ -o fin a.o -L -lthelib

1.4 Macros, Abstractions, and Shortcuts

We have now seen three make files. Let us examine some aspects of these files.

1. If we wish to use a c compiler we need to use the cc cmd. However, if we shift to MS environment we shall have to use cl in place of cc. If we use the Borland compiler then we need to use bcc. We, therefore, need to modify make files.

In addition, there are many repeated forms. For instance, the compiler used is always gnu (g++) compiler.

Both of the factors above can be handled by using variables. We could define a variable, say CC, which could be assigned the appropriate value like cl, cc, g++, or bcc depending on which environment is being used.

Variables in make are defined and interpreted exactly as in shell scripts. For instance we could define a variable as follows:

```
CC = g++ /* this is a variable definition */
# the definition extends up to new line character or
# up the beginning of the inline comment
```

In fact almost all environments support CC macro which expands to appropriate compiler command, i.e., it expands to cc in Unix, cl in MS and bcc for Borland. A typical usage in a command is shown below:

```
$(CC) -o p p.c /* here p refers to an executable */
```

Note that an in-built or a user-defined macro is used as $(defined Macro). Note that the characters $ and () are required. Such a definition of a macro helps in porting make files across the platforms. We may define CC also as a user defined macro.

2. We should notice some typical patterns used to generate targets. These may require some intermediate target which itself is further dependent on some sources. Consequently, many file-name stems are often repeated in the make file.

In addition, we have also seen repeated use of the compilation options as in -g -Wall. One way to avoid having to make mistakes during the preparation of make files is to use user defined macros to capture both file name stems and flag patterns.

Let us look at flags first. Suppose we have a set of flags for c compilation. We may
define a new macro as shown below :

```
CFLAGS = -o -L –lthelib
```

These are now captured as follows:

```
$(CC) $(CFLAGS) p.c
```

Another typical usage is when we have targets that require many objects as in the example below:

```
t : p1.o p2.o p3.o /* here t is the target and pi the program stems */
```

We can now define macros as follows:

```
TARGET = t
OBJS = p1.o p2.o p3.o
$(TARGET): $(OBJS)
```

Now let us address the issue with file name or program name stems. Often we have a situation in which a target like p.o is obtained by compiling p.c, i.e., we have the stem, (i.e. the part of string

without extension) repeated for source. All systems allow the use of a macro ($*) to get the stem. The target itself is denoted by macro
($@).

So if we have a situation as follows:

```
target : several objects
cc cflags target target_stem.c
```

This can be encoded as follows :

```
$(TARGET): $(OBJS)
$(CC) $(CFLAGS) $@ $*.c
```

1.5 Inference Rules in Make

The use of macros results in literal substitutions. However, it also gives a sense of generalization for use of a rule pattern. This is what precisely an inference rule does. In an inference rule we define the basic dependency and invoke it repeatedly to get the desired effect. For instance, we need a.c file to get an object or a.tex file to get a.dvi file.

These are established patterns and can be encoded as inference rules for make files.

An inference rule begins with a (.) (period) symbol and establishes the relationship. So a .c.o means we need an a.c file for generating the a.o file and it may appear as follows:

```
.c.o:
        $(CC) $(CFLAGS) $*.c
```

In fact because the name stems of .c and o. file is to be the same we can use a built-in macro ($<) to code the above set of lines as follows:

```
.c.o:
        $(CC) $(CFLAGS) $<
```

1.6 Some Additional Options

make allows use of some convenient options. For instance, if we wish to extend the range of options for the file extensions then we define a suffix rule as follows:

```
.SUFFIXES .tex .in
```

This allows us to use the file extensions .tex and .in in our makefile. Suppose we wish to use only a certain selected set of extensions. This can be done as follows

```
.SUFFIXES # this line erases all extension options
```

```
.SUFFIXES .in .out # this adds the selections .in and .out
```

Suppose we do not wish to see the enlisted sequences of makefile outputs then we may use a rule as follows :

```
.SILENT (also make -s as an option)
```

Suppose we wish that makefile on run should not abort when an error occurs, then we can choose an option -i or IGNORE rule.

```
.IGNORE (or use make -i as an option)
```

Use make -p to elicit information on the macros used in makefile.

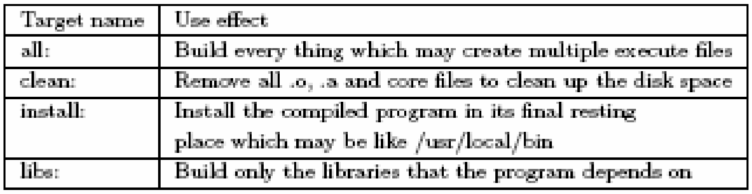
We may on occasions like to use a makefile but may wish to get rid of all the intermediate outputs. This can be done by using pseudo-targets as shown below.

```
cleanup: # absence of dependency means always true
rm *.o # remove the .o files
rm *.k # remove some other files that are not needed
```

As the case above shows we may have more than one action when multiple actions are required to be taken.

In case a command line in a make file is very long and spills beyond the terminal window width, then we can extend it to continue on the next line. This is done by using \ (backslash) at the end. This also is useful to make some statement more explicitly readable as shown below.

```
p.o : p.c \
p.h
```



Target name	Use effect
all:	Build every thing which may create multiple execute files
clean:	Remove all .o, .a and core files to clean up the disk space
install:	Install the compiled program in its final resting place which may be like /usr/local/bin
libs:	Build only the libraries that the program depends on

Table 15.1: Use of conventions.

Gnumake: Amongst the facilities, gnu make allows us to include files. This is useful in the context of defining a set of make variables in one file for an entire project and include it when needed. The include statement is used as follows:

```
include project.mk common_vars.mk other_files.mk
```

Some Standard Conventions: Over the years some conventions have emerged in using make files. These identify some often used target names. In Table 15.1 we list these with their interpretations.

Typically, one provides a makefile for each directory in which one codes. One can create a top level directory and have the make commands executed to change the directory and run the make there as shown below:

```
all:
(cd src; make)
(cd math; make)
(cd ui; make)
```

Each command launched by make gets launched within its own shell. Typically this is Bourne shell.

Imake: Imake program generates platform specific make files. One writes a set of general rules and if configured properly, it yields a suitable makefile for the individual platforms. This is achieved through a template file called “Imake.tmpl" which first determines the machine configuration and the OS (as sun.cf for sun and sgi.cf for sgi).

Next it looks for the local customization in site.def and project specific configuration in “Project.tmpl". If needed, it also uses the X11 release and motif-related information. The best way to create Imake files is to use an older file as a template and modify it to suit the present needs.

1.7 Mastermakefiles

We notice that we had to specify the dependencies explicitly. Of course, we now had the luxury of the file name stems which could be encoded. However, it often helps to use some compiler options which generate the dependencies. The basic idea is to generate such dependencies and use these repeatedly. For instance, the gnu compiler g++ with – MM option gets all the dependencies.

Once we use the (-MM) type of option to generate the options, clearly we are generating minimakefiles within a mastermakefile. This form of usage is also done by programs that install make using makefiles.

Unit-2

1.1 Some Other Tools in UNIX

We have emphasized throughout that the philosophy of Unix is to provide - (a) a large number of simple tools and (b) methods to combine these tools in flexible ways. This applies to file compression, archiving, and transfer tools as well. One can combine these tools in innovative ways to get efficient customized services and achieve enhanced productivity. Usually, the use of these tools is required when user or system files need to be moved enblock between different hosts.

In this unit we shall discuss some tools that help in archiving and storing information efficiently. Efficient utilization of space requires that the information is stored in a compressed form. We discuss some of the facilities available in Unix for compression. Also, compressed files utilize available network bandwidth better for file transfers on the internet. Without compression, communicating postscript, graphics and images or other multimedia files would be very time consuming.

One of the interesting tools we have included here is a profiler. A profiler helps in profiling programs for their performance. In program development it is very important to determine which segments of program are using what resources. We discuss a few program performance optimization strategies as well.

We begin the module with a description of archiving tools available in the Unix environment.

1.2 Tar and Other Utilities

In the past, archives were maintained on magnetic tapes. The command tar appropriately stands for the tape archiving command for historical reasons. These tapes used to be stacked away in large archival rooms. The tar command carries over to archiving files from a file system onto a hard disk. The command has a provision for a default virtual tape drive as well.

As a user, we distribute our files within a directory structure. Now suppose we need to archive or copy an entire set of files within a directory subtree to another machine. In this situation the tar command comes in handy. It would also be very handy when one has to copy a large set of system files to another machine. Many of the newer applications are also installed using a copy from a set of archived files. The *tar* command has the following structure:

As an example suppose we wish to archive all the .c files under directory. /M/RAND and place these under directory. /T, we may give a command as follows:

```
bhatt@SE-0 [~/UPE] >>tar cvf ./T/cfiles.tar ./M/RAND/*.c
a ./M/RAND/dum.c 1K
a ./M/RAND/main.c 1K
a ./M/RAND/old_unif.c 1K
a ./M/RAND/templet.c 4K
a ./M/RAND/unif.c 1K
```

We may now change to the directory T and give a *ls* command to see what we have there.

```
bhatt@SE-0 [T] >>ls
ReadMe cfiles.tar
```

The options used in *tar* command have the following interpretations:

The option c suggests to create, v suggests to give a verbose description on what goes on, and f indicates that we wish a file to be created in a file system. An absence of f means it will be created in /etc/remt0 = or a tape. To see the table of contents within a tarred file use the t option as shown below:

```
bhatt@SE-0 [T] >>tar tvf cfiles.tar
tar: blocksize = 18
-rw-r--r-- 10400/14210 414 Nov 30 15:53 1999 ./M/RAND/dum.c
-rw-r--r-- 10400/14210 364 Nov 30 16:38 1999 ./M/RAND/main.c
-rw-r--r-- 10400/14210 396 Nov 30 16:29 1999 ./M/RAND/old_unif.c
-rw-r--r-- 10400/14210 3583 Nov 29 11:22 1999 ./M/RAND/templet.c
-rw-r--r-- 10400/14210 389 Oct 16 09:53 2000 ./M/RAND/unif.c
```

To extract the files from the tarred set *cfiles*, we may use the x option as follows:

```
tar xvf cfiles.tar
```

This creates a directory M under T (M was the parent directory of RAND) under which RAND and files stood in the first place.

In particular, *tar* has the following options:

c: create an archive.
r: append files at the rear end of an existing archive.
t: list table of contents.

x : extract individual contents
f : file to be created within a file system
f : write/read from standard output/input
o : change file ownership
v : verbose mode, give details of archive information

Note that it is dangerous to tar the destination directory (i.e. take the archive of the destination directory) where we propose to locate the .tar file as this results in a recursive call. It begins to fill the file system disk which is clearly an error.

The *tar* command is very useful in taking back-ups. It is also useful when people move - they can tar their files for porting between hosts.

1.3 Compression

The need to compress arises from efficiency consideration. It is more efficient to store compressed information as the storage utilization is much better. Also, during network transfer of information one can utilize the bandwidth better. With enormous redundancy in coding of information, files generally use more bits than the minimum required to encode that information. Let us consider text files. The text files are ASCII files and use a 8 bit character code. If, however, one were to use a different coding scheme one may need fewer than 8 bits to encode. For instance, on using a frequency based encoding scheme like Huffman encoding, we would arrive at an average code length of 5 bits per character. In other words, if we compress the information, then we need to send fewer bits over the network. For transmission one may use a bit stream of compressed bits.

As such *tar*, by itself, preserves the ASCII code and does not compress information. Unix provides a set of compression utilities which include a compress and a *uuencode* command. The command structure for the compress or uncompress command is as follows:

```
compress options filename
uncompress options filename
```

On executing the *compress* command we will get file with a .Z extension, i.e. with a file filename we get filename.Z file. Upon executing *uncompress* command with filename.Z as argument, we shall recover the original file filename.

The example below shows a use of compress (also uncompress) command which results in a .Z file.

```
bhatt@SE-0 [T] >>cp cfiles.tar test; compress test; ls
M ReadMe cfiles.tar test.Z
bhatt@SE-0 [T] >>uncompress test.Z; ls
M ReadMe cfiles.tar test
```

Another method of compression is to use the *uuencode* command. It is quite common to use a phrase like *uuencode* a file and then subsequently use *uudecode* to get the original file. Let us *uuencode* our test file. The example is shown below:

```
bhatt@SE-0 [T] >>uuencode test test > test.uu ; ls; rm test ; \
ls ; uudecode test.uu ; rm test.uu; ls
ReadMe cfiles.tar test test.uu M
ReadMe cfiles.tar test.uu M
ReadMe cfiles.tar test
```

Note that in using the *uuencode* command we have repeated the input file name in the argument list. This is because the command uses the second argument (repeated file name) as the first line in the compressed file. This helps to regenerate the file with the original name on using *uudecode*. Stating it another way, the first argument gives the input file name but the second argument helps to establish the file name in the output.

One of the most common usages of the *uuencode* and *uudecode* is to send binary files. Internet expects users to employ ASCII format. Thus, to send a binary file it is best to *uuencode* it at the source and then uudecode it at the destination.

The way to use *uuencode/uudecode* is as follows:

```
uuencode my_tar.tar my_tar.tar > my_tar.uu
```

There is another way to deal with internet-based exchanges. It is to use MIME (base 64) format. MIME as well as SMIME (secure MIME), are Internet Engineering Task Force defined formats. MIME is meant to communicate non-ASCII characters over the net as attachments to a mail. Being a non-ASCII file, it is ideally suited for transmission of post-script, graphics, images, audio or video files over the net. A software *uudeview* allows one to decode and view both MIME and *uuencoded* files.

A *uuencoded* file must end with end without which the file is considered to end improperly. A program called *uudeview* is very useful to decode *uuencoded* files as well as files in the base 64 format.

Zip and unzip: Various Unix flavors, as also MS environments, provide instructions to compress a file with the *zip* command. A compressed file may be later unzipped by using an *unzip* command. In GNU environment the corresponding commands are *gzip* (to compress) and *gunzip* (to uncompress). Below is a simple example which shows use of these commands:

```
bhatt@SE-0 [T] >>gzip test; ls; gunzip test.gz; ls;
M         ReadMe         cfiles.tar test.gz
M        ReadMe         cfiles.tar test
```

In MS environment one may use .ZIP or PKZIP to compress and PKUNZIP to decompress the files.

One of the best known compression schemes is the LZW compression scheme (the letters LZW stand for the initials of the two inventors and the one who refined the scheme). It was primarily designed for graphics and image files. It is used in the .gif format. A discussion on this scheme is beyond the scope of this book.

Network file transfers: The most frequent mode of file transfers over the net is by using the file transfer protocol or FTP. To perform file-transfer from a host we use the following command.

```
ftp <host-name>
```

This command may be replaced by using an *open* command to establish a connection with the host for file transfer. One may first give the *ftp* command followed by *open* as shown below:

```
ftp
open <host-name>
```

The first *ftp* command allows the user to be in the file transfer mode. The *open* arranges to open a connection to establish a session with a remote host. Corresponding to open we may use *close* command to close a currently open connection or FTP session. Most FTP protocols would leave the user in the FTP mode when a session with a remote host is closed. In that case a user may choose to initiate another FTP session immediately. This is useful when a user wishes to connect to several machines during a session. One may use the *bye* command to exit the FTP mode. Usually, the *ftp ftp* connects a user to the local server.

With anonymous or guest logins, it is a good idea to input one's e-mail contact address as the password. A short prompt may be used sometimes to prompt the user. Below we show an example usage:

user anonymous e-mail-address

Binary files must be downloaded using the BINARY command. ASCII files too can be downloaded with binary mode enabled. FTP starts in ASCII by default. Most commonly used *ftp* commands are *get* and *put*. See the example usage of the *get* command.

```
get <rfile> <lfile>
```

This command gets the remote file named *rfile* and assigns it a local file name *lfile*. Within the FTP protocol, the *hash* command helps to see the progression of the *ftp* transfers. This is because of # displayed for every block transfer (uploaded or downloaded). A typical *get* command is shown below.

```
ftp> hash
ftp> binary
ftp> get someFileName
```

During multiple file downloads one may wish to unset interactivity (requiring a user to respond in y/n) by using the *prompt* command. It toggles on/off on use as shown in the example below:

```
ftp> prompt
ftp> mget filesFromAdirectory
```

The *mget* or *mput* commands offer a selection to determine which amongst the files need to be transferred. One may write shell scripts to use the *ftp* protocol command structure.

This may be so written to avoid prompts for *y/n* which normally shows up for each file transfer under the *mget* or *mput* commands.

Unlike *tar*, most *ftp* protocols do not support downloading files recursively from the subdirectories. However, this can be achieved in two steps. As a first step one may use the tar command to make an archive. Next, one can use the *ftp* command to effect the file transfer. Thus all the files under a directory can be transferred. In the example below, we additionally use compression on the tarred files.

1. Make a tar file: create xxx.tar file
2. Compress: generate xxx.tar.z file
3. Issue the ftp command: ftp

Below we have an example of such a usage:

1. **Step 1:** *$ tar -cf graphics.tar /pub/graphics*

This step takes all files in /pub/graphics and its subdirectories and creates a tar file named graphics.tar.

2. **Step 2:** $ compress graphics.tar

This step will create graphics.tar.z file.

3. **Step 3:** uncompress graphics.tar.z to get graphics.tar
4. **Step 4:** tar gf graphics.tar will give file gf .

1.4 Image File Formats for Internet Applications

With the emergence of the multimedia applications, it became imperative to offer services to handle a variety of file formats. In particular, we need to handle image files to support both still and dynamic images. Below we give some well known mutimedia file formats which may be used with images.

File extension	Usage
.AVI	Audio visual interleave
.DL	Animated picture files (with .PICT .MAC extensions).
.PCX and .BMP	May be IBM PC image files
.WPG	A word perfect file.
.RAW	May be 24 bit RGB picture file.

JPEG is a compression standard specification developed by the Joint Photographic Engineering Group. There are utilities that would permit viewing the images in the .jpg files. Most JPEG compressions are lossy compressions. Usage of .jpg files is very popular on internet because it achieves a very high compression. In spite of being lossy, jpeg compression invariably works quite well because it takes into account some weaknesses in human vision. In fact, it works quite adequately with 24-bit image representations.

JPEG files are compressed using a quality factor from 1 to 100 with default being 55. When sending, or receiving, .jpg files, one should seek for quality factor of 55 and above to retain image quality at an acceptable level.

The basic *uuencoding* scheme breaks groups of three 8-bit characters into four 6-bit patterns and then adds ASCII code 32 (a space) to each 6-bit character which in turn maps it on to the character which is finally transmitted. Sometimes spaces are transmitted as grave-accent (' ASCII 96). The *xxencoding* is newer and limits itself to 0-9, A-Z, a-z and +- only. In case a compressed file is *uuencoded, uudecode* may have to be followed by an *unzip* step.

File sizes up to 100{300K are not uncommon for .gif files. Often UseNet files are delimited to 64k. A typical 640*480 VGA image may require transmission in multiple parts. Typical .gif file in Unix environment begins as follows:

begin 640 image.gif

640 represents the access rights of file.

Steps for getting a .gif or .jpg files may be as follows:

1. **Step 1:** Get all the parts of the image file as part files.
2. **Step 2**: Strip mail headers for each part-file.
3. **Step 3:** Concatenate all the parts to make one .uue file.
4. **Step 4:** uudecode to get a .gif/.jpg or .zip file.
5. **Step 5:** If it is a .zip file unzip it.
6. **Step 6:** If it is a .jpg file either use jpg image viewer like cview or, alternatively, use jpg2gif utility to get .gif file.
7. **Step 7:** View image from .gif file.

On the internet several conversion utilities are available and can be downloaded.

1.5 Performance Analysis and Profiling

One of the major needs in program development environments is to analyse and improve the performance of programs. For our examples here we assume c programs.

One may use some obviously efficient steps to improve efficiency of code. In his book, *The Art Of Computer System Performance Analysis*, Raj Jain advocates the following:

- Optimize the common case. In other words, if the program has to choose amongst several operational paths, then that path which is taken most often must be made very efficient. Statements in this path must be chosen carefully and must be screened to be optimal.
- In case we need to test some conditions and choose amongst many alternative paths by using a sequence of if statements, then we should arrange these if statements such that the condition most likely to succeed is tested first, i.e. optimize the test sequence to hit the most likely path quickly.
- Within an if statement if there are a set of conditions that are ANDED, then choose the first condition which is most likely to fail. This provides the quickest exit strategy.
- If the data to be checked is in the form of arrays or tables, then put these in the order such that the most likely ones to be accessed are up front, i.e. these get checked first.
- Avoid file input and output as much as possible. Also, batch as much input, or output, as possible. For instance, suppose we need to process each line of data. In such a case, get the file first in memory to process it line-by-line rather than reading the line-by-line data from disk for each step of processing.

- In the loops make sure to pull out as much data as possible. In particular, values that do not change within the loop computations need not be within the loop.

Steps To Analyze Performance Of c Programs: The following steps will walk us through the basic steps:

1. Compile with p option, the profiler option
   ```
   cc -p a.c
   ```
   (Additionally, use -o option for linked routines.)

2. Now run the program a.out. (This step results in a mon.out file in the directory)

3. Next see the profile by using prof command as follows: prof a.out

(For more details the reader is advised to see the options in man pages for prof command.)

The profiler gives an estimate of the time spent in each of the functions. Clearly, the functions that take a large percentage of time and are often used are the candidates for optimization.

A sample program profile: First let us study the following program in c:

```
#include <stdio.h>
#include <ctype.h>
int a1;
int a2;
add() /* adds two integers */
{

int x;
int i;
for (i=1; i <=100000; i++)
x = a1 + a2;
return x;
}
main()
{
int k;
a1 = 5;
a2 = 2;
for (k=1; k <=1000000; k++);;
printf("The addition gives %d \n", add());
}
```

Now let us see the way it has been profiled.

```
bhatt@SE-0 [P] >>cc -p a.c
bhatt@SE-0 [P] >>a.out
The addition gives 7
bhatt@SE-0 [P] >>ls
```

```
ReadMe a.c a.out mon.out
bhatt@SE-0 [P] >>prof a.out

%Time       Seconds     Cumsecs     #Calls      msec/call     Name
  77.8        0.07        0.07        1            70.        main
  22.2        0.02        0.09        1            20.        Add

bhatt@SE-0 [P] >>
```

The table above lists the percentage time spent in a certain function, time in seconds, and the number of calls made, average milliseconds on calls and the name of the function activated.

Another profiler is a *gprof* program. It additionally tries to find the number of iterated cycles in the program flow graph for function calls.

Text Processing (Improving Performance): Most often computer programs at tempt text processing. One of the common strategies is the use a loop in the following way:

1. Find the *strlen* (the length of the string).
2. Use a loop and do character-by-character scan to process data.

What most users do not realize is that the *strlen* itself determines the string length by traversing the string and comparing each of the characters with null.

Clearly, we can check in the loop if the character in the string array is null and process it if it is not. This can save a lot of processing time in a text processing program.

Sometimes we are required to copy strings and use *strcpy* to do the task. If the architecture supports memory block copy, i.e. an instruction like *memcpy* then this is a preferred option as it is a block transfer instruction whereas *strcpy* copies byte-by-byte and is therefore very slow.

1.6 Source Code Control System in UNIX

A very large programming project evolves over time. It is also possible that many teams work concurrently on different parts of such a large system. In such a case each team is responsible for a certain part of the project. Their own segments evolve over time. These segments must dovetail with the rest of the project team efforts. So, during the development phase, each development team has to learn to manage changes as they happen. It is quite a common practice to proceed on the assumption of availability of some module at a future time. So, even while designing one's own mandated module, one needs to be able to account for some yet to be completed module(s). One may even have to integrate a newer version of a module with enhanced features. In some sense, a software may have several generations and versions of evolution. In industry parlance, these generations are sometimes called α or β releases. There may even be a version number to suit a certain specific configuration.

Unix supports these developmental possibilities, i.e. supporting creation and invocation of various versions by using a system support tool called SCCS. In this chapter we shall study how SCCS helps in versioning.

1.7 How Does Versioning Help

Essentially, a large software design is like an emerging collage in an art studio with many assistants assisting their master in developing a large mural. However, in the software design, the ability to *undo* has its charms. There are two major ways in which versioning helps. First, we should realize that a rollback may be required at any stage of development. Should it be the case then, it should then be possible for us to get back to an acceptable (and perhaps an operating) version of a file. From this version of file one may fork out to a newer file which is a better version. The forking may be needed to remove some errors which may have manifested or because it was felt necessary to add newer features.

Secondly, software should cater to users of various levels of abilities and workplace environments. For instance, a home version and office version of XP cater to different workplace environments even while offering truly compatible software suites. In other words, sometimes customers need to be provided with options to choose from and tune the software for an optimal usage which may entail making choices for features, adding some or leaving a few others to leverage the advantage of a configuration with as little overhead as possible.

1.8 The SCCS

During the period of development, there will be modules in various stages of progression. As more modules become *stable*, the system comes along in small increments. Each stable set yields a fairly stable working version of the system. So the support one seeks from an operating system is to be able to *lock* each such version for limited access. In addition, one should be able to *chain* various stages of locked versions in a *hierarchy* to reflect particular software's evolution. Unix obtains this capability by using a version tree support mechanism. The version tree hierarchy is automatically generated. The version number identifies the evolution of different nodes in the version tree over time. A typical version tree numbering scheme is shown in Figure 17.1. The higher the number, the more recent is the version. The numbers appear as an extension to show the evolution. In the next few sections we see what is provided and how it is achieved under Unix.

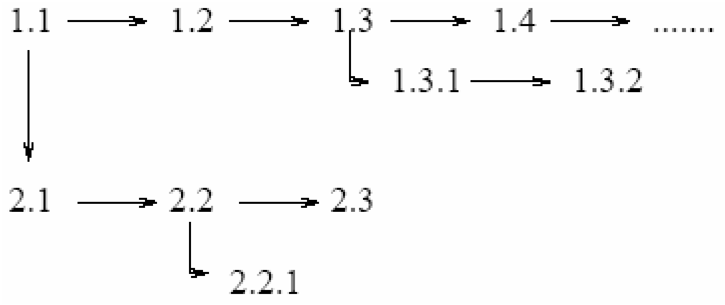


Figure 17.1: A typical version tree under SCCS.

1.8.1 How This Is Achieved

Unix obtains this capability by maintaining a version tree with the following support mechanisms.

- Version tree hierarchy is generated automatically and follows a numbering scheme as shown in figure 17.1. The numbering helps users to identify the evolution of software from the version tree. Internally, it helps Unix in maintaining differences between related versions of files.
- The SCCS creates files in a special format.
- The compilation is handled in a special way.
- Edits to the files are handled in a special way. In fact one can edit files at any level in the version tree.
- A suite of SCCS commands are provided to manage versions and create links in the version trees.

In addition, Unix provides the following facilities:

- Unix permits locking out of modules under development. Some modules which are stable may be permitted full access. However, those versions of modules that are still evolving may not be accessible to all the users. The access to these modules may be restricted to this module's developers only. General users may have access to an older stable version of the module. It is possible to also add gid for access to a new version being released. In Unix the access is permitted for all (or none) in the group. So adding gid for a release adds all the users in that group.

- For management of hierarchy, it is possible to create a tree structure amongst versions. At any stage of development it is possible to create siblings in a tree. Such a sibling can have its own child versions evolving through the sub-trees under it. As shown in Figure 17.1, the version tree grows organically as new versions emerge.
- Unix supports precedence amongst the versions of a module. So if a module has a newer version (usually with some bugs removed or with extensions), then this module shall be the one that shall have precedence during the loading of modules. Often this is true of files where a file may be updated frequently.

Various Unix versions from BSD, System V or Linux provide SCCS-like tools under a variety of utility names. These are CSSC, CVS, RCS, linCVS, etc. We will briefly discuss CVS in Linux environment in Section 1.10. Obviously, more recent versions clearly have more bells and whistles. Now that we have explained the underlying concepts, we shall next explore the command structure in SCCS that makes it all happen.

1.9 SCCS Command Structure

We can appreciate the commands under SCCS better if we first try to understand the design considerations. Below we enumerate some of the points which were borne in mind by the designers when SCCS was created initially.

- Between two revisions a file under SCCS is considered to be idle.
- Between two adjacent versions, both idles, there is only a small change in the content. This change is regarded as *delta*. Suppose for some file *F* there is a delta change *d*, then one needs to maintain both *F* and d to be able to do the version management. This is because SCCS supports versioning at every node in a version tree.
- Version numbers can be generated automatically. The delta changes finally result in creating a hierarchy. In fact, version numbers determine the traversal required on the version tree to spot a particular version.
- We need a command *use* to indicate which file we retrieve and a command *delta* to indicate the change that needs to be incorporated. That then explains the command structure for SCCS.

Figure 17.2 illustrates the manner in which an SCCS encoded file may be generated.

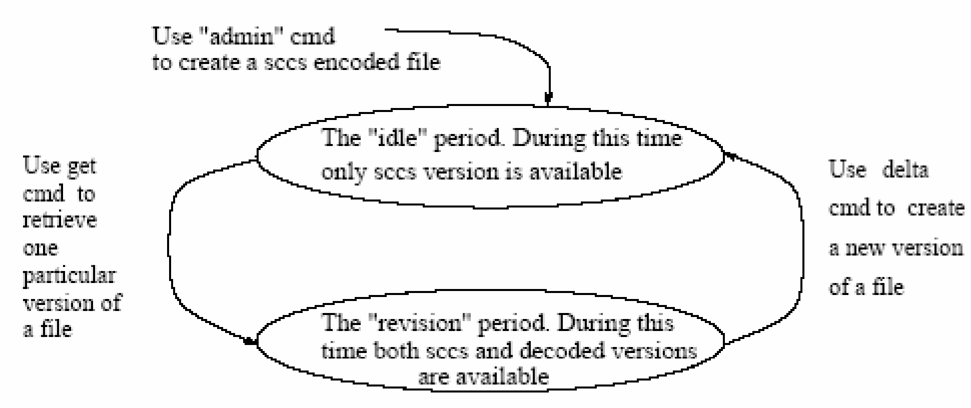

Use "admin" cmd
to create a sccs encoded file
The "idle" period. During this time
only sccs version is available
Use get
cmd to
retrieve
one
particular
version of
a file
Use delta
cmd to create
a new version
of a file
The "revision" period. During this
time both sccs and decoded versions
are available

Figure 17.2: The “idle" and “revision" periods in SCCS.

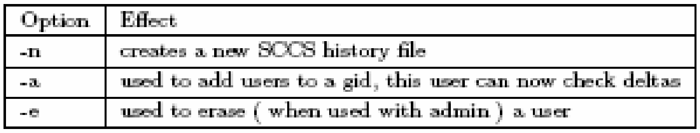
Such files may subsequently go through an evolution with several idle periods in between. When a file is retrieved, it is decoded for normal viewing and presented. Usually, one uses the delta command to create a new version. Based on the version taken up for revision, a new version number is generated automatically.

1.9.1 An Example Session

In this example session we have used some command options. In Table 17.1 we give explanation for options available with various SCCS commands.

To begin with we need to create a new file. We may do so as follows:

admin -n s.testfile



Option	Effect
-n	creates a new SCCS history file
-a	used to add users to a gid, this user can now check deltas
-e	used to erase (when used with admin) a user

Table 17.1: SCCS command options.

This command creates an SCCS formatted file with an empty body. The options used with admin command have interpretations given in table 17.1.

We can now use the visual editor and put some text into it.

get -e s.testfile

The -e option is to invoke ”open for edit". This command prints the version number and number of lines in testfile. If we perform a delta, we can create a newer version of testfile. This can be done simply as follows:

delta s.testfile

We may just add comments or actually make some changes. This can be now checked by using a visual editor again.

Unix provides a facility to view any file with a given version number. For example, to view and run a certain previous version 1.2, we may use the command shown below.

```
get -p -r1.2 s.testfile
```

The -p option is to invoke a path and -r is to invoke a run.

Like Make, SCCS also supports some macros. Readers should consult their system's documentation to study the version management in all its facets. We shall next take a brief tour of CVS.

1.10 CVS : Concurrent Versioning System

Linux environment offers a concurrent versioning utility, called CVS which supports oncurrent development of a project. The team members in the project team may make concurrent updates as the project evolves. The primary idea in CVS revolves around maintaining a project repository which always reflects the current official status of the project. CVS allows project developers to work on parts of the project by making copies of the project in their own scratch pad areas. Updates on the work-in-progress scratch pads do not affect the repository. Individuals may seek to finalize their work on the scratch pad and write back to the repository to update a certain aspect of the project.

Technically, it is quite possible for more than one individual to make copies and develop the same program. However, this can raise consistency problems. So writing back into the repository is done in a controlled way. If the updates are in different parts, these are carried out with no conflict. If the updates are in the same part of a file, a merge conflict may occur and these are reported. These conflicts need to be reconciled before a commit into the repository is performed.

Essentially, CVS is a command line utility in Linux (and also in some other flavors of Unix). There are network versions of CVS that allow access to repository over the network as well. For now we shall assume that we have a CVS utility available on a single machine with multiple users accessing the files in it. Let us examine some typical usage scenarios of CVS commands. For example, consider the command lines below.

```
export CVSROOT=/homes/iws/damu/cvs
cvs checkout project-5
```

The first command line is to look for the cvs command in user damu's home directory. The second line basically checks out what files are there in project-5. It also makes the copies of project-5 files available to the user. Typically, these may be some .c or .h files or some others like .tex files. As we stated earlier, CVS supports an update phase of operation. The update command and a typical response from it are shown next.

```
cvs update project-5
```

When one attempts to update, typically the system prompts to indicate if someone else also made any updates after you had copied the files. In other words, all the updates that occurred in sequence from different people are all shown in order. Each update obtains a distinct version number. Like SCCS, CVS also generates internally a numbering scheme which will give a versioning tree. In the case someone else’s' update is at the same location in a file, then the messages would indicate if there is a merge conflict. A response to a typical update command is shown below:

```
$ cvs update
cvs update: Updating .
RCS file: /homes/iws/damu/cvs/project-5/main.c,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into proj.h
M proj.h
U main.c
```

In the example above we had no merge conflicts. In case of a merge conflict, we get messages to indicate that. Let us assume we have merge conflict in updating main.c, then the messages we may get would look like those shown below.

```
$ cvs update

cvs update: Updating .
RCS file: /homes/iws/damu/cvs/project-5/main.c,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into main.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in main.c
C main.c
```

Usually, the conflicts are shown with some repeated character sequence to help identify where it occurred. One, therefore, needs to resolve the conflicts and then may be commit the correctly updated version to repository. A CVS *commit* generates a newer version in repository. A CVS commit command and its response is shown below.

```
$ cvs commit
cvs commit: Examining .
Checking in main.c;
/homes/iws/damu/cvs/project-5/main.c,v <-- main.c
new revision: 1.2; previous revision: 1.1
done
```

The steps shown above for checkout, update and commit are the basic steps. There are means available to *import* an entire project and effect an update to create a newer version of the project. For those details the reader is advised to refer to the man pages.

Unit-3

1.1 X Windows in UNIX

1.2 Graphical User Interface (GUI)

1.2.1 X-Window System

1.3 Some Standard X-clients

1.4 Hosts

1.5 Selecting a host for Display

1.6 X-Utilities

1.7 Startup

1.8 Motif and X

1.1 X Windows in UNIX

Presently our communication with a computer is largely using a keyboard and a visual feedback from a display unit (VDU). To support human computer interaction (HCI), we need software support. For instance, we need a display in the form of a prompt on the screen. This reassures us on what is being input. This interface is further enhanced with iconic clues which help us launch the applications in place of character input.

In addition, many applications need a support to be able to display the results of a computation in a visual form. These require a software support in the form of drawing support on VDU. This entails support to be able to draw lines, join a set of lines in the form of continuous curve. It is even possible to generate a control panel to regulate the course of a set of operations in an application.

Unix provides the X-graphics library support to both OS and applications. In this chapter, we shall discuss the X-Windows support. For most Linux installations, we have a built-in X11 version library package. We begin with a brief description of what a Graphical User Interface (GUI) is. Later we briefly describe how the X-Windows system evolved.

1.2 Graphical User Interface (GUI)

GUI, as the name suggests, is an artifact that facilitates availing operating system facilities with the aid of widgets. The widgets basically are windows and icons which provide us with a clue to whatever we may be able to do.

One of the first well known GUIs was the Xerox STAR system which allowed users to have icons for a document page folded at a corner. The STAR had a printer icon and the operations were simply “drag and drop". Subsequently, these were adapted for the Apple Macintosh. Now-a-days all PCs and Workstations support a GUI. SunOS, the predecessor of Solaris had developed a GUI for Unix users which allowed users to have multiple terminals. Each open window is a terminal connected to the machine giving an illusion as if you, as a user, are multiply connected with the machine. Essentially, this gives users a capability similar to being connected to the machine from multiple terminals. Basically, it facilitates running many tasks at a time for a user. For instance, one may edit a program file in one window and run it from another window.

1.2.1 X-Window System

The X-Window system was developed in MIT under project Athena in 1984. The XWindow system operates on the basis of a client and server architecture. Further, X facilitates operation over a set of networked computers by defining a protocol. The protocol mechanism replaced function calls to let application draw its image in a window.

The application is now the client and drawing the image is assigned to an X-server. The server handles the bit mapped graphics for the client application. In addition, the server also handles all display-related communications from the client. It is quite possible that for a single X-server there are many clients. In other words while handling communications from multiple clients, an X-server can in fact support drawing images for all connected clients. Clearly, this makes it possible to let many applications draw images on the same screen. The clients may be anywhere on a communication network.

As almost all manufacturers support X-compliant protocols, we now have a capability to run an application on a machine from one manufacturer and display the image on another machine with a terminal from another manufacturer.

X-Servers and clients: X-servers support bitmapped displays and this happens as soon as a user logs onto a machine. Some systems may require that the user has to initiate the X-server operation by giving a *xinit* or a *xstart* command. On my set-up I use a *.xinitrc* run command file to customize my display terminal.

Usually X-clients are programs that communicate with one or more X-servers.

Window managers: The metaphor of desktop is important for a user. By analogy, a user may choose to activate one of the tasks from many on his desk. The window managers are designed to provide this illusion. In other words, a window manager (WM in short) must provide facilities that can support such desk-oriented operation from the view on the screen.

All operations within a window are supported by a window manager. The window screen is usually in the ratio of 3× 4, i.e. it may be 600× 800 pixels in size. It is possible to regulate this size. The write access into the window can be protected in a common desktop environment. If a user has been away and the system switches to the power-save mode then the system may restrict access only to a valid password holder. Alternatively, there may be a provision to lock a screen using the *xlock* command.

A window manager is a program, an X-client to be precise, which supports communication with keyboard, mouse and provides the basic interface for the user. A user can instruct the system to resize, iconify, delete a window or even magnify a part of the application image on the screen.

Though mostly the window manager runs on the same machine where the display is, it may actually reside on another machine!! The window manager maintains a “focus" which identifies the currently active window. This also helps to identify the application to which the data (or event) is to be communicated from keyboard (or mouse). Window focus may be obtained by clicking or simply by a “mouse-over" event. Window managers

also help to maintain top menu bars, support widgets like the pull-down menus to select an operation or to open a new window, *iconify*, resize, or delete (close) an existing window. So technically an X-server does no management, it only serves to display what the WM client asks it to display.

Some well known WMs: Some of the well known window managers are listed below:

1. *mwm* (Motif window manager)
2. *twm* (Tom's window manager). In *tvwm* v is for virtual.
3. *olwm* (Open look window manager). Sun's Open Look WM.

Hierarchy amongst windows: Practically every window manager supports widgets like menus (pull down and cascaded), dialog boxes with a variety of buttons like check button, radio button, selection from a list, or text input windows with scroll bars and alert windows to receive inputs like o.k. or cancel buttons. Usually, there is a root window and all the widgets are themselves drawn as small windows hierarchically within the parent window.

1.3 Some Standard X-clients

The Unix environment provides a slew of facilities in the form of clients that seek specific services. Let us briefly enumerate these clients and describe their functionalities:

- *xclock*: A clock display. It is possible to have a numerical or an analog display for the current time.
- *xbiff*: A Berkeley support program that displays mail status with a flag up, to show the arrival of a new mail.

- *xterm*: It provides a user with a new terminal window. With -C option the window can receive and display console messages.

Most of these clients have options like -geometry to indicate the size of display as shown in the example below.

```
xclock -geometry 80x80-10+10
```

This command seeks to display an 80£80 pixel clock, 10 pixels away from the right-hand corner of the screen. We can choose foreground and background presentation styles. One may also choose a title for a new window as shown below and even iconise it.

```
xterm -foreground green -background yellow -title "My Window"
xterm -iconic -title "My IconisedWindow"
```

One of the standard facilities all WMs provide is to cut and paste. It is usually possible to cut from any of the windows and paste it anywhere on the receiving window. This is very useful to transfer texts and images as bit maps.

1.4 Hosts

The X protocol requires to maintain a host-table in the /etc/hosts file. The machines listed in this table alone can set up a display on a host. For instance, my machine SE-0 has the following host table entries:

```
# Internet host table
#
127.0.0.1 localhost
172.21.36.10 SE-0 loghost
210.163.147.1 a463-2.info.kochi-tech.ac.jp a463-2 canna-server loghost
210.163.147.2 main.info.kochi-tech.ac.jp a464-1 printer-server-host
172.21.36.20 SE-P0
172.21.36.2 SE-1
172.21.36.11 SE-2

.
.
172.21.36.17 SE-10
#
```

The host-table entries may be modified to add or delete hosts by using the *xhost* command as shown below.

To add an additional host:

```
xhost +addhostname
```

To delete an additional host:

xhost –deletehostname

(Caution: Not having a host name may delete all entries in the host-table)

1.5 Selecting a host for Display

With our ability to connect to a remote machine, we may run an application on a remote machine. Let us consider that we are executing the application on a host called *rem_host* while we are operating from a host loc host. However, if we wish to visualize the output on *loc_host* we shall have to choose the display to be available on loc host while the application may be generating the output for display on *rem_host.* This usually requires the use of X-protocol of display. We should have *loc_host* as well as *rem_host* identified in the host lists on both the machines. In this scenario it is required to set display on loc host. This is done as explained below. To begin with, let us interpret the following command:

xterm -display hostname:0.0

which will open a window on the hostname. The string :0.0 identifies the monitor and the *xserver* running on the hostname. Sometimes when we remote login using a telnet command, we may have to set the display from a remote computer to be on the host where we are presently logged in. This can be done by executing the following command in the window connecting to the remote machine.

setenv DISPLAY hostname:0.0

With this command executed on the remote machine we will get the application's image displayed on hostname. Note that the default, i.e. the local host is unix:0.0.

1.6 X-Utilities

Just as Unix system provides a command repertoire, X also has a suite of commands that provide some utilities. It is instructive to see the output for the following commands:

1. *showrgb*: Shows the colours that can be generated.
2. *xcalc*: Gives a hand hand calculator.
3. *xclipboard*: gives a clipboard to cut and paste.
4. *xclock*: Gives a clock running the time of the day.
5. *xfontsel*: Lists the fonts and font names.

6. *xhost*: Lists the hosts.
7. *xload*: Displays load on the system.
8. *xlsfonts*: Lists the fonts on the system.
9. *xmag*: Displays a magnified image of a part of the screen.
10. *xman*: Opens on line manual pages.
11. *xpr*: prints the image created by xwd.
12. *xrdb*: Loads resource setting in active database.
13. *xset*: Sets display, key board or mouse preferences.
14. *xterm*: emulate a character terminal.
15. *xwd*: stores a screen image.
16. *xwininfo*: Displays the window information.

The xrdb command has several options. Use it as follows:

xrdb -query ....this gives the resources defined for X-server.

xrdb -load ....use it to load new resource information (deleting old resource information).

xrdb -merge .... merges new resource information with the existing.

xrdb -remove ..... removes the existing resource information.

(Note that .Xdefaults has the list of current resources.)

1.7 Startup

In this section we shall discuss the support offered for startup by .Xsessions, .Xdefaults files and .xinitrc files.

.Xsessions and .Xdefaults files: These files are usually located under X11 directory. On systems that employ xdm (the display manager) one can set up a sessions file as .Xsessions file to launch favorite applications on logging in. This may be with commands like:

xterm -sb & (For launching a terminal with a scroll bar)

xclock & (For launching a clock)

xterm -iconic (For launching an iconised terminal)

The .xdefaults file is utilized to select options for customization of applications display. For instance, one can choose the background colour for clock display, font size for character display and its colour by choosing an appropriate terminal foreground. A typical file may be as shown below:

```
! The exclamation sign identifies that this is a comment.
!
! Turn on the scroll bar
XTerm*scrollBar: True
! Select a font size
XTerm*Font: 10x20
! Lines to save on scroll
XTerm*saveLines: 100
! Mouse control to determine active window
Mvm*keyboardFocusPolicy: Pointer
```

One can load this file using the command

```
xrdb -load $HOME/.Xdefaults
```

.xinitrc files: Usually *xinit* starts the X-server on the host. The *xinit* command may be initiated from *.cshrc* (or .xinitrc or some other log in startup file) as one logs in. The .xinitrc may even launch a user's favourite applications like a browser, mail and a clock, etc. Some parts of my .xinitrc file are displayed below:

```
if [ -f ${HOME}/.Xdefaults ]; then
xrdb -load ${HOME}/.Xdefaults
else
xrdb -load /usr/local/X11/lib/X11/Xdefaults
fi
echo `hostname` > ${HOME}/.display.machine
echo `date +"seit: "%H.%M` >> ${HOME}/.display.machine
chmod a+r $HOME/.display.machine
echo `hostname` > ${HOME}/.display.host
xclock -g 85x76+1064+743 -bw 0 &
#xsetbg background3.gif

#xsetbg test2.gif
#xsetbg fujisun.gif
xsetbg marefoal.gif
/home/marco/bin/netscape/netscape -iconic&
xterm -g 80x10+0+0 -bw 5 -C -sb -bg MediumTurquoise -fg black +vb -fn 9x15 -
iconic -fn fi xterm -g 80x10+50+50 -bw 5 -C -sb -bg MediumTurquoise -fg black +vb
-fn 9x15 -iconic -fn xterm -g 80x20+210+210 -bw 5 -sb -bg SeaGreen -fg white -vb –
fn 9x15 -iconic -fn fixed -T exec fvwm
~/clear_colormap -f /dev/cgsix0
logout
```

1.8 Motif and X

Motif seems to be currently the most favoured X-Windows interface on Unix-based systems. This helps in some respect, as all vendors tend to use the Motif library. Technically *mwm*, the Motif WM, is yet another application running on the X-server. The .mwmrc file can be used in these systems. Applications developed using the Motif library give a common look and feel just as the Windows does it for a PC.

Note that sometimes a user may have to copy the system's mwmrc file to user home directory to get the .mwmrc file. It is a good idea to take a peek at this file to see how the window (and its border), the mouse or arrow button roles bindings are defined. Below we show settings that one may find explanatory:

```
Buttons DefaultButtonBindings
{
<Btn1Down> icon|frame f.raise
<Btn2Down> icon|frame f.post_menu
<Btn2Down> root f.menu DefaultRootMenu
}
```

One may be able to customise a menu using f.exec as shown below. f.exec takes an argument which may run a X-utility.

```
Menu PrivateMenu
{
"Tools" f.title

"ManPages" f.exec "xman &"
"Cal" f.exec "xcalc &"
"Mail" f.exec "xterm -e Mail &"
}
```

Unit-4

1.1 System Administration in UNIX

In the context of the OS service provisioning, system administration plays a pivotal role. This is particularly the case when a system is accessed by multiple users. The primary task of a system administrator is to ensure that the following happens:

a. The top management is assured of efficiency in utilization of the system's resources.
b. The general user community gets the services which they are seeking.

In other words, system administrators ensure that there is very little to complain about the system's performance or service availability.

In Linux environment with single user PC usage, the user also doubles up as a system administrator. Much of what we discuss in Unix context applies to Linux as well.

In all Unix flavours there is a notion of a superuser privilege. Most major administrative tasks require that the system administrator operates in the superuser mode with root privileges. These tasks include starting up and shutting down a system, opening an account for a new user and giving him a proper working set-up. Administration tasks also involve installation of new software, distributing user disk space, taking regular back-ups, keeping system logs, ensuring secure operations and providing network services and web access.

We shall begin this module by enlisting the tasks in system administration and offering exposition on most of these tasks as the chapter develops.

1.2 Unix Administration Tasks

Most users are primarily interested in just running a set of basic applications for their professional needs. Often they cannot afford to keep track of new software releases and patches that get announced. Also, rarely they can install these themselves. In addition, these are non-trivial tasks and can only be done with super user privileges.

Users share resources like disk space, etc. So there has to be some allocation policy of the disk space. A system administrator needs to implement such a policy. System administration also helps in setting up user's working environments.

On the other hand, the management is usually keen to ensure that the resources are used properly and efficiently. They seek to monitor the usage and keep an account of system usage. In fact, the system usage pattern is often analysed to help determine the efficacy of usage. Clearly, managements' main concerns include performance and utilisation of resources to ensure that operations of the organisation do not suffer.

At this juncture it may be worth our while to list major tasks which are performed by system administrators. We should note that most of the tasks require that the system administrator operates in superuser mode with root privileges.

1.2.1 Administration Tasks List

This is not an exhaustive list, yet it represents most of the tasks which system administrators perform:

1. System startup and shutdown: In the Section 1.3, we shall see the basic steps required to start and to stop operations in a Unix operational environment.
2. Opening and closing user accounts: In Unix an administrator is both a user and a super-user. Usually, an administrator has to switch to the super-user mode with root privileges to open or close user accounts. In Section 1.4 , we shall discuss some of the nuances involved in this activity.
3. Helping users to set up their working environment: Unix allows any user to customize his working environment. This is usually achieved by using .rc files. Many users need help with an initial set-up of their .rc files. Later, a user may modify his .rc files to suit his requirements. In Section 1.5, we shall see most of the useful .rc files and the interpretations for various settings in these files.
4. Maintaining user services: Users require services for printing, mail Web access and chat. We shall deal with mail and chat in Section 1.5 where we discuss .rc files and with print services in Section 1.6 where we discuss device management and services. These services include spooling of print jobs, provisioning of print quota, etc.
5. Allocating disk space and re-allocating quotas when the needs grow: Usually there would be a default allocation. However, in some cases it may be imperative to enhance the allocation. We shall deal with the device oriented services and management issues in Section 1.6.
6. Installing and maintaining software: This may require installing software patches from time to time. Most OSs are released with some bugs still present. Often with usage these bugs are identified and patches released. Also, one may have some software installed

which satisfies a few of the specialized needs of the user community. As a convention this is installed in the directory /usr/local/bin. The local is an indicator of the local (and therefore a non-standard) nature of software.

We shall not discuss the software installation as much of it is learned from experienced system administrators by assisting them in the task.

7. Installing new devices and upgrading the configuration: As a demand on a system grows, additional devices may need to be installed. The system administrator will have to edit configuration files to identify these devices. Some related issues shall be covered in section 1.6 later in this chapter.
8. Provisioning the mail and internet services: Users connected to any host shall seek Mail and internet Web access. In addition, almost every machine shall be a resource within a local area network. So for resource too the machine shall have an IP address. In most cases it would be accessible from other machine as well. We shall show the use .mailrc files in this context later in Section 1.5.
9. Ensuring security of the system: The internet makes the task of system administration both interesting and challenging. The administrators need to keep a check on spoofing and misuse. We have discussed security in some detail in the module on OS and Security.
10. Maintaining system logs and profiling the users: A system administrator is required to often determine the usage of resources. This is achieved by analyzing system logs. The system logs also help to profile the users. In fact, user profiling helps in identifying security breaches as was explained in the module entitled OS and Security.
11. System accounting: This is usually of interest to the management. Also, it helps system administrators to tune up an operating system to meet the user requirements. This also involves maintaining and analyzing logs of the system operation.
12. Reconfiguring the kernel whenever required: Sometimes when new patches are installed or a new release of the OS is received, then it is imperative to compile the kernel. Linux users often need to do this as new releases and extensions become available.

Let us begin our discussions with the initiation of the operations and shutdown procedures.

1.3 Starting and Shutting Down

First we shall examine what exactly happens when the system is powered on. Later, we shall examine the shutdown procedure for Unix systems. Unix systems, on being powered on, usually require that a choice be made to operate either in single or in multiple-user mode. Most systems operate in multi-user mode. However, system administrators use single-user mode when they have some serious reconfiguration or installation task to perform. Family of Unix systems emanating from System V usually operate with a *run level*. The single-user mode is identified with run level *s*, otherwise there are levels from 0 to 6. The run level 3 is the most common for multi-user mode of operation.

On being powered on, Unix usually initiates the following sequence of tasks:

1. The Unix performs a sequence of self-tests to determine if there are any hardware problems.
2. The Unix kernel gets loaded from a root device.
3. The kernel runs and initializes itself.
4. The kernel starts the init process. All subsequent processes are spawned from init process.
5. The init checks out the file system using fsck.
6. The init process executes a system boot script.
7. The init process spawns a process to check all the terminals from which the system may be accessed. This is done by checking the terminals defined under /etc/ttytab or a corresponding file. For each terminal a getty process is launched. This reconciles communication characteristics like baud rate and type for each terminal.
8. The *getty* process initiates a login process to enable a prospective login from a terminal.

During the startup we notice that *fsck* checks out the integrity of the file system. In case the *fsck* throws up messages of some problems, the system administrator has to work around to ensure that there is a working configuration made available to the users. It will suffice here to mention that one may monitor disk usage and reconcile the disk integrity.

The starting up of systems is a routine activity. The most important thing to note is that on booting, or following a startup, all the temporary files under *tmp* directory are cleaned up. Also, zombies are cleaned up. System administrators resort to booting when there are a number of zombies and often a considerable disk space is blocked in the *tmp* directory.

We next examine the shutdown. Most Unix systems require invoking the shutdown utility. The shutdown utility offers options to either halt immediately, or shutdown after a pre-assigned period. Usually system administrators choose to shutdown with a preassigned period. Such a shutdown results in sending a message to all the terminals that the system shall be going down after a certain interval of time, say 5 minutes. This cautions all the users and gives them enough time to close their files and terminate their active processes. Yet another shutdown option is to reboot with obvious implications.

The most commonly used shutdown command is as follows:

```
shutdown -h time [message]
```

Here the time is the period and message is optional, but often it is intended to advise users to take precautions to terminate their activity gracefully. This mode also prepares to turn power off after a proper shutdown. There are other options like k, r, n etc. The readers are encouraged to find details about these in Unix man pages. For now, we shall move on to discuss the user accounts management and run command files.

1.4 Managing User Accounts

When a new person joins an organisation he is usually given an account by the system administrator. This is the login account of the user. Now a days almost all Unix systems support an admin tool which seeks the following information from the system administrator to open a new account:

1. Username: This serves as the login name for the user.
2. Password: Usually a system administrator gives a simple password. The users are advised to later select a password which they feel comfortable using. User's password appears in the shadow files in encrypted forms. Usually, the /etc/passwd file contains the information required by the login program to authenticate the login name and to initiate appropriate shell as shown in the description below:

   ```
   bhatt:x:1007:1::/export/home/bhatt:/usr/local/bin/bash
   damu:x:1001:10::/export/home/damu:/usr/local/bin/bash
   ```

Each line above contains information about one user. The first field is the name of the user; the next a dummy indicator of password, which is in another file, a shadow file. Password programs use a trap-door algorithm for encryption.

3. Home directory: Every new user has a home directory defined for him. This is the default login directory. Usually it is defined in the run command files.
4. Working set-up: The system administrators prepare .login and .profile files to help users to obtain an initial set-up for login. The administrator may prepare .cshrc, .xinitrc .mailrc .ircrc files. In Section 1.5 we shall later see how these files may be helpful in customizing a user's working environment. A natural point of curiosity would be: what happens when users log out? Unix systems receive signals when users log out. Recall, in Section 1.3 we mentioned that a user logs in under a login process initiated by *getty* process. Process *getty* identifies the terminal being used. So when a user logs out, the *getty* process which was running to communicate with that terminal is first killed. A new *getty* process is now launched to enable yet another prospective login from that terminal.

   The working set-up is completely determined by the startup files. These are basically .rc (run command) files. These files help to customize the user's working environment. For instance, a user's .cshrc file shall have a path variable which defines the access to various Unix built-in shell commands, utilities, libraries etc.

   In fact, many other shell environmental variables like HOME, SHELL, MAIL, TZ (the time zone) are set up automatically. In addition, the .rc files define the access to network services or some need-based access to certain licensed software or databases as well. To that extent the *.rc* files help to customize the user's working environment.

   We shall discuss the role of run command files later in Section 1.5.
5. Group-id: The user login name is the user-id. Under Unix the access privileges are determined by the group a user belongs to. So a user is assigned a group-id. It is possible to obtain the id information by using an id command as shown below:

```
[bhatt@iiitbsun OS]$id
uid=1007(bhatt) gid=1(other)
[bhatt@iiitbsun OS]$
```

6. Disc quota: Usually a certain amount of disk space is allocated by default. In cases where the situation so warrants, a user may seek additional disk space. A user may interrogate the disk space available at any time by using the *df* command. Its usage is shown below:

   df [options] [name] : to know the free disk space.

   where name refers to a mounted file system, local or remote. We may specify directory if we need to know the information about that directory. The following options may help with additional information:

   -l : for local file system
   -t : reports total no. of allocated blocks and i-nodes on the device.

   The Unix command *du* reports the number of disk blocks occupied by a file. Its usage is shown below:

   du [options] [name]... where name is a directory or a file

   Above name by default refers to the current directory. The following options may help with additional information:

   -a : produce output line for each file
   -s : report only the total usage for each name that is a directory i.e. not individual files.
   -r : produce messages for files that cannot be read or opened

7. Network services: Usually a user shall get a mail account. We will discuss the role of .mailrc file in this context in section 1.5. The user gets an access to Web services too.
8. Default terminal settings: Usually vt100 is the default terminal setting. One can attempt alternate terminal settings using *tset*, *stty*, *tput*, tabs with the control sequences defined in *terminfo termcap* with details recorded in /etc/ttytype or /etc/tty files and in shell variable TERM. Many of these details are discussed in Section 1.6.1 which specifically deals with terminal settings. The reader is encouraged to look up that section for details.

Once an account has been opened the user may do the following:

1. Change the pass-word for access to one of his liking.
2. Customize many of the run command files to suit his needs.

Closing a user account: Here again the password file plays a role. Recall in section 1.2 we saw that /etc/password file has all the information about the users' home directory, password, shell, user and group-id, etc. When a user's account is to be deleted, all of this information needs to be erased. System administrators login as root and delete the user entry from the password file to delete the account.

1.5 The *.rc* Files

Usually system administration offers a set of start-up run command files to a new user. These are files that appear as .rc files. These may be *.profile, .login, .cshrc, .bashrc .xinitrc, .mailrc .ircrc,* etc. The choice depends upon the nature of the login shell. Typical allocations may be as follows:

0 Bourne or Korn shell: *.profile*
1 C-Shell: *.login, .cshrc*
2 BASH: *.bashrci*
3 TCSH: *.tcshrc*

BASH is referred as Bourne-again shell. TCSH is an advanced C-Shell with many shortcuts like pressing a tab may complete a partial string to the extent it can be covered unambiguously. For us it is important to understand what is it that these files facilitate.

Role of *.login* and *.profile* files: The basic role of these files is to set up the *environment* for a user. These may include the following set-ups.

- Set up the terminal characteristics: Usually, the set up may include terminal type, and character settings for the prompt, erase, etc.
- Set up editors: It may set up a default editor or some specific editor like emacs.
- Set up protection mode: This file may set up umask, which stands for the user mask. umask determines access right to files.
- Set up environment variables: This file may set up the path variable. The path variable defines the sequence in which directories are searched for locating the commands and utilities of the operating system.
- Set up some customization variables: Usually, these help to limit things like selecting icons for mail or core dump size up to a maximum value. It may be used for setting up the limit on the scope of the command history, or some other preferences.

A typical *.login* file may have the following entries:

```
# A typical .login file
umask 022
setenv PATH /usr/ucb:/usr/bin:/usr/sbin:/usr/local/bin
setenv PRINTER labprinter

setenv EDITOR vi
biff y
set prompt='hostname'=>
```

The meanings of the lines above should be obvious from the explanation we advanced earlier. Next we describe *.cshrc* files and the readers should note the commonalities between these definitions of initialisation files.

The *.cshrc* file: The C-shell makes a few features available over the Bourne shell. For instance, it is common to define aliases in *.cshrc* files for very frequently used commands like *gh* for *ghostview* and *c* for *clear*. Below we give some typical entries for *.cshrc* file in addition to the many we saw in the *.login* file in this section:

```
if (! $?TERM) setenv TERM unknown
if ("TERM" == "unknown" || "$TERM" == "network") then echo -n 'TERM?
[vt100]: ';
set ttype=($<)
if (ttype == "") set ttype="vt100"
if (ttype == "pc") then set ttype="vt100"
endif
setenv TERM $ttype
endif
alias cl clear
alias gh ghostview
set history = 50
set nobeep
```

Note that the above, in the first few lines in the script, system identifies the nature of terminal and sets it to operate as vt100. It is highly recommended that the reader should examine and walk-through the initialization scripts which the system administration provides. Also, a customization of these files entails that as a user we must look up these files and modify them to suit our needs.

There are two more files of interest. One corresponds to regulating the mail and the other which controls the screen display. These are respectively initialized through *.mailrc* and *.xinitrc*. We discussed the latter in the chapter on X Windows. We shall discuss the settings in *.mailrc* file in the context of the mail system.

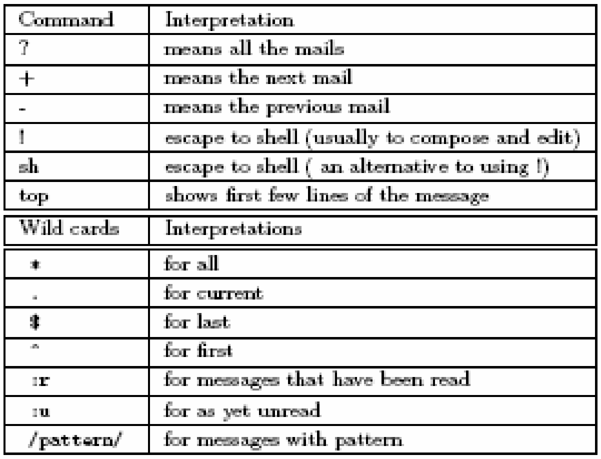
The mail system: *.mailrc* file : From the viewpoint of the user's host machine, the mail program truly acts as the main anchor for our internet-based communication. The Unix sendmail program together with the uu class of programs form the very basis of the mail under Unix. Essentially, the mail system has the following characteristics:

1. The mail system is a *Store and forward* system.
2. Mail is picked up from the mail server periodically. The *mail* daemon, picks up the mail running as a background process.
3. Mail is sent by *sendmail* program under Unix.
4. The *uu* class of programs like *uucp* or Unix-to-Unix copy have provided the basis for developing the mail tools. In fact, the file attachments facility is an example of it.

On a Unix system it is possible to invoke the mail program from an auto-login or .cshrc program.

Every Unix user has a mailbox entry in the /usr/spool/mail directory. Each person's mail box is named after his own username. In Table 19.1 we briefly review some very useful mail commands and the wild card used with these commands.

We next give some very useful commands which help users to manage their mails efficiently:



Command	Interpretation
?	means all the mails
+	means the next mail
-	means the previous mail
!	escape to shell (usually to compose and edit)
sh	escape to shell (an alternative to using !)
top	shows first few lines of the message

Wild cards	Interpretations
*	for all
.	for current
$	for last
^	for first
:r	for messages that have been read
:u	for as yet unread
/pattern/	for messages with pattern

Table 19.1: Various command options for mail.

d:r : delete all read messages.
d:usenet : delete all messages with usenet in body
p:r : print all read messages.
p:bhatt : print all from user ``bhatt".

During the time a user is composing a mail, the mail system tools usually offer facility to escape to a shell. This can be very useful when large files need to be edited along side the mail being sent. These use ~ commands with the interpretations shown below:

~! escape to shell,
~d include dead.letter
~h edit header field

The mail system provides for command line interface to facilitate mail operations using some of the following commands. For instance, every user has a default mail box called mbox. If one wishes to give a different name to the mailbox, he may choose a new name for it. Other facilities allow a mail to be composed with, or without, a subject or to see the progress of the mail as it gets processed. We show some of these options and their usage with mail command below.

mail -s greetings user@machine.domain

-s: option is used to send a mail with subject.

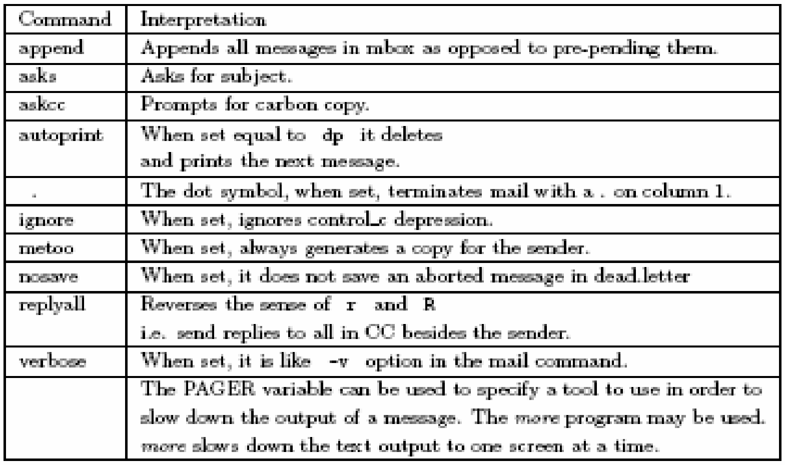
-v: option is for the verbose option, it shows mails' progress

-f mailbox: option allows user to name a new mail box

mail -f newm: where newm may be the new mail box option which a user may opt for in place of mbox (default option).

Next we describe some of the options that often appear inside *.mailrc* user files. Generally, with these options we may have aliases (nick-names) in place of the full mail address. One may also set or unset some flags as shown in the example below:

unset askcc
set verbose
set append



Command	Interpretation
append	Appends all messages in mbox as opposed to pre-pending them.
asks	Asks for subject.
askcc	Prompts for carbon copy.
autoprint	When set equal to dp it deletes and prints the next message.
.	The dot symbol, when set, terminates mail with a . on column 1.
ignore	When set, ignores control_c depression.
metoo	When set, always generates a copy for the sender.
nosave	When set, it does not save an aborted message in dead.letter
replyall	Reverses the sense of r and R i.e. send replies to all in CC besides the sender.
verbose	When set, it is like -v option in the mail command.
	The PAGER variable can be used to specify a tool to use in order to slow down the output of a message. The *more* program may be used. *more* slows down the text output to one screen at a time.

Table 19.2: Various options for .mailrc file.

In Table 19.2, we offer a brief explanation of the options which may be set initially in .mailrc files.

In addition, in using the mail system the following may be the additional facilities which could be utilized:

1. To subscribe to listserv@machine.domain, the body of the message should contain “subscribe", the group to subscribe to and the subscribers' e-mail address as shown in the following example.

```
subscribe allmusic me@mymachine.mydomain.
```

2. To unsubscribe use logout allmusic. In addition to the above there are vacation programs which send mails automatically when the receiver is on vacation. Mails may also be encrypted. For instance, one may use a pretty good privacy (PGP) for encrypting mails.

Facilitating chat with *.ircrc* **file:** System administrators may prepare terminals and offer Inter Relay Chat or IRC facility as well. IRC enables real-time conversation with one or more persons who may be scattered anywhere globally. IRC is a multi-user system. To use IRC's, Unix-based IRC versions, one may have to set the terminal emulation to vt100 either from the keyboard or from an auto-login file such as .login in bin/sh or .cshrc in /bin/csh.

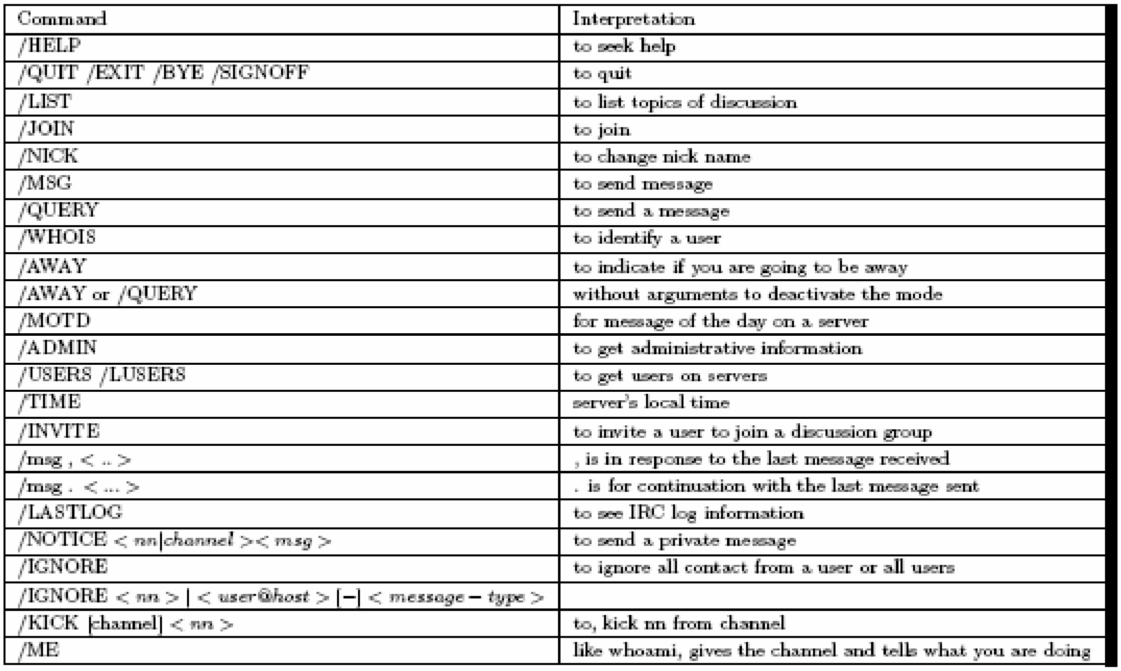
```
$ set TERM=vt100
$ stty erase "^h"
```

The most common way to use the IRC system is to make a telnet call to the IRC server. There are many IRC servers. Some servers require specification of a port number as in irc.ibmpcug.co.uk9999.

When one first accesses the IRC server, many channels are presented. A channel may be taken as a discussion area and one may choose a channel for an online chat (like switch a channel on TV). IRCs require setting up an .ircrc file. Below we give some sample entries for a .ircrc file. The .ircrc files may also set internal variables.

```
/COMMENT .....
/NICK <nn>
/JOIN <ch>
```

IRC commands begin with a \/" character. In Table 19.3, we give a few of the commands for IRC with their interpretations.



Command	Interpretation
/HELP	to seek help
/QUIT /EXIT /BYE /SIGNOFF	to quit
/LIST	to list topics of discussion
/JOIN	to join
/NICK	to change nick name
/MSG	to send message
/QUERY	to send a message
/WHOIS	to identify a user
/AWAY	to indicate if you are going to be away
/AWAY or /QUERY	without arguments to deactivate the mode
/MOTD	for message of the day on a server
/ADMIN	to get administrative information
/USERS /LUSERS	to get users on servers
/TIME	server's local time
/INVITE	to invite a user to join a discussion group
/msg , < .. >	, is in response to the last message received
/msg . < ... >	. is for continuation with the last message sent
/LASTLOG	to see IRC log information
/NOTICE < *nn|channel* >< *msg* >	to send a private message
/IGNORE	to ignore all contact from a user or all users
/IGNORE < *nn* > | < *user@host* > [−] < *message − type* >	
/KICK [channel] < *nn* >	to, kick nn from channel
/ME	like whoami, gives the channel and tells what you are doing

Table 19.3: Various commands with interpretation.

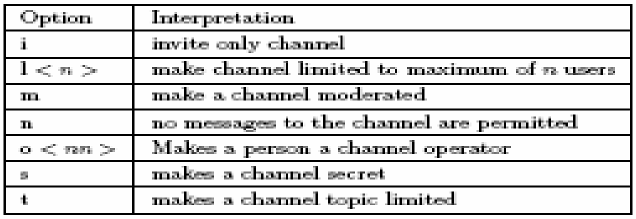
IRCs usually support a range of channels. Listed below are a few of the channel types:

Limbo or Null
Public
Private
Secret
Moderated

Limited
Topic limited
Invite Only
Message disabled.

The above channel types are realized by using a mode command. The modes are set or unset as follows. The options have the interpretations shown in Table 19.4.

MODE sets (with +) and unsets (with -) the mode of channel with the following options
/MODE <channel> +<channel options> < parameters>
/MODE <channel> -<channel options> < parameters>



Option	Interpretation
i	invite only channel
l < *n* >	make channel limited to maximum of *n* users
m	make a channel moderated
n	no messages to the channel are permitted
o < *nn* >	Makes a person a channel operator
s	makes a channel secret
t	makes a channel topic limited

Table 19.4: Various options for channels.

1.5.1 Sourcing Files

As we have described above, the .rc files help to provide adequate support for a variety of services. Suppose we are logged to a system and seek a service that requires a change in one of the *.rc* files. We may edit the corresponding file. However, to affect the changed behavior we must source the file. Basically, we need to execute the source command with the file name as argument as shown below where we source the *.cshrc* file:

```
source .cshrc
```

1.6 Device Management and Services

Technically the system administrator is responsible for every device, for all of its usage and operation. In particular, the administrator looks after its installation, upgrade, configuration, scheduling, and allocating quotas to service the user community. We shall, however, restrict ourselves to the following three services:

1. Terminal-based services, discussed in Section 1.6.1.
2. Printer services, discussed in Section 1.6.2.
3. Disc space and file services, discussed in Section 1.6.3.

We shall begin with the terminal settings and related issues.

1.6.1 The Terminal Settings

In the context of terminal settings the following three things are important:

1. Unix recognizes terminals as special files.
2. Terminals operate on serial lines. Unix has a way to deal with files that are essentially using serial communication lines.
3. The terminals have a variety of settings available. This is so even while the protocols of communication for all of them are similar.

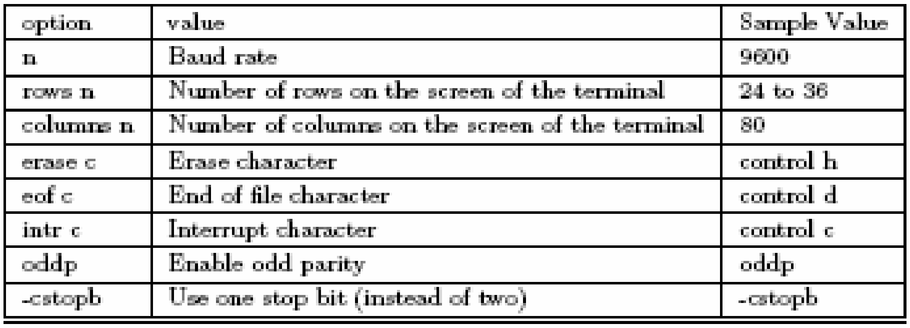
From the point of terminal services provisioning and system configuration, system administration must bear the above three factors in mind. Unix maintains all terminal related information in *tty* files in /etc/dev directory. These files are special files which adhere to the protocols of communication with serial lines. This includes those terminals that use modems for communication. Some systems may have a special file for console like /etc/dev/console which can be monitored for messages as explained in the chapter on X-Windows. Depending upon the terminal type a serial line control protocol is used which can interrogate or activate appropriate pins on the hardware interface plug.

The following brief session shows how a terminal may be identified on a host:

```
login: bhatt
Password:
Last login: Tue Nov 5 00:25:21 from 203.197.175.174
[bhatt@iiitbsun bhatt]$hostname
iiitbsun
[bhatt@iiitbsun bhatt]$tty
/dev/pts/1
[bhatt@iiitbsun bhatt]$
```

termcap **and** *terminfo* **files:** The *termcap* and *terminfo* files in the directory /etc or in */usr/share/lib/terminfo* provide the terminal database, information and programs for use in the Unix environment. The database includes programs that may have been compiled to elicit services from a specific terminal which may be installed. The programs that control the usage of a specific terminal are identified in the environment variable TERM as shown in the example below:

```
[bhatt@localhost DFT02]$ echo $TERM
xterm
[bhatt@localhost DFT02]$
```



option	value	Sample Value
n	Baud rate	9600
rows n	Number of rows on the screen of the terminal	24 to 36
columns n	Number of columns on the screen of the terminal	80
erase c	Erase character	control h
eof c	End of file character	control d
intr c	Interrupt character	control c
oddp	Enable odd parity	oddp
-cstopb	Use one stop bit (instead of two)	-cstopb

Table 19.5: Options under *stty*.

There are specific commands like tic, short for terminal information compilation. Also, there are programs that convert *termcap* to *terminfo* whenever required. For detailed discussions on terminal characteristics and how to exploit various features the reader may refer to [2]. We shall, however, elaborate on two specific commands here.

These are the tset and stty commands.

1. ***tset* Command**: The tset command is used to initialize a terminal. Usually, the command sets up initial settings for characters like erase, kill, etc. Below we show how under C-Shell one may use the tset command:

   ```
   $setenv TERM `tset - Q -m ":?vt100"
   ```

   Sometimes one may prepare a temporary file and source it.

2. ***stty* command**: We briefly encountered the *stty* command in Section 1.3. Here we shall elaborate on *stty* command in the context of options and the values which may be availed by using the *stty* command. In Table 19.5 we list a few of the options with their corresponding values.

There are many other options. In Table 19.5 we have a sample of those that are available. Try the command *stty* -a to see the options for your terminal. Below is shown the setting on my terminal:

```
[bhatt@localhost DFT02]$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc ixany imaxbel

opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
[bhatt@localhost DFT02]$
```

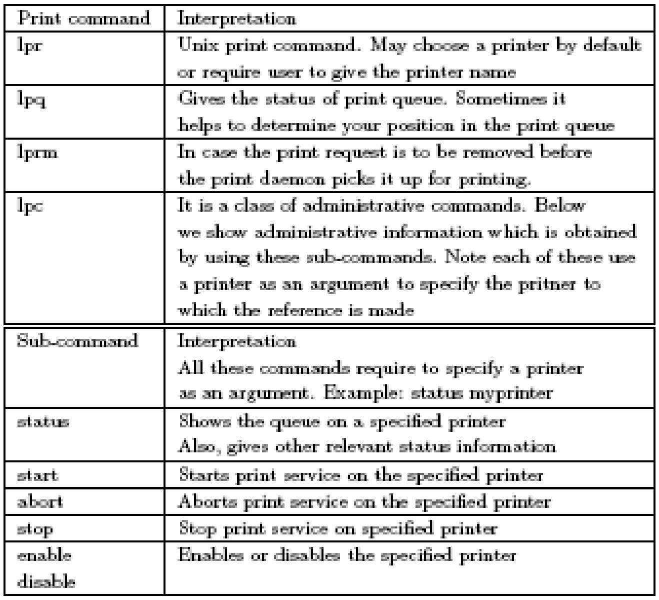
Lastly, we discuss how to attach a new terminal. Basically we need to connect a terminal and then we set-up the entries in *termcap* and/or in *terminfo* and configuration files. Sometimes one may have to look at the /etc/inittab or /etc/ttydefs as well. It helps to reboot the system on some occasions to ensure proper initialization following a set-up attempt.

1.6.2 Printer Services

Users obtain print services through a printer daemon. The system arranges to offer print services by spooling print jobs in a spooling directory. It also has a mechanism to service the print requests from the spooling directory. In addition, system administrators need to be familiar with commands which help in monitoring the printer usage. We shall begin with a description of the *printcap* file.

The printcap file: Unix systems have their print services offered using a spooling system. The spooling system recognizes print devices that are identified in */etc/printcap* file. The *printcap* file serves not only as a database, but also as a configuration file. Below we see the *printcap* file on my machine:

```
# /etc/printcap
#
# DO NOT EDIT! MANUAL CHANGES WILL BE LOST!
# This file is autogenerated by printconf-backend during lpd init.
#
# Hand edited changes can be put in /etc/printcap.local, and will be included.
iiitb:\
:sh:\
:ml=0:\
:mx=0:\
:sd=/var/spool/lpd/iiitb:\
:lp=|/usr/share/printconf/jetdirectprint:\
:lpd_bounce=true:\

:if=/usr/share/
printconf/mf_wrap
per:
```



Print command	Interpretation
lpr	Unix print command. May choose a printer by default or require user to give the printer name
lpq	Gives the status of print queue. Sometimes it helps to determine your position in the print queue
lprm	In case the print request is to be removed before the print daemon picks it up for printing.
lpc	It is a class of administrative commands. Below we show administrative information which is obtained by using these sub-commands. Note each of these use a printer as an argument to specify the pritner to which the reference is made
Sub-command	Interpretation All these commands require to specify a printer as an argument. Example: status myprinter
status	Shows the queue on a specified printer Also, gives other relevant status information
start	Starts print service on the specified printer
abort	Aborts print service on the specified printer
stop	Stop print service on specified printer
enable disable	Enables or disables the specified printer

Table 19.6: The *printcap* file: printer characteristics.

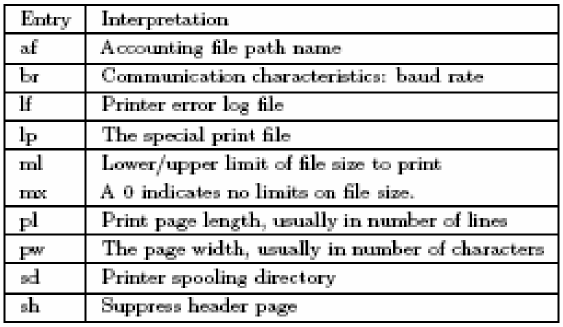
The *printcap* file is a read-only file except that it can be edited by super user ROOT.

Printer spooling directory: As we explained earlier, print requests get spooled first. Subsequently, the printer daemon *lpd* honours the print request to print. To achieve this, one may employ a two layered design. Viewing it bottom up, at the bottom layer maintain a separate spooling directory for each of the printers. So, when we attach a new printer, we must create a new spooling directory for it. At the top level, we have a spooling process which receives each print request and finally spools it for printer(s).

Note that the owner of the spool process is a group *daemon.*

The entries in *printcap* files can be explained using Table 19.6. With the file description and the table we can see that the spooling directory for our printer, with printer name iiitb is at /var/spool. Also note we have no limit on file size which can be printed.

Printer monitoring commands: The printer commands help to monitor both the health of the services as also the work in progress. In table 19.7 we elaborate on the commands and their interpretations.



Entry	Interpretation
af	Accounting file path name
br	Communication characteristics: baud rate
lf	Printer error log file
lp	The special print file
ml mx	Lower/upper limit of file size to print A 0 indicates no limits on file size.
pl	Print page length, usually in number of lines
pw	The page width, usually in number of characters
sd	Printer spooling directory
sh	Suppress header page

Table 19.7: The printer commands.

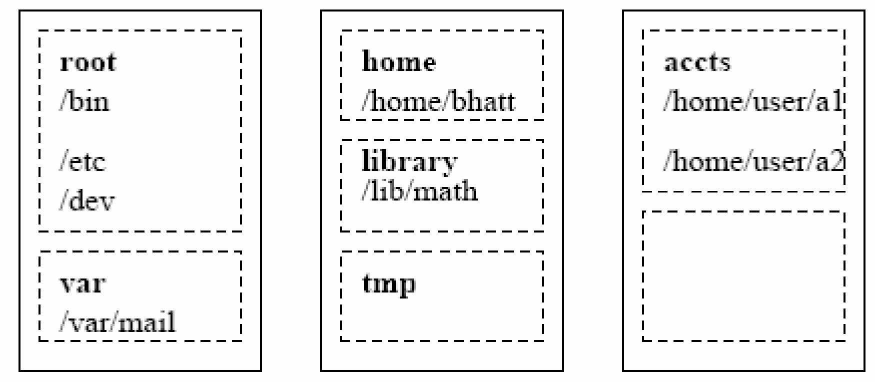
To add a printer one may use a *lpadmin* tool. Some of the system administration practices are best learned by assisting experienced system administrators rarely can be taught through a textbook.

1.6.3 Disk space allocation and management

In this section we shall discuss how does a system administrator manage the disk space. We will The partitions may be physical or logical. In case of a physical partition we have the file system resident within one disk drive. In case of logical partition, the file system may extend over several drives. In either of these cases the following issues are at stake:

1. **Disk file system:** In Chapter 2 we indicated that system files are resident in the root file system. Similarly, the user information is maintained in home file system created by the administrator. Usually, a physical disk drive may have one or more file systems resident

on it. As an example, consider the mapping shown in Figure 19.1. We notice that there are three physical drives with mapping or root and



The names of file systems are shown in **bold l**etters.

Figure 19.1: Mapping file systems on physical drives.

other file systems. Note that the disk drive with the root file system co-locates the *var* file system on the same drive. Also, the file system home extends over two drives. This is possible by appropriate assignment of the disk partitions to various file systems. Of course, system programmers follow some method in both partitioning and allocating the partitions. Recall that each file system maintains some data about each of the files within it.

System administrators have to reallocate the file systems when new disks become available, or when some disk suffers damage to sectors or tracks which may no longer be available.

2. **Mounting and unmounting:** The file systems keep the files in a directory structure which is essentially a tree. So a new file system can be created by specifying the point of mount in the directory tree. A typical *mount* instruction has the following format.

   ```
   mount a-block-special-file point-of-mount
   ```

   Corresponding to a *mount* instruction, there is also an instruction to unmount. In Unix it is *umount* with the same format as *mount.*

   In Unix every time we have a new disk added, it is mounted at a suitable point of mount in the directory tree. In that case the mount instruction is used exactly as explained. Of course, a disk is assumed to be formatted.

3. **Disk quota:** Disk quota can be allocated by reconfiguring the file system usually located at */etc/fstab*. To extend the allocation quota in a file system we first have to modify the

corresponding entry in the */etc/fstab* file. The system administration can set hard or soft limits of user quota. If a hard limit has been set, then the user simply cannot exceed the allocated space. However, if a soft limit is set, then the user is cautioned when he approaches the soft limit. Usually, it is expected that the user will resort to purging files no longer in use. Else he may seek additional disk space. Some systems have quota set at the group level. It may also be possible to set quota for individual users. Both these situations require executing an edit quota instruction with user name or group name as the argument. The format of *edquota* instruction is shown below.

edquota user-name

4. Integrity of file systems: Due to the dynamics of temporary allocations and moving files around, the integrity of a file system may get compromised. The following are some of the ways the integrity is lost:

   - Lost files. This may happen because a user ahs opened the same file from multiple windows and edited them.
   - A block may be marked free but may be in use.
   - A block may be marked in use but may be free.
   - The link counts may not be correct.
   - The data in the file system table and actual files may be different.

   The integrity of the file system is checked out by using a *fsck* instruction. The argument to the command is the file system which we need to check as shown below.

   fsck file-system-to-be-checked

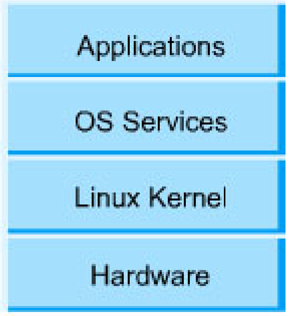
   On rebooting the system these checks are mandatory and routinely performed.Consequently, the consistency of the file system is immediately restored on rebooting.

5. Access control: As explained earlier in this chapter, when an account is opened, a user is allocated a group. The group determines the access. It is also possible to offer an initial set-up that will allow access to special (licensed) software like matlab suite of software.
6. Periodic back-up: Every good administrator follows a regular back-up procedure so that in case of a severe breakdown, at least a stable previous state can be achieved.

Unit-5

1.1 More on LINUX (Linux Kernel Architecture)

It is a good idea to look at the Linux kernel within the overall system's overall context..



Applications and OS services:

These are the user application running on the Linux system. These applications are not fixed but typically include applications like email clients, text processors etc. OS services include utilities and services that are traditionally considered part of an OS like the windowing system, shells, programming interface to the kernel, the libraries and compilers etc.

1.1.1 Linux Kernel:

Kernel abstracts the hardware to the upper layers. The kernel presents the same view of the hardware even if the underlying hardware is different. It mediates and controls access to system resources.

1.1.2 Hardware

This layer consists of the physical resources of the system that finally do the actual work. This includes the CPU, the hard disk, the parallel port controllers, the system RAM etc.

1.1.3 The Linux Kernel

After looking at the big picture we should zoom into the Linux kernel to get a closer look.

1.1.4 Purpose of the Kernel

The Linux kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge (most of the time) of the type of the physical hardware that constitutes the computer. The Linux kernel abstracts all hardware into a consistent interface.

In addition, Linux Kernel supports multi-tasking in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and mediates access to hardware resources so that each process has fair access while inter-process security is maintained.

The kernel code executes in privileged mode called kernel mode. Any code that does not need to run in privileged mode is put in the system library. The interesting thing about Linux kernel is that it has a modular architecture – even with binary codes: Linux kernel can load (and unload) modules dynamically (at run time) just as it can load or unload the system library modules.

Here we shall explore the conceptual view of the kernel without really bothering about the implementation issues (which keep on constantly changing any way). Kernel code provides for arbitrations and for protected access to HW resources. Kernel supports services for the applications through the system libraries. System calls within applications (may be written in C) may also use system library. For instance, the buffered file handling is operated and managed by Linux kernel through system libraries. Programs like utilities that are needed to initialize the system and configure network devices are classed as user mode programs and do not run with kernel privileges (unlike in Unix).

Programs like those that handle login requests are run as system utilities and also do not require kernel privileges (unlike in Unix).

1.2 The Linux Kernel Structure Overview

The “loadable” kernel modules execute in the privileged kernel mode – and therefore have the capabilities to communicate with all of HW.

Linux kernel source code is free. People may develop their own kernel modules. However, this requires recompiling, linking and loading. Such a code can be distributed under GPL. More often the modality is:

Start with the standard minimal basic kernel module. Then enrich the environment by the addition of customized drivers.

This is the route presently most people in the embedded system area are adopting worldwide.

The commonly loaded Linux system kernel can be thought of comprising of the following main components:

1.2.1 Process Management

User process as also the kernel processes seek the cpu and other services. Usually a fork system call results in creating a new process. System call *execve* results in execution of a newly forked process. Processes, have an id (PID) and also have a user id (UID) like in Unix. Linux additionally has a personality associated with a process. Personality of a process is used by emulation libraries to be able to cater to a range of implementations. Usually a forked process inherits parent's environment.

In Linux Two vectors define a process: these are argument vector and environment vector. The environment vector essentially has a (name, value) value list wherein different environment variable values are specified. The argument vector has the command line arguments used by the process. Usually the environment is inherited however, upon execution of *execve* the process body may be redefined with a new set of environment variables. This helps in the customization of a process's operational environment. Usually a process also has some indication on its scheduling context.

Typically a process context includes information on scheduling, accounting, file tables, capability on signal handling and virtual memory context.

In Linux, internally, both processes and threads have the same kind of representation. Linux processes and threads are POSIX compliant and are supported by a threads library package which provides for two kinds of threads: user and kernel. User-controlled scheduling can be used for user threads. The kernel threads are scheduled by the kernel.

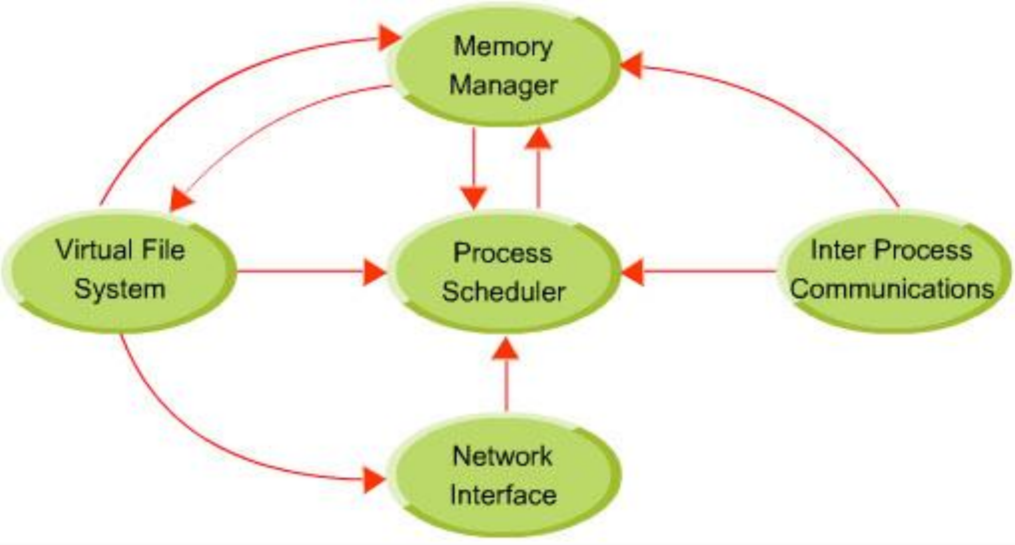
While in a single processor environment there can be only one kernel thread scheduled. In a multiprocessor environment one can use the kernel supported library and clone system call to have multiple kernel threads created and scheduled.

1.2.2 Scheduler

Schedulers control the access to CPU by implementing some policy such that the CPU is

shared in a way that is fair and also the system stability is maintained. In Linux scheduling is required for the user processes and the kernel tasks. Kernel tasks may be internal tasks on behalf of the drivers or initiated by user processes requiring specific OS services. Examples are: a page fault (induced by a user process) or because some device driver raises an interrupt. In Linux, normally, the kernel mode of operation cannot be pre-empted. Kernel code runs to completion -

unless it results in a page fault, or an interrupt of some kind or kernel code itself calls the scheduler. Linux is a time sharing system. So a timer interrupt happens and rescheduling may be initiated at that time. Linux uses a credit based scheduling algorithm. The process with the highest credits gets scheduled. The credits are revised after every run. If all run-able processes exhaust all the credits a priority based fresh credit allocation takes place. The crediting system usually gives higher credits to interactive or IO bound processes – as these require immediate responses from a user. Linux also implements Unix like nice process characterization.



1.2.3 The Memory Manager

Memory manager manages the allocation and de-allocation of system memory amongst the processes that may be executing concurrently at any time on the system. The memory manager ensures that these processes do not end up corrupting each other's memory area.

Also, this module is responsible for implementing virtual memory and the paging mechanism within it. The loadable kernel modules are managed in two stages:

First the loader seeks memory allocation from the kernel. Next the kernel returns the address of the area for loading the new module.

- The linking for symbols is handled by the compiler because whenever a new module is loaded recompilation is imperative.

1.2.4 The Virtual File System (VFS)

Presents a consistent file system interface to the kernel. This allows the kernel code to be independent of the various file systems that may be supported (details on virtual file system VFS follow under the files system).

1.2.5 The Network Interface

Provides kernel access to various network hardware and protocols.

1.3 Inter Process Communication

The IPC primitives for processes also reside on the same system. With the explanation above we should think of the typical loadable kernel module in Linux to have three main components:

- Module management,
- Driver registration and
- Conflict resolution mechanism.

Module Management

For new modules this is done at two levels – the management of kernel referenced symbols and the management of the code in kernel memory. The Linux kernel maintains a symbol table and symbols defined here can be exported (that is these definitions can be used elsewhere) explicitly. The new module must seek these symbols. In fact this is like having an external definition in C and then getting the definition at the kernel compile time. The module management system also defines all the required communications interfaces for this newly inserted module. With this done, processes can request the services (may be of a device driver) from this module.

Driver registration

The kernel maintains a dynamic table which gets modified once a new module is added – some times one may wish to delete also. In writing these modules care is taken to ensure that initializations and cleaning up operations are defined for the driver. A module may register one or more drivers of one or more types of drivers. Usually the registration of drivers is maintained in a registration table of the module.

The registration of drives entails the following:

1. Driver context identification: as a character or bulk device or a network driver.

2. File system context: essentially the routines employed to store files in Linux virtual file system or network file system like NFS.
3. Network protocols and packet filtering rules.
4. File formats for executable and other files.

Conflict Resolution

The PC hardware configuration is supported by a large number of chip set configurations and with a large range of drivers for SCSI devices, video display devices and adapters, network cards. This results in the situation where we have module device drivers which vary over a very wide range of capabilities and options. This necessitates a conflict resolution mechanism to resolve accesses in a variety of conflicting concurrent accesses. The conflict resolution mechanisms help in preventing modules from having an access conflict to the HW – for example an access to a printer. Modules usually identify the HW resources it needs at the time of loading and the kernel makes these available by using a reservation table. The kernel usually maintains information on the address to be used for accessing HW - be it DMA channel or an interrupt line. The drivers avail kernel services to access HW resources.

1.4 System Calls

Let us explore how system calls are handled. A user space process enters the kernel. From this point the mechanism is somewhat CPU architecture dependent. Most common examples of system calls are: - open, close, read, write, exit, fork, exec, kill, socket calls etc.

The Linux Kernel 2.4 is non preemptable. Implying once a system call is executing it will run till it is finished or it relinquishes control of the CPU. However, Linux kernel 2.6 has been made partly preemptable. This has improved the responsiveness considerably and the system behavior is less ‘jerky’.

1.4.1 Systems Call Interface in Linux

System call is the interface with which a program in user space program accesses kernel functionality. At a very high level it can be thought of as a user process calling a function in the Linux Kernel. Even though this would seem like a normal C function call, it is in fact handled differently. The user process does not issue a system call directly - instead, it is internally invoked by the C library.

Linux has a fixed number of system calls that are reconciled at compile time. A user process can access only these finite set of services via the system call interface. Each system call has a unique identifying number. The exact mechanism of a system call implementation is platform dependent. Below we discuss how it is done in the x86 architecture.

To invoke a system call in x86 architecture, the following needs to be done. First, a system call number is put into the EAX hardware register. Arguments to the system call are put into other hardware registers. Then the int0x80 software interrupt is issued which then invokes the kernel service.

Adding one’s own system call is a pretty straight forward (almost) in Linux. Let us try to implement our own simple system call which we will call ‘simple’ and whose source we will put in simple.c.

```
/* simple.c */
/* this code was never actually compiled and tested */
#include<linux/simple.h>
asmlinkage int sys_simple(void)
{
return 99;
}
```

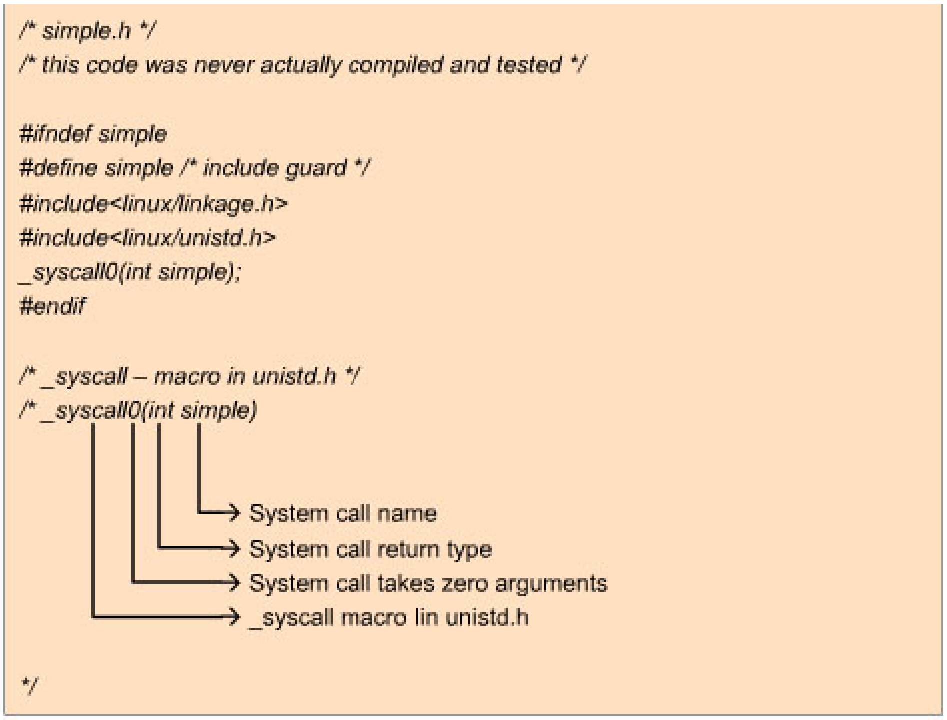
As can be seen that this a very dumb system call that does nothing but return 99. But that is enough for our purpose of understanding the basics.

This file now has to be added to the Linux source tree for compilation by executing:

/usr/src/linux.*.*/simple.c

Those who are not familiar with kernel programming might wonder what “asmlinkage” stands for in the system call. ‘C’ language does not allow access hardware directly. So, some assembly code is required to access the EAX register etc. The asmlinkage macro does the dirty work fortunately.

The asmlinkage macro is defined in *XXXX/linkage.h.* It initiates another macro_syscall in *XXXXX/unistd.h.* The header file for a typical system call will contain the following.



```
/* simple.h */
/* this code was never actually compiled and tested */

#ifndef simple
#define simple /* include guard */
#include<linux/linkage.h>
#include<linux/unistd.h>
_syscall0(int simple);
#endif

/* _syscall – macro in unistd.h */
/* _syscall0(int simple)
```

System call name
System call return type
System call takes zero arguments
_syscall macro lin unistd.h

```
*/
```

After defining the system call we need to assign a system call number. This can be done by adding a line to the file unistd.h . unistd.h has a series of #defines of the form:

#define _NR_sys_exit 1

Now if the last system call number is 223 then we enter the following line at the bottom

#define _NR_sys_simple 224

After assigning a number to the system call it is entered into system call table. The system call number is the index into a table that contains a pointer to the actual routine. This table is defined in the kernel file ‘entry.S’ .We add the following line to the file :

* this code was never actually compiled and tested

*/.long SYSMBOL_NAME(sys_simple)

Finally, we need to modify the makefile so that our system call is added to the kernel when it is compiled. If we look at the file /usr/src/linux.*.*/kernel/Makefile we get a line of the following format.

obj_y= sched.o + dn.o …….etc we add: obj_y += simple.o

Now we need to recompile the kernel. Note that there is no need to change the config file. With the source code of the Linux freely available, it is possible for users to make their own versions of the kernel. A user can take the source code select only the parts of the kernel that are relevant to him and leave out the rest. It is possible to get a working Linux kernel in single 1.44 MB floppy disk. A user can modify the source for the kernel so that the kernel suits a targeted application better. This is one of the reasons why Linux is the successful (and preferred) platform for developing embedded systems In fact, Linux has reopened the world of system programming.

1.5 The Memory Management Issues

The two major components in Linux memory management are:

- The page management
- The virtual memory management

1. The page management: Pages are usually of a size which is a power of 2. Given the main memory Linux allocates a group of pages using a buddy system. The allocation is the responsibility of a software called "page allocator". Page allocator software is responsible for both allocation, as well as, freeing the memory. The basic memory allocator uses a buddy heap which allocates a contiguous area of size 2*n* > the required memory with minimum *n* obtained by successive generation of "buddies" of equal size. We explain the buddy allocation using an example.

   An Example: Suppose we need memory of size 1556 words. Starting with a memory size 16K we would proceed as follows:

   1. First create 2 buddies of size 8k from the given memory size ie. 16K
   2. From one of the 8K buddy create two buddies of size 4K each
   3. From one of the 4k buddy create two buddies of size 2K each.
   4. Use one of the most recently generated buddies to accommodate the 1556 size memory requirement.

   Note that for a requirement of 1556 words, memory chunk of size 2K words satisfies the property of being the smallest chunk larger than the required size.

   Possibly some more concepts on page replacement, page aging, page flushing and the changes done in Linux 2.4 and 2.6 in these areas.

2. **Virtual memory Management:** The basic idea of a virtual memory system is to expose address space to a process. A process should have the entire address space exposed to it to make an allocation or deallocation. Linux makes a conscious effort to allocate logically, "page aligned" contiguous address space. Such page aligned logical spaces are called regions in the memory.

Linux organizes these regions to form a binary tree structure for fast access. In addition to the above logical view the Linux kernel maintains the physical view ie maps the hardware page table entries that determine the location of the logical page in the exact location on a disk. The process address space may have private or shared pages. Changes made to a page require that locality is preserved for a process by maintaining a copy-on-write when the pages are private to the process where as these have to be visible when they are shared.

A process, when first created following a fork system call, finds its allocation with a new entry in the page table – with inherited entries from the parent. For any page which is shared amongst the processes (like parent and child), a reference count is maintained.

Linux has a far more efficient page swapping algorithm than Unix – it uses a second chance algorithm dependent on the usage pattern. The manner it manifests it self is that a page gets a few chances of survival before it is considered to be no longer useful.

Frequently used pages get a higher age value and a reduction in usage brings the age closer to zero – finally leading to its exit.

The Kernel Virtual Memory: Kernel also maintains for each process a certain amount of "kernel virtual memory" – the page table entries for these are marked "protected".

The kernel virtual memory is split into two regions. First there is a static region which has the core of the kernel and page table references for all the normally allocated pages that cannot be modified. The second region is dynamic - page table entries created here may point anywhere and can be modified.

Loading, Linking and Execution: For a process the execution mode is entered following an *exec* system call. This may result in completely rewriting the previous execution context – this, however, requires that the calling process is entitled an access to the called code. Once the check is through the loading of the code is initiated. Older versions of Linux used to load binary files in the a.out format. The current version also loads binary files in ELF format. The ELF format is flexible as it permits adding additional information for debugging etc. A process can be executed when all the needed library routines have also been linked to form an executable module. Linux supports dynamic linking. The dynamic linking is achieved in two stages:

1. First the linking process downloads a very small statically linked function – whose task is to read the list of library functions which are to be dynamically linked.

2. Next the dynamic linking follows - resolving all symbolic references to get a loadable executable.

1.6 Linux File Systems

Introduction

Linux retains most fundamentals of the Unix file systems. While most Linux systems retain Minix file systems as well, the more commonly used file systems are VFS and ext2FS which stand for virtual file system and extended file systems. We shall also examine some details of proc file system and motivation for its presence in Linux file systems.

As in other UNIXES in Linux the files are mounted in one huge tree rooted at /. The file may actually be on different drives on the same or on remotely networked machines. Unlike windows, and like unixes, Linux does not have drive numbers like A: B: C: etc.

The *mount* operation: The unixes have a notion of *mount* operation. The mount operation is used to attach a filesystem to an existing filesystem on a hard disk or any other block oriented device. The idea is to attach the filesystem within the file hierarchy at a specified mount point. The mount point is defined by the path name for an identified directory. If that mount point has contents before the mount operation they are hidden till the file system is un-mounted. The un-mount requires issuance of *umount* command.

Linux supports multiple filesystems. These include ext, ext2, xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660,sysv, hpfs, affs and ufs etc. More file systems will be supported in future versions of LINUX. All block capable devices like floppy drives, IDE hard disks etc. can run as a filesystem. The “look and feel” of the files is the same regardless of the type of underlying block media. The Linux filesystems treat nearly all media as if they are linear collection of blocks. It is the task of the device driver to translate the file system calls into appropriate cylinder head number etc. if needed. A single disk partition or the entire disk (if there are no partitions) can have only one filesystem. That is, you cannot have a half the file partition running EXT2 and the remaining half running FAT32. The minimum granularity of a file system is a hard disk partition.

On the whole the EXT2 filesystem is the most successful file system. It is also now a part of the more popular Linux Distributions. Linux originally came with the Minix filesystem which was quite primitive and 'academic' in nature. To improve the situation a new file system was designed for Linux in 1992 called the Exteneded File System or the EXT file system. Mr Remy Card (**Rémy Card, Laboratoire MASI--Institut Blaise Pascal, E-Mail: card@masi.ibp.fr**) further improved the system to offer the Extended File System -2 or the ext-2 file system. This was an important addition to Linux that was added along with the virtual file system which permitted Linux to interoperate with different file systems.

Description:

Basic File Systems concepts:

Every Linux file system implements the basic set of concepts that have been a part of the Unix filesystem along the lines described in “The Design of the Unix” Book by Maurice Bach. Basically, these concepts are that every file is represented by an inode. Directories are nothing but special files with a list of entries. I/O to devices can be handled by simply reading or writing into special files (Example: To read data from the serial port we can do cat /dev/ttyS0).

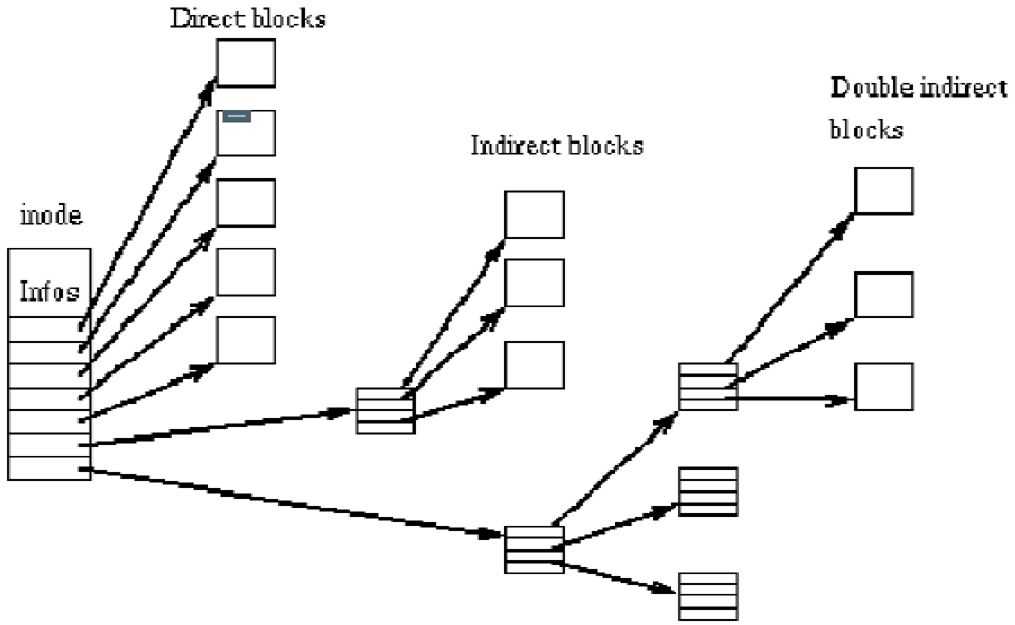
Superblock:

Super block contains the meta-data for the entire filesystem.

Inodes:

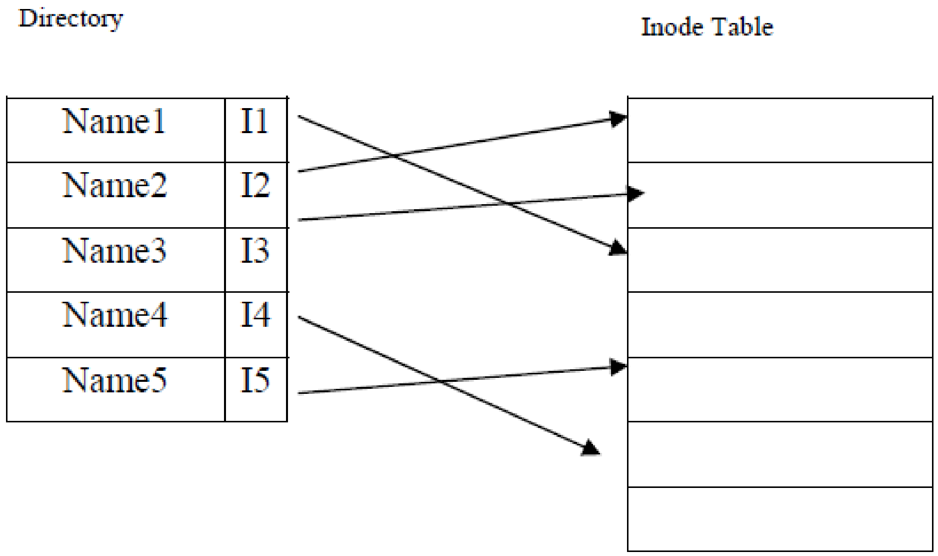
Each file is associated with a structure called an inode. Inode stores the attributes of the file which include File type, owner time stamp, size pointers to data blocks etc.

Whenever a file is accessed the kernel translates the offset into a block number and then uses the inode to figure out the actual address of the block. This address is then used to read/write to the actual physical block on the disk. The structure of an inode is as shown below in the figure.



Directories:

Directories are implemented as special files. Actually, a directory is nothing but a file containing a list of entries. Each entry contains a file name and a corresponding inode number. Whenever a path is resolved by the kernel it looks up these entries for the corresponding inode number. If the inode number is found it is loaded in the memory and used for further file access.



Name1	I1
Name2	I2
Name3	I3
Name4	I4
Name5	I5

Links:

UNIX operating systems implement the concept of links. Basically there are two types of links: Hard links and soft links. Hard link is just another entry in directory structure pointing to the same inode number as the file name it is linked to. The link count on the pointed inode is incremented. If a hard link is deleted the link count is decremented. If the

link count becomes zero the inode is deallocated if the linkcount becoms zero. It is impossible to have cross file systems hard links.

Soft links are just files which contain the name of the file they are pointing to. Whenever the kernel encounters a soft link in a path it replaces the soft-link with it contents and restarts the path resolution. With soft links it is possible to have cross file system links.

Softlinks that are not linked to absolute paths can lead to havoc in some cases. Softlinks also degrade system performance.

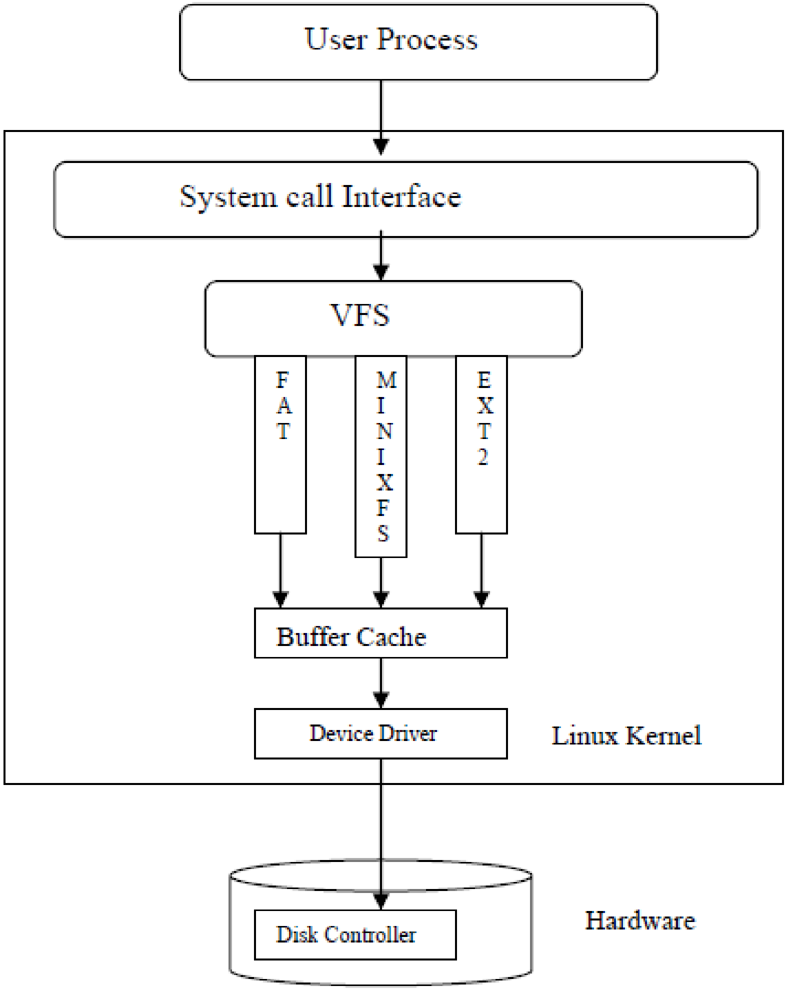
1.6.1 Device specific files

UNIX operating systems enable access to devices using special files. These file do not take up any space but are actually used to connect the device to the correct device driver. The device driver is located based on the major number associated with the device file. The minor number is passed to the device driver as an argument. Linux kernel 2.4 introduced a new file system for accessing device files called as the device file system. (Look at the section on device drivers)

1.6.2 The Virtual File system

When the Linux Kernel has to access a file system it uses a file system type independent interface, which allows the system to carry out operations on a File System without knowing its construction or type. Since the kernel is independent of File System type or construction, it is flexible enough to accommodate future File Systems as and when they become available.

Virtual File System is an interface providing a clearly defined link between the operating system kernel and the different File Systems.



1.7 The VFS Structure and file management in VFS:

For management of files, VFS employs an underlying definition for three kinds of objects:

1. inode object
2. file object
3. file system object

Associated with each type of object is a function table which contains the operations that can be performed. The function table basically maintains the addresses of the operational routines. The file objects and inode objects maintain all the access mechanism for each file's access. To access an inode object the process must obtain a pointer to it from the corresponding file object. The file object maintains from where a certain file is currently being read or written to ensure sequential

IO. File objects usually belong to a single process. The inode object maintains such information as the owner, time of file creation and modification.

The VFS knows about file-system types supported in the kernel. It uses a table defined during the kernel configuration. Each entry in this table describes filesystem type: it contains the name of the filesystem type and a pointer to a function called during the mount operation. When a file-system is to be mounted, the appropriate mount function is called. This function is responsible for reading the super-block from the disk, initializing its internal variables, and returning a mounted file-system descriptor to the VFS. The VFS functions can use this descriptor to access the physical file-system routines subsequently. A mounted file-system descriptor contains several kinds of data: information that is common to every file-system type, pointers to functions provided by the physical file-system kernel code, and private data maintained by the physical filesystem code. The function pointers contained in the file-system descriptors allow the VFS to access the file-system internal routines. Two other types of descriptors are used by the VFS: an inode descriptor and an open file descriptor. Each descriptor contains information related to files in use and a set of operations provided by the physical filesystem code. While the inode descriptor contains pointers to functions that can be used to act on any file (e.g. create, unlink), the file descriptors contains pointer to functions which can only act on open files (e.g. read, write).

1.8 The Second Extended File System (EXT2FS)

Standard Ext2fs features:

This is the most commonly used file system in Linux. In fact, it extends the original Minix FS which had several restrictions – such as file name length being limited to 14 characters and the file system size limited to 64 K etc. The ext2FS permits three levels of indirections to store really large files (as in BSD fast file system). Small files and fragments are stored in 1KB (kilo bytes) blocks. It is possible to support 2KB or 4KB blocks sizes. 1KB is the default size. The Ext2fs supports standard *nix file types: regular files, directories, device special files and symbolic links. Ext2fs is able to manage file systems created on really big partitions. While the original kernel code restricted the maximal file-system size to 2 GB, recent work in the VFS layer have raised this limit to 4 TB. Thus, it is now possible to use big disks without the need of creating many partitions.

Not only does Ext2fs provide long file names it also uses variable length directory entries. The maximal file name size is 255 characters. This limit could be extended to 1012, if needed. Ext2fs reserves some blocks for the super user (root). Normally, 5% of the blocks are reserved. This allows the administrator to recover easily from situations where user processes fill up file systems.

As we had earlier mentioned physical block allocation policy attempts to place logically related blocks physically close so that IO is expedited. This is achieved by having two forms of groups:

1. Block group
2. Cylinder group.

Usually the file allocation is attempted with the block group with the inode of the file in the same block group. Also within a block group physical proximity is attempted. As for the cylinder group, the distribution depends on the way head movement can be optimized.

1.8.1 Advanced Ext2fs features

In addition to the standard features of the *NIX file systems ext2fs supports several advanced features.

File attributes allow the users to modify the kernel behavior when acting on a set of files.

One can set attributes on a file or on a directory. In the later case, new files created in the directory inherit these attributes. (Examples: Compression Immutability etc) BSD or System V Release 4 semantics can be selected at mount time. A mount option allows the administrator to choose the file creation semantics. On a file-system mounted with BSD semantics, files are created with the same group id as their parent directory.

System V semantics are a bit more complex: if a directory has the setgid bit set, new files inherit the group id of the directory and subdirectories inherit the group id and the setgid bit; in the other case, files and subdirectories are created with the primary group id of the calling process.

BSD-like synchronous updates can be used in Ext2fs. A mount option allows the administrator to request that metadata (inodes, bitmap blocks, indirect blocks and directory blocks) be written synchronously on the disk when they are modified. This can be useful to maintain a strict metadata consistency but this leads to poor performances.

Ext2fs allows the administrator to choose the logical block size when creating the filesystem.

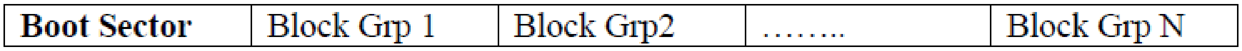
Block sizes can typically be 1024, 2048 and 4096 bytes.

Ext2fs implements fast symbolic links. A fast symbolic link does not use any data block on the file-system. The target name is not stored in a data block but in the inode itself. Ext2fs keeps track of the file-system state. A special field in the superblock is used by the kernel code to indicate the status of the file system. When a file-system is mounted in read or write mode, its state is set to ``Not Clean". Whenever filesystem is unmounted, or re-mounted in read-only mode, its state is reset to: ``Clean". At boot time, the file-system checker uses this information to decide if a file-system must be checked. The kernel code also records errors in this field. When an inconsistency is detected by the kernel code, the file-system is marked as ``Erroneous". The file-system checker tests this to force the check of the file-system regardless of its apparently clean state.

Always skipping filesystem checks may sometimes be dangerous, so Ext2fs provides two ways to force checks at regular intervals. A mount counter is maintained in the superblock. Each time the filesystem is mounted in read/write mode, this counter is incremented. When it reaches a maximal value (also recorded in the superblock), the filesystem checker forces the check even if the filesystem is ``Clean". A last check time and a maximal check interval are also maintained in the superblock. These two fields allow the administrator to request periodical checks. When the maximal check interval has been reached, the checker ignores the filesystem state and forces a filesystem check. Ext2fs offers tools to tune the filesystem behavior like tune2fs.

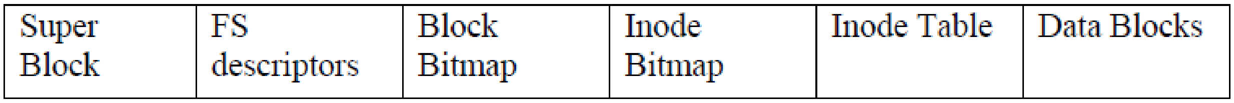
1.8.2 Physical Structure

The physical structure of Ext2 filesystems has been strongly influenced by the layout of the BSD filesystem .A filesystem is made up of block groups. The physical structure of a filesystem is represented in this table:



Boot Sector	Block Grp 1	Block Grp2	……..	Block Grp N

Each block group contains a redundant copy of crucial filesystem control informations (superblock and the filesystem descriptors) and also contains a part of the filesystem (a block bitmap, an inode bitmap, a piece of the inode table, and data blocks). The structure of a block group is represented in this table:



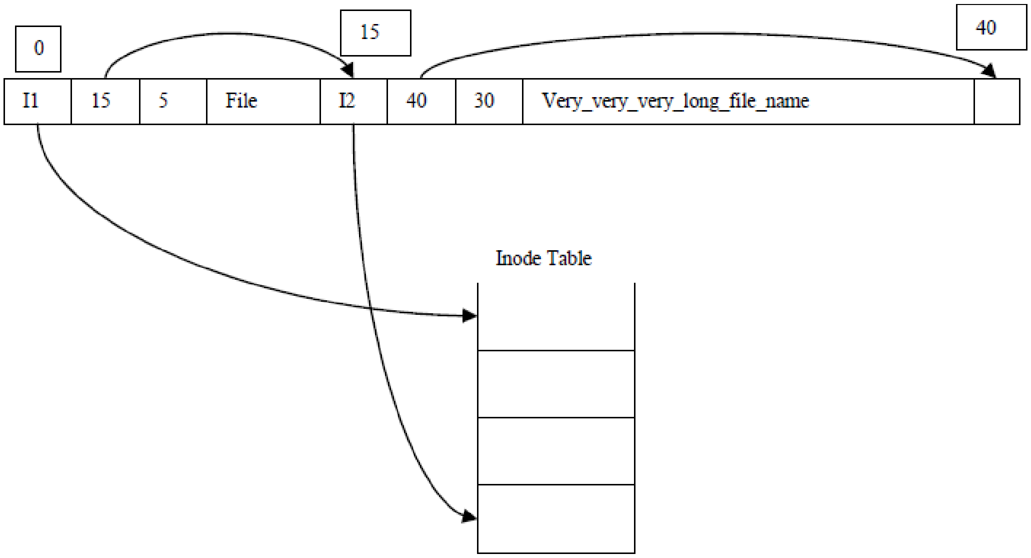
Super Block	FS descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

Using block groups is a big factor contributing to the reliability of the file system: since the control structures are replicated in each block group, it is easy to recover from a filesystem where the superblock has been corrupted. This structure also helps to get good performances: by reducing the distance between the inode table and the data blocks, it is possible to reduce the disk head seeks during I/O on files.

In Ext2fs, directories are managed as linked lists of variable length entries. Each entry contains the inode number, the entry length, the file name and its length. By using variable length entries, it is possible to implement long file names without wasting disk space in directories.

As an example, the next table represents the structure of a directory containing three files: File, Very_long_name, and F2. The first entry in the table is inode number; the second entry is the entire entry length: the third field indicates the length of the file name and the last entry is the name of the file itself

I1	15	05	File
I2	40	30	Very_very_very_long_file_name
I3	12	03	



1.8.3 The EXT3 file system

The ext2 file system is in fact a robust and well tested system. Even so some problem areas have been identified with ext2fs. These are mostly with the shutdown fsck (for filesystem health check at the time of shutdown). It takes unduly long to set it right using e2fsck . The solution was to add journaling to the filesystem. One more line about journaling. Another issue with the ext2 file system is its poor capability to scale to very large drives and files. The EXT3 file system which is in some sense an extension of the ext2 filesystem will try to address these shortcomings and also offer many other enhancements.

1.8.4 THE PROC FILE SYSTEM

Proc file system shows the power of the Linux virtual file system. The Proc file system is a special file system which actually displays the present state of the system. In fact we can call it a ‘pretend’ file system. If one explores the /proc directory one notices that all the files have zero bytes as the file size. Many commands like *ps* actually parse the /proc files to generate their output. Interestingly enough Linux does not have any system call to get process information. It can only be accessed by reading the proc file system. The proc file system has a wealth of information. For example the file /proc/cpuinfo gives a lot of things about the host processor.

A sample output could be as shown below:

```
processor : 0
vendor_id : AuthenticAMD
cpu family : 5
model : 9
model name : AMD-K6(tm) 3D+ Processor
stepping : 1
cpu MHz : 400.919
cache size : 256 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
```

```
fpu_exception : yes
cpuid level : 1
wp : yes
flags : fpu vme de pse tsc msr mce cx8 pge mmx syscall 3dnow k6_mtrr
bogomips : 799.53
```

/proc also contains, apart from other things, properties of all the processes running on the system at that moment. Each property is grouped together into a directory with a name equal to the PID of the process. Some of the information that can be obtained is shown as follows.

/proc/PID/cmdline

Command line arguments.

/proc/PID/cpu

Current and last cpu in which it was executed.

/proc/PID/cwd

Link to the current working directory.

/proc/PID/environ

Values of environment variables.

/proc/PID/exe

Link to the executable of this process.

/proc/PID/fd

Directory, which contains all file descriptors.

/proc/PID/maps

Memory maps to executables and library files.

/proc/PID/mem

Memory held by this process.

/proc/PID/root

Link to the root directory of this process.

/proc/PID/stat

Process status.

/proc/PID/statm

Process memory status information.

/proc/PID/status

Process status in human readable form

1.9 DEVICE DRIVERS ON LINUX

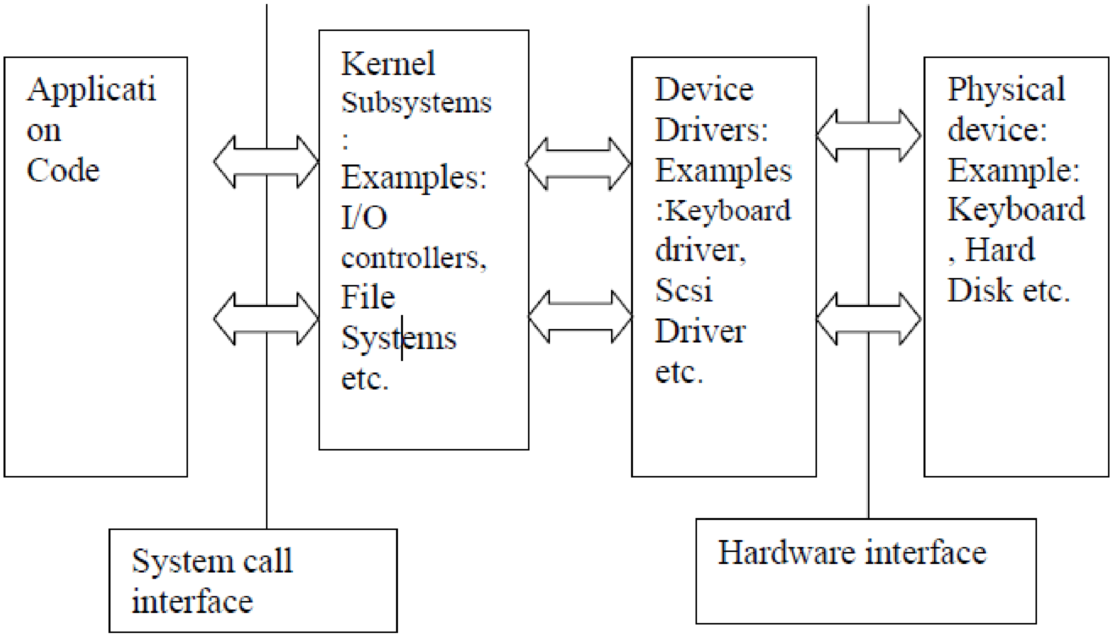
Introduction:

Most of the Linux code is independent of the hardware it runs on. Applications are often agnostic to the internals of a hardware device they interact with. They interact with the devices as a black box using operating system defined interfaces. As far as applications are concerned, inside the black box sits a program that exercises a protocol to interact with the device completely. This program interacts with the device at a very low level and abstracts away all the oddities and peculiarities of the underlying hardware to the invoking application. Obviously every device has a different device driver. The demand for device drivers is increasing as more and more devices are being introduced and the old ones become obsolete.

In the context of Linux as an open source OS, device drivers are in great demand. There are two principal drivers behind this. Firstly, many hardware manufacturers do not ship a Linux driver so it is left for someone from the open source community to implement a driver. Second reason is the large proliferation of Linux in the embedded system market.

Some believe that Linux today is number one choice for embedded system development work. Embedded devices have special devices attached to them that require specialized drivers. An example could be a microwave oven running Linux and having a special device driver to control its turntable motor.

In Linux the device driver can be linked into the kernel at compile time. This implies that the driver is now a part of the kernel and it is always loaded. The device driver can also be linked into the kernel dynamically at runtime as a pluggable module.


Application Code
Kernel Subsystems: Examples: I/O controllers, File Systems etc.
Device Drivers: Examples :Keyboard driver, Scsi Driver etc.
Physical device: Example: Keyboard , Hard Disk etc.
System call interface
Hardware interface

Almost every system call eventually maps to a physical device. With the exception of the processor, memory and a few other entities, all device control operations are performed by code that is specific to the device. This code as we know is called the device driver.

Kernel must have device drivers for all the peripherals that are present in the system right from the keyboard to the hard disk etc.

1.9.1 Device classes

Char devices:

These devices have a stream oriented nature where data is accessed as a stream of bytes example serial ports. The drivers that are written for these devices are usually called “char device drivers”. These devices are accessed using the normal file system. Usually they are mounted in the /dev directory. If ls –al command is typed on the command prompt in the /dev directory these devices appear with a ‘c’ in the first column.

Example:

```
crw-rw-rw-    1 root        tty     2, 176 Apr 11     2002 ptya0
crw-rw-rw-    1 root        tty     2, 177 Apr 11     2002 ptya1
crw-rw-rw-    1 root        tty     2, 178 Apr 11     2002 ptya2
crw-rw-rw-    1 root        tty     2, 179 Apr 11   2002 ptya3
```

1.9.2 Block devices

These devices have a ‘block’ oriented nature where data is provided by the devices in blocks. The drivers that are written for these devices are usually called as block device drivers. Classic example of a block device is the hard disk. These devices are accessed using the normal file system. Usually they are mounted in the /dev directory. If a ls –al command is typed on the command prompt in the /dev directory these devices appear with a ‘b’ in the first column.

Example:

```
brw-rw----    1 root    disk       29,    0 Apr 11 2002  aztcd
brw-rw----    1 root    disk       41,    0 Apr 11 2002  bpcd
brw-rw----    1 root    floppy      2,    0 Apr 11 2002  fd0
```

1.9.3 Network devices

These devices handle the network interface to the system. These devices are not accessed via the file system. Usually the kernel handles these devices by providing special names to the network interfaces e.g. eth0 etc.

Note that Linux permits a lot of experimentation with regards to checking out new device drivers. One need to learn to load, unload and recompile to check out the efficacy of any newly introduced device driver. The cycle of testing is beyond the scope of discussion here.

1.9.4 Major/minor numbers

Most devices are accessed through nodes in the file system. These nodes are called special files or device files or simply nodes of the file system tree. These names are usually mounted in the /dev/ directory.

If a ls –al command is issued in this directory we can see two comma separated numbers that appear where usually the file size is mentioned. The first number (from left side) is called the device major number and the second number is called the device minor number.

Example: `crw-rw-rw-   1 root   tty      2, 176 Apr 11 2002 ptya0`

Here the major number is 2 and the minor number is 176

The major number is used by the kernel to locate the device driver for that device. It is an index into a static array of the device driver entry points (function pointers). The minor number is passed to the driver as an argument and the kernel does not bother about it. The minor number may be used by the device driver to distinguish between the different types of devices of the same type it supports. It is left to the device driver, what it does with the minor numbers. For the Linux kernel 2.4 the major and minor numbers are eight bit quantities. So at a given time you can have utmost 256 drivers of a particular type and 256 different types of devices loaded in a system. This value is likely to increase in future releases of the kernel.

Kernel 2.4 has introduced a new (optional) file system to handle device. This file system is called the device file system . In this file system the management of devices is much more simplified. Although it has lot of user visible incompatibilities with the previous file system, at present device file system is not a standard part of most Linux distributions.

In future, things might change in favour of the device file system. Here it must be mentioned that the following discussion is far from complete. There is no substitute for looking at the actual source code. The following section will mainly help the reader to know what to *grep* for in the source code.

We will now discuss each of the device class drivers that is block, character and network drivers in more detail.

1.10 Character Drivers

Driver Registeration/Uregisteration:

We register a device driver with the Linux kernel by invoking a routine (<Linux/fs.h>) int register_chrdev(unsigned int major, const char * name, struct file_operations * fops); Here the major argument is the major number associated with the device. Name signifies the device driver as it will appear in the /proc/devices once it is successfully registered.

The fops is a pointer to the structure containing function pointers to the devices' functionalities. We will discuss fops in detail later.

Now the question arises: how do we assign a major number to our driver:

Assigning major numbers:

Some numbers are permanently allocated to some common devices. The reader may like to explore: /Documentation/devices.txt in the source tree. So if we are writing device drivers for these devices we simply use these major numbers.

If that is not the case then we can use major numbers that are allocated for experimental usage. Major numbers in the range 60-63, 120-127, 240-254 are for experimental usage.

But how do we know that a major number is not already used especially when we are shipping a driver to some other computer.

By far the best approach is to dynamically assign the major number. The idea is to get a free major number by looking at the present state of the system and then assigning it to our driver. If the register_chrdev function is invoked with a zero in the major number field, the function, if it registers the driver successfully, returns the major number allocated to it. What it does is that it searches the system for an unused major number, assigns it to the driver and then returns it. The story does not end here. To access our device we need to add our device to the file system tree. That is, we need to do *mknod* for the device into the tree. For that we need to know the major number for the driver. For a statically assigned major number that is not a problem. Just use that major number you assigned to the device. But for a dynamically assigned number how do we get the major number? The answer is: parse the /proc/devices file and find out the major number assigned to our device. A script can also be written to do the job.

Removing a driver from the system is easy. We invoke the unregister_chrdev(unsigned int major, const char * name);

1.11 Important Data Structure

The file Structure<linux/fs.h>:

Every Linux open file has a corresponding file structure associated with it. Whenever a method of the device driver is invoked the kernel will pass the associated file structure to the method. The method can then use the contents of this structure to do its job. We list down some important fields of this structure.

mode_t f_mode;

This field indicates the mode of the file i.e for read write or both etc.

loff_t f_pos;

The current offset in the file.

unsigned int f_flags;

This fields contains the flags for driver access for example synchronous access (blocking) or asynchronous (non blocking) access etc.

struct file_operations * fops;

This structure contains the entry points for the methods that device driver supports. This is an important structure we will look at it in more detail in the later sections.

void * private_data;

This pointer can be allocated memory by the device driver for its own personal use. Like
for maintaining states of the driver across different function calls.

sruct dentry * f_dentry;

The directory entry associated with the file.

Etc.

1.11.1 The file operations structure(fops):<linux/fs.h>

This is the most important structure as far as device driver writer are concerned. It contains pointers to the driver functions. The file structure discussed in the previous section contains a pointer to the fops structure. The file (device) is the object and fops contains the methods that act on this object. We can see here object oriented approach in the Linux Kernel.

Before we look at the members of the fops structure it will be useful if we look at taggd structure initialization:

Tagged structure initializations

The fops structure has been expanding with every kernel release. This can lead to compatibility problems of the driver across different kernel versions.

This problem is solved by using tagged structure initialization. Tagged structure initialization is an extension of ANSI C by GNU. It allows initialization of structure by name tags rather than positional initialization as in standard C.

Example:

```
struct fops myfops={
……………………..
………………….
open : myopen;
close : myclose:
…………..
…………
}
```

The intilization can now be oblivious of the change in the structure (Provided obviously that the fields have not been removed).

Pointers to functions that are implemented by the driver are stored in the fops structure.

Methods that are not implemented are made NULL.

Now we look at some of the members of the fops structure:

```
loff_t (*llseek) (struct file *,loff_t);
/* This method can be used to change the present offset in a file. */
ssize_t (*read) (struct file*,char *,size_t,loff_t *);
/* Read data from a device.*/
ssize_t(*write) (struct file *,const char *,size_t,loff_t *);
/* Write data to the device. */
int (* readdir) (struct file *,void *,fill_dir_t);
/* Reading directories. Useful for file systems.*/
unsigned int (* poll) (struct file *,struct poll_table_struct *);
/* Used to check the state of the device. */
int (*ioctl)(struct inode *,struct file *,unsigned int,unsigned long);
/* The ioctl is used to issue device specific calls(example setting the baud rate of the
serial port). */
int (*mmap) (struct file *,struct vm_area_struct *);
/* Map to primary memory */
int (* open) (struct inode *,struct file *);
/ * Open device.*/
int ( *flush) (struct file *) ;
/* flush the device*/
int (*release) (struct inode *,struct file *);
/* Release the file structure */
int(*fsync) (struct inode *,struct dentry *);
/* Flush any pending data to the device. */
```

Etc.

1.12 Advance Char Driver Operations

Although most of the following discussion is valid to character as well as network and block drivers, the actual implementation of these features is explained with respect to char drivers.

1.12.1 Blocking and non-blocking operations

Device drivers usually interact with hardware devices that are several orders of time slower than the processor. Typically if a modern PC processor takes a second to process a byte of data from a keyboard, the keyboard takes several thousand years to produce a single byte of data. It will be very foolish to keep the processor waiting for data to arrive from a hardware device. It could have severe impact on the overall system performance and throughput. Another cause that can lead to delays in accessing devices, which has nothing to do with the device characteristics, is the policy in accessing the device. There might be cases where device may be blocked by other drivers . For a device driver writer it is of paramount importance that the processor is freed to perform other tasks when the device is not ready.

We can achieve this by the following ways.

One way is blocking or the synchronous driver access. In this way of access we cause the invoking process to sleep till the data arrives. The CPU is then available for other processes in the system. The process is then awakened when the device is ready.

Another method is in which the driver returns immediately whether the device is ready or not allowing the application to poll the device.

Also the driver can be provided asynchronous methods for indicating to the application when the data is available.

Let us briefly look at the Linux kernel 2.4 mechanisms to achieve this.

There is a flag called O_NONBLOCK flag in filp->f_flags (<linux/fcntl.h>).

If this flag is set it implies that the driver is being used with non-blocking access. This flag is cleared by default. This flag is examined by the driver to implement the correct semantics.

Blocking IO:

There are several ways to cause a process to sleep in Linux 2.4. All of them will use the same basic data structure, the wait queue (wait_queue_head_t). This queue maintains a linked list of processes that are waiting for an event.

A wait queue is declared and initialized as follows:

```
wait_queue_head_t my_queue; /* declaration */
init_waitqueue_head(&my_queue) /* initialization */
/* 2.4 kernel requires you to intialize the wait queue, although some earlier versions of
the kernel did not */
```

The process can be made to sleep by calling any of the following:

```
sleep_on(wait_queue_head_t * queue);
/* Puts the process to sleep on this queue. */
/* This routine puts the process into non-interruptible sleep */
/* this a dangerous sleep since the process may end up sleeping forever */
interruptible_sleep_on(wait_queue_head_t * queue)
/* same as sleep_on with the exception that the process can be awoken by a signal */
sleep_on_timeout(wait_queue_head_t * queue,long timeout)
/* same as sleep_on except that the process will be awakened when a timeout happens.
The timeout parameter is measured in jiffies */
interruptible_sleep_on_timeout(wait_queue_head_t * queue,long timeout)
/* same as interruptible_sleep_on except that the process will be awakened when a
timeout happens. The timeout parameter is measured in jiffies */
void wait_event(wait_queue_head_t * queue,int condition)
int wait_event_interruptible(wait_queue_head_t * queue, int condition)
/* sleep until the condition evaluates to true that is non-zero value */
/* preferred way to sleep */
```

If a driver puts a process to sleep there is usually some other part of the driver that awakens it, typically it is the interrupt service routine.

One more important point is that if a process is in interruptible sleep it might wake up even on a signal even if the event it was waiting on, has not occurred. The driver must in this case put a process in sleep in a loop checking for the event as a condition in the loop.

The kernel routines that are available to wake up a process are as follows:

```
wake_up(wait_queue_head_t * queue)
/* Wake proccess in the queue */
wake_up_interruptible(wait_queue_head_t * queue)
/* wake process in the queue that are sleeping on interruptible sleep in the queue rest of
```

the procccess are left undisturbed */

wake_up_sync(wait_queue_head_t_ * queue)

wake_up_interruptible_sync(wait_queue_head_t_ * queue)

/* The normal wake up calls can cause an immediate reschedule of the processor */

/* these calls will only cause the process to go into runnable state without rescheduling
the CPU */

Non Blocking IO:

If O_NONBLOCK flag is set then driver does not block even if data is not available for the call to complete. The normal semantics for a non-blocking IO is to return –EAGAIN which really tells the invoking application to try again. Usually devices that are using non-blocking access to devices will use the poll system call to find out if the device is ready with the data. This is also very useful for an application that is accessing multiple devices without blocking.

Polling methods: Linux provides the applications 'poll' and 'select' system calls to check if the device is ready without blocking. (There are two system calls offering the same functionality for historical reasons. These calls were implemented in UNIX at nearly same time by two different distributions: BSD Unix (select) System 5(poll))

Both the calls have the following prototype:

unsigned int (*poll)(struct file * ,poll_table *);

The poll method returns a bit mask describing what operations can be performed on the device without blocking.

1.13 Asynchronous Notification

Linux provides a mechanism by which a drive can asynchronously notify the application if data arrives. Basically a driver can signal a process when the data arrives. User processes have to execute two steps to enable asynchronous notification from a device.

1. The process invokes the F_SETOWN command using the fcntl system call, the process ID of the process is saved in filp->f_owner. This is the step needed basically for the kernel to route the signal to the correct process.
2. The asynchronous notification is then enabled by setting the FASYNC flag in the device by means of F_SETFEL fcntl command.

   After these two steps have been successfully executed the user process can request the delivery of a SIGIO signal whenever data arrives.

1.14 Interrupt Handling in LINUX 2.4

The Linux kernel has a single entry point for all the interrupts. The number of interrupt lines is platform dependent. The earlier X86 processors had just 16 interrupt lines. But
now this is no longer true. The current processors have much more than that. Moreover new hardware comes with programmable interrupt controllers that can be programmed among other things to distribute interrupts in an intelligent and a programmable way to different processors for a multi-processors system. Fortunately the device driver writer does not have to bother too much about the underlying hardware, since the Linux kernel nicely abstracts it. For the Intel x86 architecture the Linux kernel still uses only 16 lines.

The Linux kernel handles all the interrupts in the same manner. On the receipt of an interrupt the kernel first acknowledges the interrupt. Then it looks for registered handlers for that interrupt. If a handler is registered, it is invoked. The device driver has to register a handler for the interrupts caused by the device.

The following API is used to register an interrupt handler.

```
<linux/sched.h>
int request_irq(unsigned int irq, void ( * interruptHandler ) (int, void *,struct pt_regs *),
unisgned long flags, const char * dev_name, void * dev_id);
/* irq -> The interrupt number being requested */
/* interruptHandler -> function pointer to the interrupt handler */
/* flags -> bitwise orable flags one of
SA_INTERRUPT implies 'fast handler' which basically means that the interrupt handler finishes
its job quickly and can be run in the interrupt context with interrupts disabled. SA_SHIRQ
implies that the interrupt is shared
SA_SAMPLE_RANDOM implies that the interrupt can be used to increase the entropy of the
system */
/* dev_name ->A pointer to a string which will appear in /proc/interrupts to signify the owner of
the interrupt */
/* dev_id-> A unique identifier signifying which device is interrupting. Is mostly used
```

when the interrupt line is shared. Otherwise kept NULL*/
/* the interrupt can be freeed implying that the handler associated with it can be removed
*/void free_irq(unsigned int irq,void * dev_id);
/* by calling the following function. Here the meaning of the parameters is the same as in request_irq
*/Now the question that arises: how do we know which interrupt line our device is going to use. Some device use predefined fixed interrupt lines. So they can be used. Some devices have jumper settings on them that let you decide which interrupt line the device will use. There are devices (like device complying to the PCI standard) that can on request tell which interrupt line they are going to use. But there are devices for which we cannot tell before hand which interrupt number they are going to use. For such device we need the driver to probe the IRQ number. Basically what is done is the device is asked to interrupt and then we look at all the free interrupt lines to figure out which line got interrupted. This is not a clean method and ideally a device should itself announce which interrupt it wants to use. (Like PCI).

The kernel provides helper functions for probing of interrupts(<linux/interrupt.h> probe_irq_on , probe_irq_off) or the drive can do manual probing for interrupts.

1.15 Top Half And Bottom Half Processing

One problem with interrupt processing is that some interrupt service routines are rather long and take a long time to process. These can then cause interrupts to be disabled for a long time degrading system responsiveness and performance. The method used in Linux (and in many other systems) to solve this problem is to split up the interrupt handler into two parts : The “top half” and the “bottom half”. The top half is what is actually invoked at the interrupt context. It will just do the minimum required processing and then wake up the bottom half. The top half is kept very short and fast. The bottom half then does the time consuming processing at a safer time.

Earlier Linux had a predefined fixed number of bottom halves (32 of them) for use by the driver. But now the (Kernel 2.3 and later) the kernel uses “tasklets” to do the bottom half processing. Tasklet is a special function that may be scheduled to run in interrupt context, at a system determined safe time. A tasklet may be scheduled to run multiple times, but it only runs once. An interesting consequence of this is that a top half may be executed several times before a bottom half gets a chance to execute. Now since only a single tasklet will be run, the tasklet should be able to handle such a situation. The top half should keep a count of the number of interrupts that have happened. The tasklet can use this count to figure out what to do.

```
/* Takelets are declared using the following macro */
DECLARE_TASKLET(taskLetName,Function,Data);
/* taskLetName -> Name of the tasklet */
/* the function to be run as a tasklet. The function has the following prototype */
/* void Function(usigned long ) */
/* Data is the argument to be passed to the function */
/* the tasklet can be scheduled using this function */
tasklet_schedule(&takletName)
```

Interprocess Communication in Linux:

Again there is considerable similarity with Unix. For example, in Linux, *signals* may be utilized for communication between parent and child processes. Processes may synchronize using *wait* instruction. Processes may communicate using *pipe* mechanism. Processes may use *shared memory mechanism* for communication.

Probably need some more points on this topic on IPC and the different mechanisms available. I found a good url “http://cne.gmu.edu/modules/ipc/map.html”.

(Show these using animation)

Let us examine how the communication is done in the networked environment. The networking features in Linux are implemented in three layers:

1. Socket interface
2. Protocol drivers
3. Network drivers.

Typically a user applications’ first I/F is the socket. The socket definition is similar to BSD 4.3 Unix which provides a general purpose interconnection framework. The protocol layer supports what is often referred to as protocol stack. The data may come from either an application or from a network driver. The protocol layer manages routing, error reporting, reliable retransmission of data for networking the most important support is the IP suite which guides in routing of packets between hosts. On top of the routing are built higher layers like UDP or TCP.

The routing is actually done by IP driver. The IP driver also helps in disassembly / assembly of the packets. The routing gets done in two ways:

1. By using recent cached routing decisions
2. By using a table which acts as a persistent forwarding base

Generally the packets are stored in a buffer and have a tag to identify the protocol that need to be used. After the selection of the appropriate protocol the IP driver then hands it over to the network device driver to manage the packet movement.

As for security, the firewall management maintains several chains – with each chain having its own set of rules of filtering the packets.

Real Time Linux:

Large number of projects both open source and commercial have been dedicated to get real time functionality from the Linux kernel. Some of the projects are listed below

Commercial distributions:

FSMLabs: RTLinuxPro

Lineo Solutions: uLinux

LynuxWorks: BlueCat RT

MontaVista Software: Real-Time Solutions for Linux

Concurrent: RedHawk

REDSonic: REDICE-Linux

Open source distributions:

ADEOS –

ART Linux

KURT -- The KU Real-Time Linux

Linux/RK

QLinuxRealTimeLinux.org

RED-Linux

RTAI

RTLinux

1.16 Linux Installation

Amongst various flavors of UNIX, Linux is currently the most popular OS. Linux is also part of the GNU movement which believes in free software distribution. A large community of programmers subscribe to it. Linux came about mainly through the efforts of Linus Torvalds from Finland who wanted a UNIX environment on his PC while he was a university student. He drew inspiration from Prof. Andrew Tanenbaum of University of Amsterdam, who had earlier designed a small OS called Minix. Minix was primarily used as a teaching tool with its code made widely available and distributed. Minix code could be modified and its capability extended. Linus Torvalds not only designed a PC-based Unix for his personal use, but also freely distributed it. Presently, there is a very large Linux community worldwide. Every major university, or urban centre, has a Linux group. Linux found ready acceptance and the spirit of free distribution has attracted many willing voluntary contributors. Now a days Linux community regulates itself by having all contributions evaluated to ensure quality and to take care of compatibility. This helps in ensuring a certain level of acceptance. If you do a Google search you will get a lot of information on Linux. Our immediate concerns here are to help you have your own Linux installation so that you can practice with many of the tools available under the broad category of Unix-based OSs.

1.16.1 The Installation

Linux can be installed on a wide range of machines. The range may span from one's own PDA to a set of machines which cooperate like Google's 4000 node Linux cluster. For now we shall assume that we wish to install it on a PC. Most PCs have a bootable CD player and BIOS. This means in most cases we can use the CD boot and install procedure. Older PC's did not have these features. In that case one was required to use a set of floppies. The first part of this basic guide is about getting the installation program up and running: using either a CD or a set of floppies.

1.16.2 The Installation Program

In this section we describe the Linux installation. The main point in the installation is to select the correct configuration.

Typically Red Hat Linux is installed by booting to the install directory from a CD-ROM. The other options may include the following.

* Booting to install using a floppy disk.

* Using a hard drive partition to hold the installation software.

* Booting from a DOS Command line.

* Booting to an install and installing software using FTP or HTTP protocols.

* Booting to an install and installing software from an NFS-mounted hard drive.

Installing from CD-ROM : Most PCs support booting directly from a CD-ROM drive. Set your PC's BIOS (if required). Now insert the CD-ROM and reboot to the PC to install Red Hat Linux. You should see a boot screen that offers a variety of options for booting .The options typically would be as follows:

* <ENTER> - Start the installation using a Graphical interface

* text - Start the install using a text interface

* nofb - Start the install using a video frame buffer

* expert

* Linux rescue

* Linux dd

At this stage if you press key F2 then it provides a help screen for the text-based installation. Type the word text at the boot prompt and press Enter to continue. You shall be asked to select a language. So select a language of your choice. Highlight OK button and press Enter. You will then be asked to select a keyboard for install. So highlight OK Button and press Enter after selecting a keyboard. You shall be next asked to select a pointing device, select a suitable mouse and press OK.

Next you will be asked: Select the type of installation from?

* Workstation

* Server

* Laptop

* Custom

* Upgrade an existing system

Select the suitable option, for example, select server install and press Enter. Next you will choose a partitioning scheme. The choices include the following:

* Auto Partition

* Disk Druid

* Fdisk

The *Auto Partition* will the format hard drive according to the type of selected installation. It will automatically configure the partitions for use with Linux. The Disk Druid will launch a graphical editor listing the free spaces available. The *Fdisk* option offers an ability to create nearly 60 different types of partitions.

On clicking Disk Druid, you will get an option of creating new partitions if you are using a new hard drive. If you are using an old hard disk the partitions are recognized. Create the appropriate partitions or use existing ones as the case may be. Finally, press OK to continue.

Red Hat Linux requires a minimum of two partitions. One is a swap partition and the other a root(/) partition. The swap partition should be more than twice as large as the installed amount of memory. Other partitions may be */remote* and */home.* These can be created after the installation as well.

You will now be as asked to select a boot-loader for booting Linux. The choice of not using a boot-loader is also available. The options available are GRUB and LILO. Select the appropriate boot loader and press OK. Grub and Lilo are typically installed in the MBR of the first IDE hard drive in the PC. You will now be asked for to choose kernel parameters for booting Linux. Enter the arguments in the dialog box or use the OK Button to continue.

If for some reason we cannot arrive at dual booting automatically, then add this code at
the end of the file */etc/boot/grud/grub.conf* file

```
title Windows
        rootnoverify(hd0,0)
        chainloader +1
        makeactive
```

You can now configure a dual boot system, if required by configuring the boot-loader. When finished click OK and you will be asked to select a firewall configuration. Use a security level from

    * High
    * Medium
    * None

After this you will have to set the incoming service requests followed by a time-zone selection dialog box. Select the appropriate settings and press OK to continue.

You will now be prompted to enter a user-id and password. The password will not be echoed onto the screen. Now is the time to create user accounts. Each has home directory home usually under */home/usr* directory.

Next you have to select packages you want to install. Use the spacebar to select the various groups of software packages. The size of the installed software will dynamically reflect the choices. Use the select individual package item to choose the individual software packages. The installer will now start installing the packages selected from the CD-ROM drive onto the new Linux partitions.

At the end of the installation you will get an option of creating a boot-disk for later use. You can create the boot disk later using the *mkbootdisk* command.

After this, your installation is done. Press OK and Red Hat Linux will eject the CD ROM and reboot. After rebooting you will be able to log onto a Linux session. To shutdown your computer use the *shutdown -h now* command.

Usually most distributions allow you to Test the set-up. It helps to see if it works. The auto detection (like in Red Hat) takes care of most of the cards and monitor types.

1.16.3 Finishing the installation

With the above steps, we should have installed a good working Linux machine. The install program will usually prompt to take out all boot-disks, etc. and the machine will be rebooted (sometimes you may have to reboot). You will see the Linux loader coming up. It is also known as LILO. Newer versions or distributions like Mandrake come up with their own LILO's. RedHat 7.X comes with a graphical screen and menu for startup.

Anyway, one may see options like Linux and/or DOS or Windows . Normally we fill in these names during the installations. Another popular boot-loader called GRUB has become the default for RedHat.