Module 3: Processes and Process management

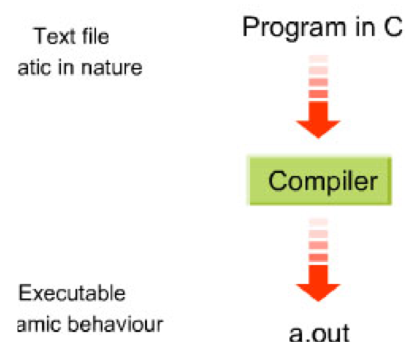
Recall from Module 1 that a process is a ***program in execution***. In this module we shall explain how a process comes into existence and how processes are managed.

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.

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 it self 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.

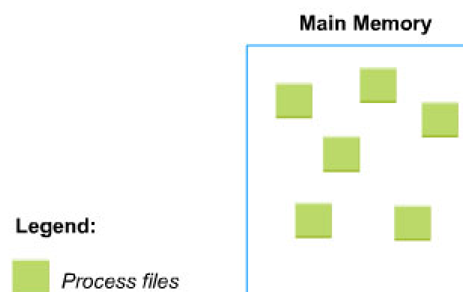
For instance, an output may have to appear on a monitor or a printed output may be needed. In other words, process management requires not only making the processor available for execution but, in addition, allocate main memory, files and IO. The process management component then requires coordination 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.

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.

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. We shall study the memory management in module no. 4.



Main memory management

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. We shall study IO management in module no. 5.

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).

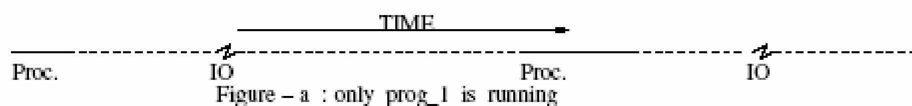


Figure 3.1(a): Multiple-program Processing

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 under utilization of the processor.

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

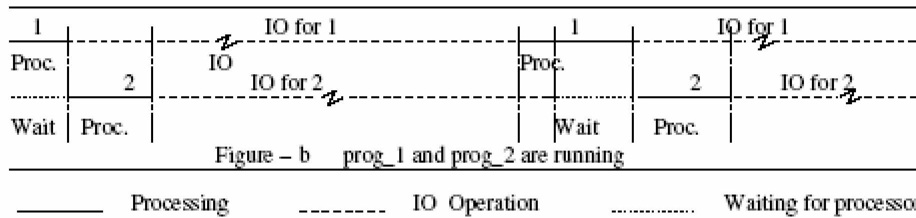
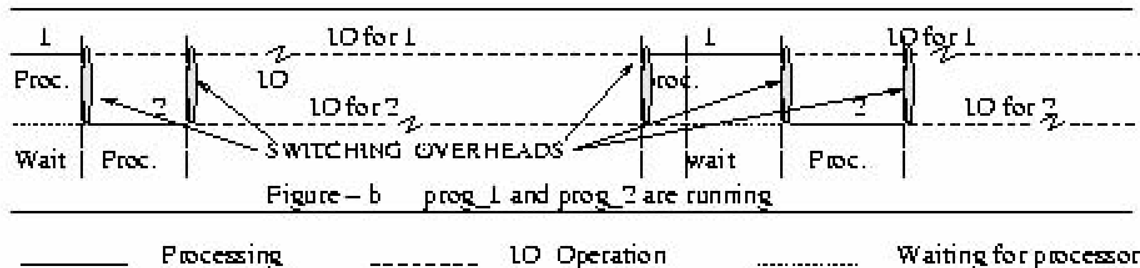


Figure 3.1(b): Multiple-program Processing

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.

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.



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.

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 over head. 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 some where 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 over all computer system.

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 describe 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.

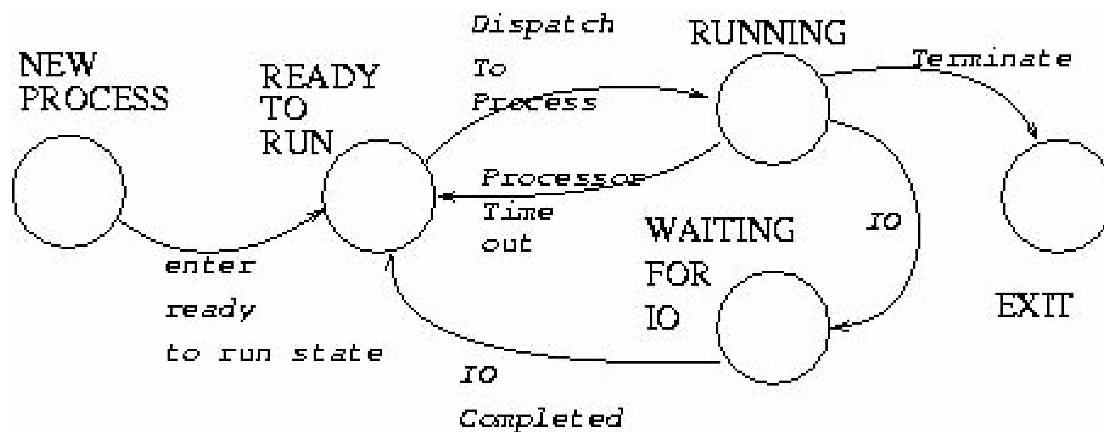


Figure 3.3: Modeling Process States

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. Some times 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.

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.

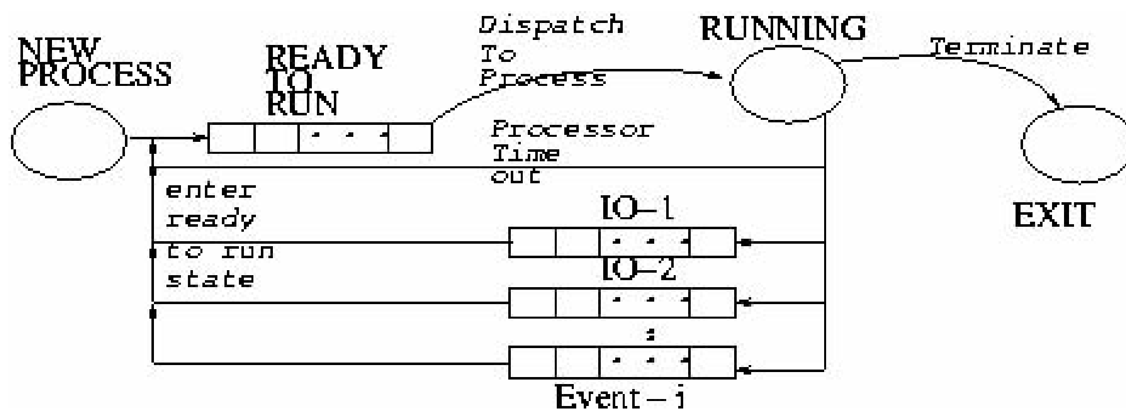


Figure 3.4: Queues-based Model

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.

Scheduling: A Few Scenarios

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 a ``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 some times). 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 out come 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 are 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.

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 dead-line 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. Other policy issues shall be studied later.

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 turn around 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 users 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.

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.

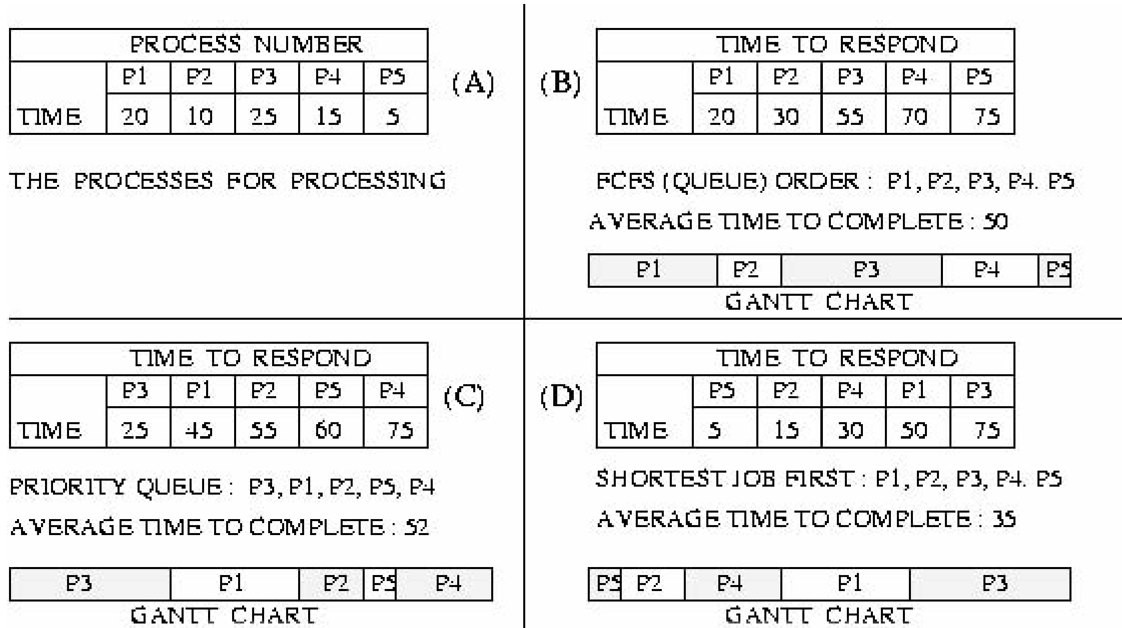


Figure 3.5: Comparison of three non-preemptive scheduling policies

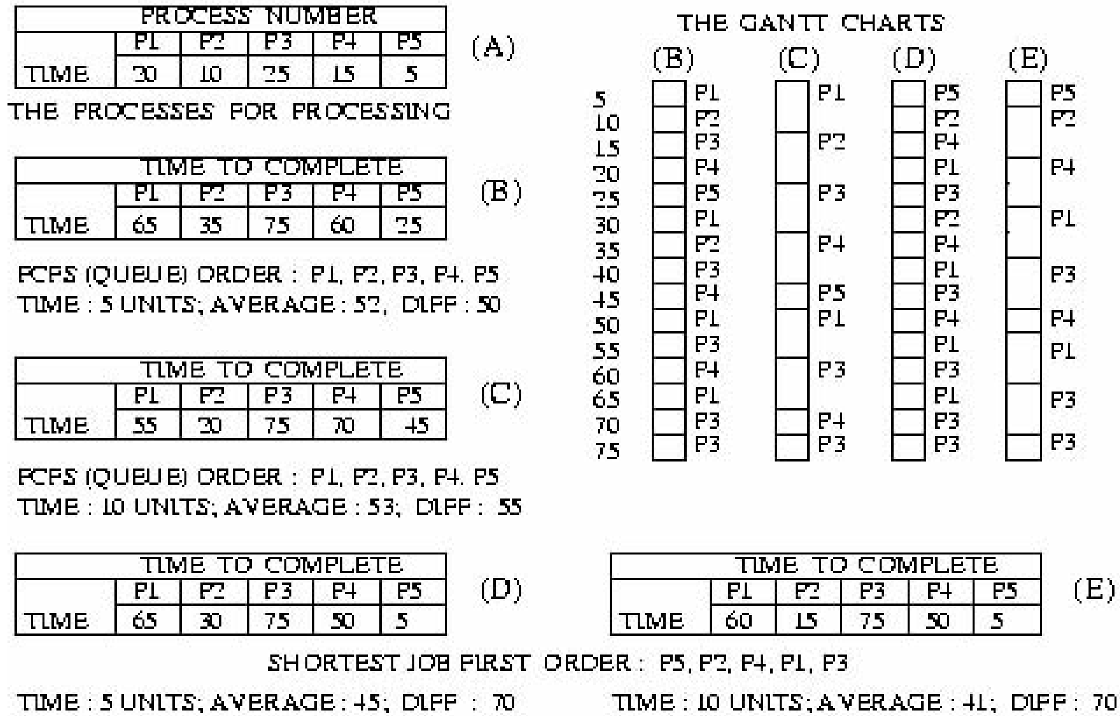
We assume non-preemptive operations for comparison of all the cases. We show the processing of jobs on a Gantt chart. Let us first assume processing in the FCFS or internal queue order i.e. p1, p2, p3, p4 and p5 (see 3.5(B)). Next we assume that jobs are arranged in a priority queue order (see3.5(C)). 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.

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)



(A) THE PROCESSES FOR PROCESSING

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

(B)

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

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

(C)

	TIME TO COMPLETE				
	P1	P2	P3	P4	P5
TIME	55	20	75	70	45

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

(D)

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

(E)

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

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

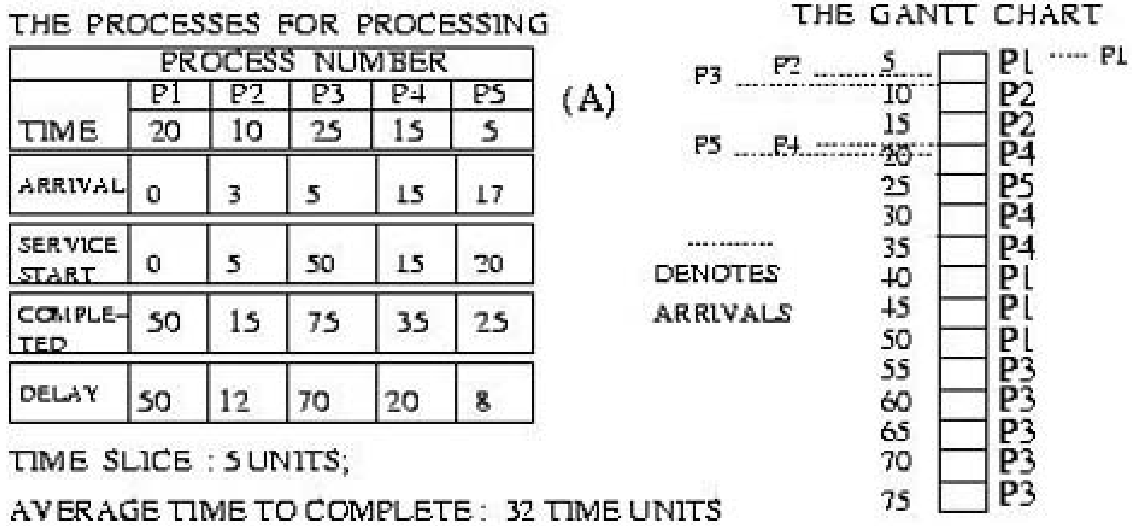
One of the interesting exercises is to find a good value for time slice for processor time allocation. OS designers spend a lot of time finding a good value for time slice.

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

Figure 3.7: Shortest Remaining Time Schedule

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.

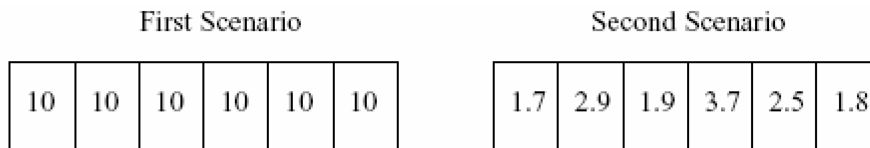


Figure 3.9: Processor time utilisation.

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.

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 $0 \le \alpha \le 1$ 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 worth while 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:

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

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

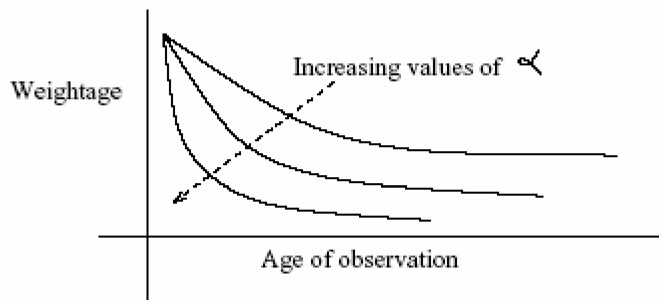


Figure 3.10: Processor time utilisation.

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

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.

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:

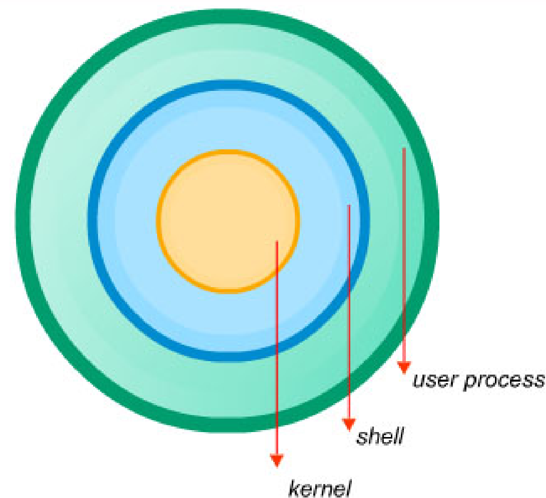
1. Program counter
2. Registers (like stack, index etc.) currently in use
3. Changed state (changed from Running to ready-to-run)
4. The base and limit register values
5. IO status (files opened; IO blocked or completed etc.)
6. Accounting
7. Scheduling information
8. Any other relevant information.

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

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.

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).

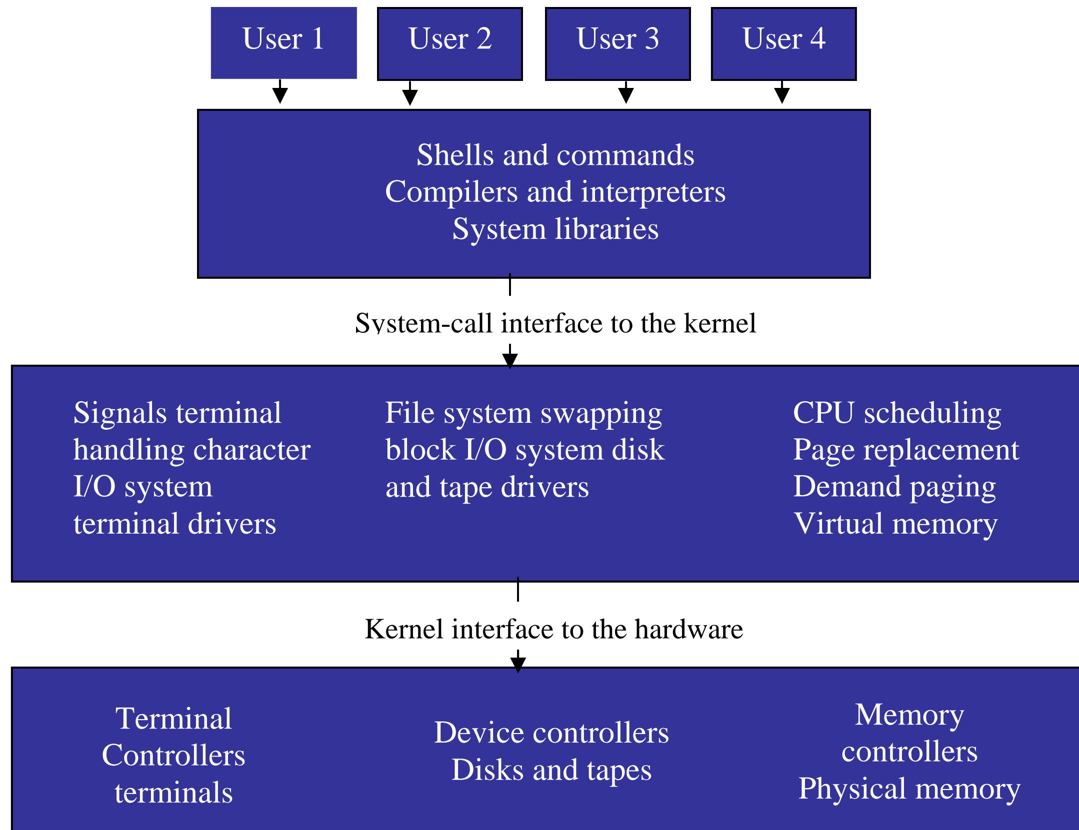
1. To create or terminate processes.
2. To access or allocate memory.
3. To get or set process attributes.
4. To create, open, read, write files.
5. To change access rights on files.
6. To mount or un-mount devices in a file system.

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.

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.

ardware



Unix Viewed as a Layered OS

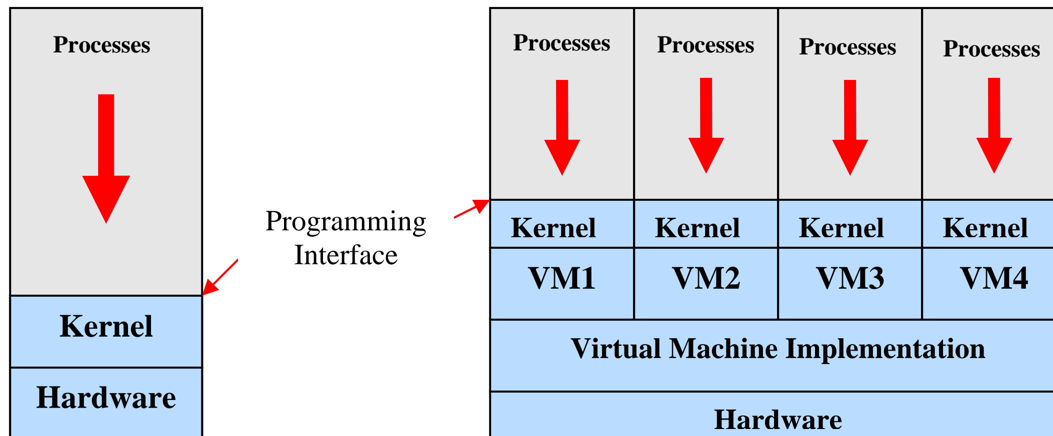
The Virtual Machine Concept:

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.

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.

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.

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.

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

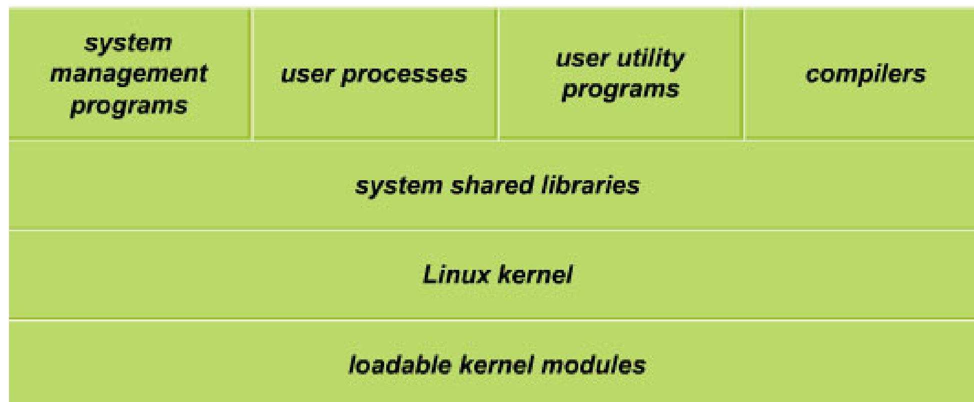
often cited and a very heavily used environment using Linux clusters is the famous Google search engine. Google uses geographically distributed clusters, each having any where up to 1000 Linux machines.

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.



We shall study more about Linux in module 19.