UNIT 2 PROCESSES

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier unit we have studied the overview and the functions of an operating system. In this unit we will have detailed discussion on the processes and their management by the operating system. The other resource management features of operating systems will be discussed in the subsequent units.

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when it is created, some initialisation data (input) may be passed along. For example, a process whose function is to display the status of a file, say F1, on the screen, will get the name of the file F1 as an input and execute the appropriate program to obtain the desired information.

We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program; they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes;
- The suspension is resumption of processes;
- The provision of mechanisms for process synchronization, and
- The provision of mechanisms for deadlock handling.

We will learn the operating system view of the processes, types of schedulers, different types of scheduling algorithms, in the subsequent sections of this unit.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the concepts of process, various states in the process and their scheduling;
- define a process control block;
- classify different types of schedulers;
- understand various types of scheduling algorithms, and
- compare the performance evaluation of the scheduling algorithms.

2.2 THE CONCEPT OF PROCESS

The term "process" was first used by the operating system designers of the MULTICS system way back in 1960s. There are different definitions to explain the concept of process. Some of these are, a process is:

- An instance of a program in execution
- An asynchronous activity
- The "animated spirit" of a procedure
- The "locus of control" of a procedure in execution
- The "dispatchable" unit
- Unit of work individually schedulable by an operating system.

Formally, we can define a **process** is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the process is the activity.

In multiprogramming systems, processes are performed in a pseudo-parallelism as if each process has its own processor. In fact, there is only one processor but it switches back and forth from process to process. Henceforth, by saying *execution* of a process, we mean the processor's operations on the process like changing its variables, etc. and *I/O work* means the interaction of the process with the I/O operations like reading something or writing to somewhere. They may also be named as "*processor (CPU) burst"* and *"I/O burst"* respectively. According to these definitions, we classify programs as:

- **Processor- bound program**: A program having long processor bursts (execution instants)
- **I/O- bound program**: A program having short processor bursts.

The operating system works as the computer system software that assists hardware in performing process management functions. Operating system keeps track of all the active processes and allocates system resources to them according to policies devised to meet design performance objectives. To meet process requirements OS must maintain many data structures efficiently. The process abstraction is fundamental means for the OS to manage concurrent program execution. OS must interleave the execution of a number of processes to maximize processor use while providing reasonable response time. It must allocate resources to processes in conformance with a specific policy. In general, a process will need certain resources such as CPU time, memory, files, I/O devices etc. to accomplish its tasks. These resources are allocated to the process when it is created. A single processor may be shared among several processes with some scheduling algorithm being used to determine when to stop work on one process and provide service to a different one which we will discuss later in this unit.

Operating systems must provide some way to create all the processes needed. In simple systems, it may be possible to have all the processes that will ever be needed be present when the system comes up. In almost all systems however, some way is needed to create and destroy processes as needed during operations. In UNIX, for instant, processes are created by the *fork* system call, which makes an identical copy of the calling process. In other systems, system calls exist to create a process, load its memory, and start it running. In general, processes need a way to create other processes. Each process has one parent process, but zero, one, two, or more children processes.

For an OS, the process management functions include:

- Process creation
- Termination of the process
- Controlling the progress of the process
- Process Scheduling
- Dispatching
- Interrupt handling / Exceptional handling
- Switching between the processes
- Process synchronization
- Interprocess communication support
- Management of Process Control Blocks.

2.2.1 Implicit and Explicit Tasking

A separate process at run time can be either–

- Implicit tasking and
- Explicit tasking.

Implicit tasking means that processes are defined by the system. It is commonly encountered in general purpose multiprogramming systems. In this approach, each program submitted for execution is treated by the operating system as an independent process. Processes created in this manner are usually transient in the sense that they are destroyed and disposed of by the system after each run.

Explicit tasking means that programs explicitly define each process and some of its attributes. To improve the performance, a single logical application is divided into various related processes. Explicit tasking is used in situations where high performance in desired system programs such as parts of the operating system and real time applications are common examples of programs defined processes. After dividing process into several independent processes, a system programs defines the confines of each individual process. A parent process is then commonly added to create the environment for and to control execution of individual processes.

Common reasons for and uses of explicit tasking include:

- **Speedup:** Explicit tasking can result in faster execution of applications.
- **Driving I/O devices that have latency:** While one task is waiting for I/O to complete, another portion of the application can make progress towards completion if it contains other tasks that can do useful work in the interim.
- **User convenience:** By creating tasks to handle individual actions, a graphical interface can allow users to launch several operations concurrently by clicking on action icons before completion of previous commands.
- **Multiprocessing and multicomputing:** A program coded as a collection of tasks can be relatively easily posted to a multiprocessor system, where individual tasks may be executed on different processors in parallel.

Distributed computing network server can handle multiple concurrent client sessions by dedicating an individual task to each active client session.

2.2.2 Processes Relationship

In the concurrent environment basically processes have two relationships, ***competition*** *and* ***cooperation***. In the concurrent environment, processes compete with each other for allocation of system resources to execute their instructions. In addition, a collection of related processes that collectively represent a single logical application cooperate with each other. There should be a proper operating system to support these relations. In the competition, there should be proper resource allocation and protection in address generation.

We distinguish between *independent process* and *cooperating process*. A process is independent if it cannot affect or be affected by other processes executing in the system.

Independent process: These type of processes have following features:

- Their state is not shared in any way by any other process.
- Their execution is deterministic, i.e., the results of execution depend only on the input values.
- Their execution is reproducible, i.e., the results of execution will always be the same for the same input.
- Their execution can be stopped and restarted without any negative effect.

Cooperating process: In contrast to independent processes, cooperating processes can affect or be affected by other processes executing the system. They are characterised by:

- Their states are shared by other processes.
- Their execution is not deterministic, i.e., the results of execution depend on relative execution sequence and cannot be predicted in advance.

Their execution is irreproducible, i.e., the results of execution are not always the same for the same input.

2.2.3 Process States

As defined, a process is an independent entity with its own input values, output values, and internal state. A process often needs to interact with other processes. One process may generate some outputs that other process uses as input. For example, in the shell command

cat file1 file2 file3 | grep tree

The first process, running *cat*, concatenates and outputs three files. Depending on the relative speed of the two processes, it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is ready and able to run to be blocked because the operating system is decided to allocate the CPU to other process for a while.

A process state may be in one of the following:

- **New :** The process is being created.
- **Ready :** The process is waiting to be assigned to a processor.
- **Running :** Instructions are being executed.
- **Waiting/Suspended/Blocked :** The process is waiting for some event to occur.
- **Halted/Terminated :** The process has finished execution.

The transition of the process states are shown in *Figure 1.*and their corresponding transition are described below:

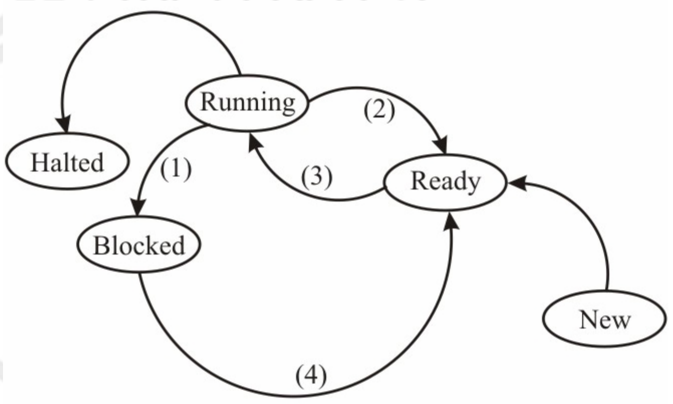


Figure 1: Typical process states

As shown in *Figure 1*, four transitions are possible among the states.
Transition 1 appears when a process discovers that it cannot continue. In order to get into blocked state, some systems must execute a system call *block*. In other systems, when a process reads from a pipe or special file and there is no input available, the process is automatically blocked.

Transition 2 and 3 are caused by the process scheduler, a part of the operating system. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all other processes have had their share and it is time for the first process to run again.

Transition 4 appears when the external event for which a process was waiting was happened. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes easier to think about what is going on inside the system. There are many processes like user processes, disk processes, terminal processes, and so on, which may be blocked when they are waiting for some thing to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is ready to run again.

The process model, an integral part of an operating system, can be summarized as follows. The lowest level of the operating system is the scheduler with a number of processes on top of it. All the process handling, such as starting and stopping processes are done by the scheduler. More on the schedulers can be studied is the subsequent sections.

2.2.4 Implementation of Processes

To implement the process model, the operating system maintains a table, an array of structures, called the *process table or process control block (PCB) or Switch frame.* Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information. In other words, it must contain everything about the process that must be saved when the process is switched from the running state to the ready state so that it can be restarted later as if it had never been stopped. The following is the information stored in a PCB.

- Process state, which may be new, ready, running, waiting or halted;
- Process number, each process is identified by its process number, called process ID;
- Program counter, which indicates the address of the next instruction to be executed for this process;
- CPU registers, which vary in number and type, depending on the concrete microprocessor architecture;
- Memory management information, which include base and bounds registers or page table;
- I/O status information, composed I/O requests, I/O devices allocated to this process, a list of open files and so on;
- Processor scheduling information, which includes process priority, pointers to scheduling queues and any other scheduling parameters;
- List of open files.

A process structure block is shown in *Figure 2*.

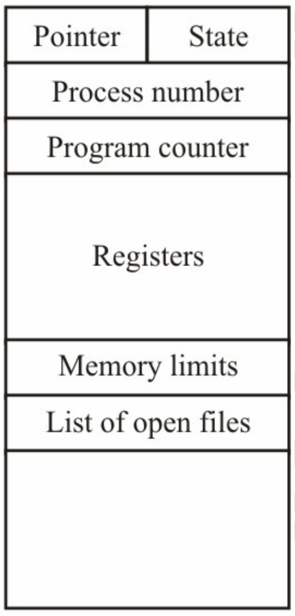


Figure 2: Process Control Block Structure

Context Switch

A *context switch* (also sometimes referred to as a *process switch* or a *task switch*) is the switching of the CPU (central processing unit) from one process or to another. A *context* is the contents of a CPU's registers and *program counter* at any point in time.

A context switch is sometimes described as the kernel suspending *execution of one process* on the CPU and resuming *execution of some other process* that had previously been suspended.

Context Switch: Steps

In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue normally.

The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block (PCB). Now, in order to switch processes, the PCB for the first process must be created and saved. The PCBs are sometimes stored upon a per-process stack in the kernel memory, or there may be some specific operating system defined data structure for this information.

Let us understand with the help of an example. Suppose if two processes A and B are in ready queue. If CPU is executing Process A and Process B is in wait state. If an interrupt occurs for Process A, the operating system suspends the execution of the first process, and stores the current information of Process A in its PCB and context to the second process namely Process B. In doing so, the program counter from the PCB of Process B is loaded, and thus execution can continue with the new process. The switching between two processes, Process A and Process B is illustrated in the *Figure 3* given below:

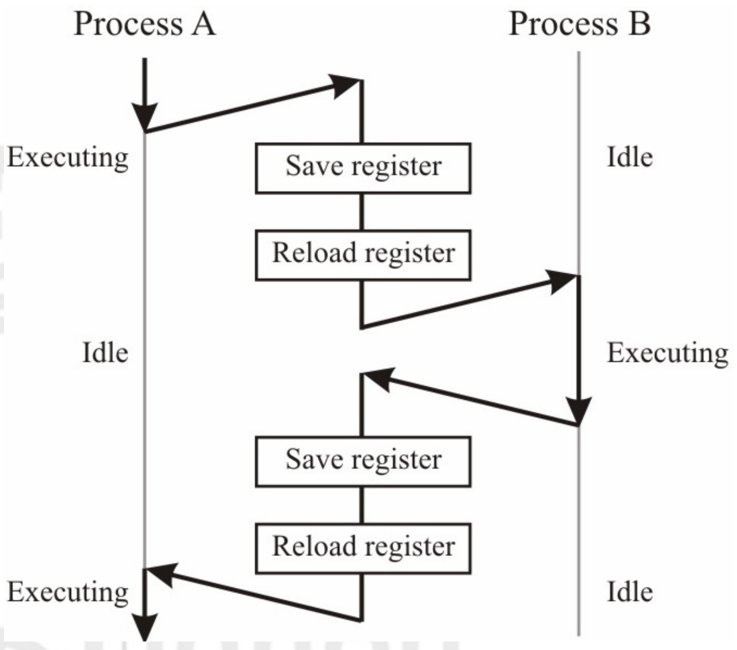


Figure 3: Process Switching between two processes

2.2.5 Process Hierarchy

Modern general purpose operating systems permit a user to create and destroy processes. A process may create several new processes during its time of execution. The creating process is called parent process, while the new processes are called child processes. There are different possibilities concerning creating new processes:

- **Execution**: The parent process continues to execute concurrently with its children processes or it waits until all of its children processes have terminated (sequential).
- **Sharing**: Either the parent and children processes share all resources (likes memory or files) or the children processes share only a subset of their parent's resources or the parent and children processes share no resources in common.

A parent process can terminate the execution of one of its children for one of these reasons:

- The child process has exceeded its usage of the resources it has been allocated. In order to do this, a mechanism must be available to allow the parent process to inspect the state of its children processes.
- The task assigned to the child process is no longer required.

Let us discuss this concept with an example. In UNIX this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process. After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes. This results in a process tree. The root of the tree is a special process created by the OS during startup. A process can *choose* to wait for children to terminate. For example, if C issued a wait( ) system call it would block until G finished. This is shown in the *Figure 4*.

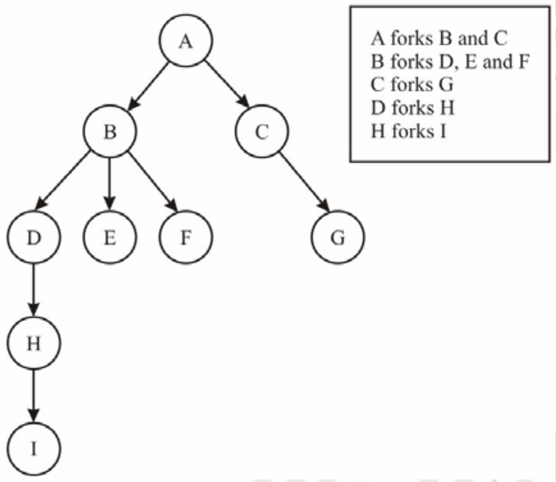


Figure 4: Process hierarchy

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.

2.2.6 Threads

Threads, sometimes called lightweight processes (LWPs) are independently scheduled parts of a single program. We say that a task is *multithreaded* if it is composed of several independent sub-processes which do work on common data, and if each of those pieces could (at least in principle) run in parallel.

If we write a program which uses threads – there is only one program, one executable file, one task in the normal sense. Threads simply enable us to split up that program into logically separate pieces, and have the pieces run independently of one another, until they need to communicate. In a sense, threads are a further level of *object orientation* for multitasking systems. They allow certain *functions* to be executed in parallel with others.

On a truly parallel computer (several CPUs) we might imagine parts of a program (different subroutines) running on quite different processors, until they need to communicate. When one part of the program needs to send data to the other part, the two independent pieces must be synchronized, or be made to wait for one another. But what is the point of this? We can always run independent procedures in a program as separate *programs*, using the process mechanisms we have already introduced. They could communicate using normal interprocesses communication. Why introduce another new concept? Why do we need threads?

The point is that threads are cheaper than normal processes, and that they can be scheduled for execution in a user-dependent way, with less overhead. Threads are cheaper than a whole process because they do not have a full set of resources each. Whereas the process control block for a heavyweight process is large and costly to context switch, the PCBs for threads are much smaller, since each thread has only a stack and some registers to manage. It has no open file lists or resource lists, no accounting structures to update. All of these resources are shared by all threads within the process. Threads can be assigned priorities – a higher priority thread will get put to the front of the queue. Let's define heavy and lightweight processes with the help of a table.

Object	**Resources**
Thread (Light Weight Processes)	Stack, set of CPU registers, CPU time
Task (Heavy Weight Processes)	1 thread, PCB, Program code, memory segment etc.
Multithreaded task	*n*- threads, PCB, Program code, memory segment etc.

Why use threads?

The sharing of resources can be made more effective if the scheduler knows known exactly what each program was going to do in advance. Of course, the scheduling algorithm can never know this – but the programmer who wrote the program does know. Using threads it is possible to organise the execution of a program in such a way that something is always being done, whenever the scheduler gives the heavyweight process CPU time.

- Threads allow a programmer to switch between lightweight processes when it is best for the program. (The programmer has control).
- A process which uses threads does not get more CPU time than an ordinary process – but the CPU time it gets is used to do work on the threads. It is possible to write a more efficient program by making use of threads.
- Inside a heavyweight process, threads are scheduled on a FCFS basis, unless the program decides to force certain threads to wait for other threads. If there is only one CPU, then only one thread can be running at a time.
- Threads context switch without any need to involve the kernel−the switching is performed by a user level library, so time is saved because the kernel doesn't need to know about the threads.

2.2.7 Levels of Threads

In modern operating systems, there are two levels at which threads operate: system or kernel threads and user level threads. If the kernel itself is multithreaded, the scheduler assigns CPU time on a thread basis rather than on a process basis. A kernel level thread behaves like a virtual CPU, or a power-point to which user-processes can connect in order to get computing power. The kernel has as many system level threads as it has CPUs and each of these must be shared between all of the user-threads on the system. In other words, the maximum number of user level threads which can be active at any one time is equal to the number of system level threads, which in turn is equal to the number of CPUs on the system.

Since threads work "inside" a single task, the normal process scheduler cannot normally tell which thread to run and which not to run – that is up to the program. When the kernel schedules a process for execution, it must then find out from that process which is the next thread it must execute. If the program is lucky enough to have more than one processor available, then several threads can be scheduled at the same time.

Some important implementations of threads are:

- The Mach System / OSF1 (user and system level)
- Solaris 1 (user level)

- Solaris 2 (user and system level)
- OS/2 (system level only)
- NT threads (user and system level)
- IRIX threads
- POSIX standardized user threads interface.

☞ **Check Your Progress 1**

1) Explain the difference between a process and a thread with some examples.

……………………………………………………………………………………..

…………..……………………………………………………………………………

……………………………………………………………………………………..

2) Identify the different states a live process may occupy and show how a process moves between these states.

……………………………………………………………………………………….

…..…………………………………………………………………………………

……………………………………………………………………………………..

3) Define what is meant by a context switch. Explain the reason many systems use two levels of scheduling.

………………………………………………………………………………………

…..……………………………………………………………………………………..

……………………………………………………………………………………….

2.3 SYSTEM CALLS FOR PROCESS MANAGEMENT

In this section we will discuss system calls typically provided by the kernels of multiprogramming operating systems for process management. System calls provide the interface between a process and the operating system. These system calls are the routine services of the operating system. As an example of how system calls are used, consider writing a simple program to read data from one file and to copy them to another file. There are two names of two different files in which one input file and the other is the output file. One approach is for the program to ask the user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the character that the two files have. Once the two file names are obtained the program must open the input file and create the output file. Each of these operations requires another system call and may encounter possible error conditions. When the program tries to open the input file, it may find that no file of that name exists or that the file is protected against access. In these cases the program should print the message on the console and then terminate abnormally which require another system call. If the input file exists then we must create the new output file. We may find an output file with the same name. This situation may cause the program to abort or we may delete the existing file and create a new one. After opening both files, we may enter a loop that reads from input file and writes to output file. Each read and write must return status information regarding various possible error conditions. Finally, after the entire file is copied the program may close both files. Examples of some operating system calls are:

Create: In response to the *create* call the operating system creates a new process with the specified or default attributes and identifier. Some of the parameters definable at the process creation time include:

- level of privilege, such as system or user
- priority
- size and memory requirements
- maximum data area and/or stack size
- memory protection information and access rights
- other system dependent data.

Delete: The delete service is also called destroy, terminate or exit. Its execution causes the operating system to destroy the designated process and remove it from the system.

Abort: It is used to terminate the process forcibly. Although a process could conceivably abort itself, the most frequent use of this call is for involuntary terminations, such as removal of malfunctioning process from the system.

Fork/Join: Another method of process creation and termination is by means of *FORK/JOIN* pair, originally introduced as primitives for multiprocessor system. The *FORK* operations are used to split a sequence of instruction into two concurrently executable sequences. *JOIN* is used to merge the two sequences of code divided by the *FORK* and it is available to a parent process for synchronization with a child.

Suspend: The *suspend* system call is also called *BLOCK* in some systems. The designated process is suspended indefinitely and placed in the *suspend* state. A process may be suspended itself or another process when authorised to do so.

Resume: The *resume* system call is also called *WAKEUP* in some systems. This call resumes the target process, which is presumably suspended. Obviously a suspended process can not resume itself because a process must be running to have its operating system call processed. So a suspended process depends on a partner process to issue the resume.

Delay: The system call *delay* is also known as *SLEEP*. The target process is suspended for the duration of the specified time period. The time may be expressed in terms of system clock ticks that are system dependent and not portable or in standard time units such as seconds and minutes. A process may delay itself or optionally, delay some other process.

Get_Attributes: It is an enquiry to which the operating system responds by providing the current values of the process attributes, or their specified subset, from the PCB.

Change Priority: It is an instance of a more general SET-PROCESS-ATTRIBUTES system call. Obviously, this call is not implemented in systems where process priority is static.

2.4 PROCESS SCHEDULING

Scheduling is a fundamental operating system function. All computer resources are scheduled before use. Since CPU is one of the primary computer resources, its scheduling is central to operating system design. **Scheduling** refers to a set of policies and mechanisms supported by operating system that controls the order in which the work to be done is completed. A **scheduler** is an operating system program (module) that selects the next job to be admitted for execution. The main objective of scheduling is to increase CPU utilisation and higher throughput. **Throughput** is the amount of work accomplished in a given **time interval.** CPU scheduling is the basis of operating system which supports multiprogramming concepts. By having a number of programs in computer memory at the same time, the CPU may be shared among them. This mechanism improves the overall efficiency of the computer system by getting more work done in less time. In this section we will describe the scheduling objectives, the three types of schedulers and performance criteria that schedulers may use in maximizing system performance. Finally at the end of the unit, we will study various scheduling algorithms.

2.4.1 Scheduling Objectives

The primary objective of scheduling is to improve system performance. Various objectives of the scheduling are as follows:

- **Maximize throughput:** Scheduling should attempt to service the largest possible number of processes per unit time.
- Maximize the number of interactive user receiving acceptable response times.
- **Be predictable:** A given job should utilise the same amount of time and should cost the same regardless of the load on the system.
- **Minimize overhead:** Scheduling should minimize the wasted resources overhead.
- **Balance resource use:** The scheduling mechanisms should keep the resources of the system busy. Processes that will use under utilized resources should be favoured.
- **Achieve a balance between response and utilisation:** The best way to guarantee good response times is to have sufficient resources available whenever they are needed. In real time system fast responses are essential, and resource utilisation is less important.
- **Avoid indefinite postponement :** It would be fair if all processes are treated the same, and no process can suffer indefinite postponement.
- **Enforce Priorities:** In environments in which processes are given priorities, the scheduling mechanism should favour the higher-priority processes.
- **Give preference to processes holding key resources:** Even though a low priority process may be holding a key resource, the process may be in demand by high priority processes. If the resource is not perceptible, then the scheduling mechanism should give the process better treatment that it would ordinarily receive so that the process will release the key resource sooner.
- **Degrade gracefully under heavy loads:** A scheduling mechanism should not collapse under heavy system load. Either it should prevent excessive loading by not allowing new processes to be created when the load in heavy or it should provide service to the heavier load by providing a moderately reduced level of service to all processes.

2.4.2 Types of Schedulers

If we consider batch systems, there will often be more processes submitted than the number of processes that can be executed immediately. So, the incoming processes are spooled onto a disk. There are three types of schedulers. They are:

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

Short term scheduler: The short-term scheduler selects the process for the processor among the processes which are already in queue (in memory). The scheduler will execute quite frequently (mostly at least once every 10 milliseconds). It has to be very fast in order to achieve a better processor utilisation. The short term scheduler, like all other OS programs, has to execute on the processor. If it takes 1 millisecond to choose a process that means (1 / (10 + 1)) = 9% of the processor time is being used for short time scheduling and only 91% may be used by processes for execution.

Long term scheduler: The long-term scheduler selects processes from the process pool and loads selected processes into memory for execution. The long-term scheduler executes much less frequently when compared with the short term scheduler. It controls the degree of multiprogramming (no. of processes in memory at a time). If the degree of multiprogramming is to be kept stable (say 10 processes at a time), the long-term scheduler may only need to be invoked till the process finishes execution. The long-term scheduler must select a good process mix of I/O-bound and processor bound processes. If most of the processes selected are I/O-bound, then the ready queue will almost be empty, while the device queue(s) will be very crowded. If most of the processes are processor-bound, then the device queue(s) will almost be empty while the ready queue is very crowded and that will cause the short-term scheduler to be invoked very frequently. Time-sharing systems (mostly) have no long-term scheduler. The stability of these systems either depends upon a physical limitation (no. of available terminals) or the self-adjusting nature of users (if you can't get response, you quit). It can sometimes be good to reduce the degree of multiprogramming by removing processes from memory and storing them on disk. These processes can then be reintroduced into memory by the **medium-term scheduler.** This operation is also known as swapping. Swapping may be necessary to improve the process mix or to free memory.

2.4.3 Scheduling Criteria

The goal of scheduling algorithm is to identify the process whose selection will result in the "best" possible system performance. There are different scheduling algorithms, which has different properties and may favour one class of processes over another, which algorithm is best, to determine this there are different characteristics used for comparison. The scheduling algorithms which we will discuss in the next section, determines the importance of each of the criteria.

In order to achieve an efficient processor management, OS tries to select the most appropriate process from the ready queue. For selecting, the relative importance of the following may be considered as performance criteria:

CPU Utilization: The key idea is that if the CPU is busy all the time, the utilization factor of all the components of the system will be also high. CPU utilization is the ratio of busy time of the processor to the total time passes for processes to finish.

Processor Utilization = (Processor busy time) / (Processor busy time + Processor idle time)

Throughput: It refers to the amount of work completed in a unit of time. One way to measure throughput is by means of the number of processes that are completed in a unit of time. The higher the number of processes, the more work apparently is being done by the system. But this approach is not very useful for comparison because this is dependent on the characteristics and resource requirement of the process being executed. Therefore to compare throughput of several scheduling algorithms it should be fed into the process with similar requirements. The throughput can be calculated by using the formula:

Throughput = (No. of processes completed) / (Time unit)

Turnaround Time : It may be defined as interval from the time of submission of a process to the time of its completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, CPU time and I/O operations.

Turnaround Time = t(Process completed) – t(Process submitted)

Waiting Time: This is the time spent in the ready queue. In multiprogramming operating system several jobs reside at a time in memory. CPU executes only one job at a time. The rest of jobs wait for the CPU. The waiting time may be expressed as turnaround time, less than the actual processing time.

Waiting time = Turnaround Time - Processing Time

But the scheduling algorithm affects or considers the amount of time that a process spends waiting in a ready queue. Thus rather than looking at turnaround time waiting time is usually the waiting time for each process.

Response time: It is most frequently considered in time sharing and real time operating system. However, its characteristics differ in the two systems. In time sharing system it may be defined as interval from the time the last character of a command line of a program or transaction is entered to the time the last result appears on the terminal. In real time system it may be defined as interval from the time an internal or external event is signalled to the time the first instruction of the respective service routine is executed.

Response time = t(first response) – t(submission of request)

One of the problems in designing schedulers and selecting a set of its performance criteria is that they often conflict with each other. For example, the fastest response time in time sharing and real time system may result in low CPU utilisation.

Throughput and CPU utilization may be increased by executing the large number of processes, but then response time may suffer. Therefore, the design of a scheduler usually requires balance of all the different requirements and constraints. In the next section we will discuss various scheduling algorithms.

☞ Check Your Progress 2

1) Distinguish between a foreground and a background process in UNIX.

…………………………………………………………………………….……...

……………………………………………………………………………………

……………………………………………………………………………………

2) Identify the information which must be maintained by the operating system for each live process.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2.5 SCHEDULING ALGORITHMS

Now let's discuss some processor scheduling algorithms again stating that the goal is to select the most appropriate process in the ready queue.

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU. There are several scheduling algorithms which will be examined in this section. A major division among scheduling algorithms is that whether they support **pre-emptive** or **non-preemptive scheduling** discipline.

Preemptive scheduling: Preemption means the operating system moves a process from running to ready without the process requesting it. An OS implementing this algorithms switches to the processing of a new request before completing the processing of the current request. The preempted request is put back into the list of the pending requests. Its servicing would be resumed sometime in the future when it is scheduled again. Preemptive scheduling is more useful in high priority process which requires immediate response. For example, in Real time system the consequence of missing one interrupt could be dangerous.

Round Robin scheduling, priority based scheduling or event driven scheduling and SRTN are considered to be the preemptive scheduling algorithms.

Non–Preemptive scheduling: A scheduling discipline is non-preemptive if once a process has been allotted to the CPU, the CPU cannot be taken away from the process. A non-preemptible discipline always processes a scheduled request to its completion. In non-preemptive systems, jobs are made to wait by longer jobs, but the treatment of all processes is fairer.

First come First Served (FCFS) and Shortest Job First (SJF), are considered to be the non-preemptive scheduling algorithms.

The decision whether to schedule preemptive or not depends on the environment and the type of application most likely to be supported by a given operating system.

2.5.1 First-Come First-Serve (FCFS)

The simplest scheduling algorithm is First Come First Serve (FCFS). Jobs are scheduled in the order they are received. FCFS is non-preemptive. Implementation is easily accomplished by implementing a queue of the processes to be scheduled or by storing the time the process was received and selecting the process with the earliest time.

FCFS tends to favour CPU-Bound processes. Consider a system with a CPU-bound process and a number of I/O-bound processes. The I/O bound processes will tend to execute briefly, then block for I/O. A CPU bound process in the ready should not have to wait long before being made runable. The system will frequently find itself with all the I/O-Bound processes blocked and CPU-bound process running. As the I/O operations complete, the ready Queue fill up with the I/O bound processes.

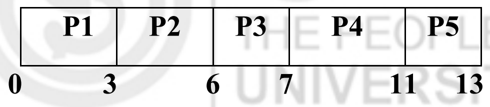
Under some circumstances, CPU utilisation can also suffer. In the situation described above, once a CPU-bound process does issue an I/O request, the CPU can return to process all the I/O-bound processes. If their processing completes before the CPU-bound process's I/O completes, the CPU sits idle. So with no preemption, component utilisation and the system throughput rate may be quite low.

Example:

Calculate the turn around time, waiting time, average turnaround time, average waiting time, throughput and processor utilization for the given set of processes that arrive ***at a given arrive time*** shown in the table, with the length of processing time given in milliseconds:

Process	Arrival Time	Processing Time
P1	0	3
P2	2	3
P3	3	1
P4	5	4
P5	8	2

If the processes arrive as per the arrival time, the Gantt chart will be



Time	Process Completed	Turn around Time = t(Process Completed) –t(Process Submitted)	Waiting Time = Turn around time – Processing time
0	-	-	-
3	P1	3 – 0 = 3	3 – 3 = 0
6	P2	6 – 2 = 4	4 – 3 = 1
7	P3	7 – 1 = 6	6 – 1 = 5
11	P4	11– 4 = 7	7 – 4 = 3
13	P5	13 – 2 = 11	11– 2 = 9

Average turn around time = (3+4+6+7+11) / 5 = 6.2

Average waiting time = (0+1+5+3+9) / 5 = 3.6

Processor utilization = (13/13)*100 = 100%

Throughput = 5/13 = 0.38

Note: If all the processes arrive at time 0, then the order of scheduling will be P3, P5, P1, P2 and P4.

2.5.2 Shortest-Job First (SJF)

This algorithm is assigned to the process that has smallest next CPU processing time, when the CPU is available. In case of a tie, FCFS scheduling algorithm can be used. It is originally implemented in a batch-processing environment. SJF relied on a time estimate supplied with the batch job.

As an example, consider the following set of processes with the following processing time which arrived at the same time.

Process	**Processing Time**
P1	06
P2	08
P3	07
P4	03

Using SJF scheduling because the shortest length of process will first get execution, the Gantt chart will be:

P4	**P1**	**P3**	**P2**

0 3 9 16 24

Because the shortest processing time is of the process P4, then process P1 and then P3 and Process P2. The waiting time for process P1 is 3 ms, for process P2 is 16 ms, for process P3 is 9 ms and for the process P4 is 0 ms as –

Time	**Process Completed**	**Turn around Time = t (Process Completed)– t (Process Submitted)**	**Waiting Time = Turn around time – Processing time**
0	-	-	-
3	P4	3 – 0 = 3	3 – 3 = 0
9	P1	9 – 0 = 9	9 – 6 = 3
16	P3	16 – 0 = 16	16 – 7 = 9
24	P2	24 – 0= 24	24 – 8 = 16

Average turn around time = (3+9+16+24) / 4 = 13

Average waiting time = (0+3+9+16) / 4 = 7

Processor utilization = (24/24)*100 = 100%

Throughput = 4/24 = 0.16

2.5.3 Round Robin (RR)

Round Robin (RR) scheduling is a preemptive algorithm that relates the process that has been waiting the longest. This is one of the oldest, simplest and widely used algorithms. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system environment where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. Basically the CPU time is divided into time slices.

Each process is allocated a small time-slice called quantum. No process can run for more than one quantum while others are waiting in the ready queue. If a process needs more CPU time to complete after exhausting one quantum, it goes to the end of ready queue to await the next allocation. To implement the RR scheduling, Queue data structure is used to maintain the Queue of Ready processes. A new process is added at the tail of that Queue. The CPU schedular picks the first process from the ready Queue, Allocate processor for a specified time Quantum. After that time the CPU schedular will select the next process is the ready Queue.

Consider the following set of process with the processing time given in milliseconds.

Process	Processing Time
P1	24
P2	03
P3	03

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

P1	**P2**	**P3**	**P1**	**P1**	**P1**	**P1**	**P1**

0 4 7 10 14 18 22 26 30

Process	**Processing Time**	**Turn around time = t(Process Completed) – t(Process Submitted)**	**Waiting Time = Turn around time – Processing time**
P1	24	30 – 0 = 30	30 – 24 = 6
P2	03	7 – 0 = 7	7 – 3 = 4
P3	03	10 – 0 =10	10 – 3 = 7

Average turn around time = (30+7+10)/3 = 47/3 = 15.66

Average waiting time = (6+4+7)/3 = 17/3 = 5.66

Throughput = 3/30 = 0.1

Processor utilization = (30/30) * 100 = 100%

2.5.4 Shortest Remaining Time Next (SRTN)

This is the preemptive version of shortest job first. This permits a process that enters the ready list to preempt the running process if the time for the new process (or for its next burst) is less than the *remaining* time for the running process (or for its current burst). Let us understand with the help of an example.

Consider the set of four processes arrived as per timings described in the table:

Process	Arrival time	Processing time
P1	0	5
P2	1	2
P3	2	5
P4	3	3

At time 0, only process P1 has entered the system, so it is the process that executes. At time 1, process P2 arrives. At that time, process P1 has 4 time units left to execute At this juncture process 2's processing time is less compared to the P1 left out time (4 units). So P2 starts executing at time 1. At time 2, process P3 enters the system with the processing time 5 units. Process P2 continues executing as it has the minimum number of time units when compared with P1 and P3. At time 3, process P2 terminates and process P4 enters the system. Of the processes P1, P3 and P4, P4 has the smallest remaining execution time so it starts executing. When process P1 terminates at time 10, process P3 executes. The Gantt chart is shown below:

P1	**P2**	**P4**	**P1**	**P3**

0 1 3 6 10 15

Turnaround time for each process can be computed by subtracting the time it terminated from the arrival time.

Turn around Time = t(Process Completed)– t(Process Submitted)

The turnaround time for each of the processes is:

P1: 10 – 0 = 10

P2: 3 – 1 = 2

P3: 15 – 2 = 13

P4: 6 – 3 = 3

The average turnaround time is (10+2+13+3) / 4 = 7

The waiting time can be computed by subtracting processing time from turnaround time, yielding the following 4 results for the processes as

P1: 10 – 5 = 5

P2: 2 – 2 = 0

P3: 13 – 5 = 8

P4: 3 – 3 = 0

The average waiting time = (5+0+8+0) / 4 = 3.25milliseconds

Four jobs executed in 15 time units, so throughput is 15 / 4 = 3.75 time units/job.

2.5.5 Priority Based Scheduling or Event-Driven (ED) Scheduling

A priority is associated with each process and the scheduler always picks up the highest priority process for execution from the ready queue. Equal priority processes are scheduled FCFS. The level of priority may be determined on the basis of resource requirements, processes characteristics and its run time behaviour.

A major problem with a priority based scheduling is indefinite blocking of a lost priority process by a high priority process. In general, completion of a process within finite time cannot be guaranteed with this scheduling algorithm. A solution to the problem of indefinite blockage of low priority process is provided by aging priority. Aging priority is a technique of gradually increasing the priority of processes (of low priority) that wait in the system for a long time. Eventually, the older processes attain high priority and are ensured of completion in a finite time.

As an example, consider the following set of five processes, assumed to have arrived at the same time with the length of processor timing in milliseconds: –

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

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

P2	**P5**	**P1**	**P3**	**P4**

0 1 6 16 18 19

Time	**Process Completed**	**Turn around Time = t(Process Completed) – t(Process Submitted)**	**Waiting Time = Turn around time – Processing time**
0	-	-	-
1	P2	1 – 0 = 1	1 – 1 = 0
6	P5	6 – 0 = 6	6 – 2 = 4
16	P1	16 – 0 = 16	16 – 10 = 6
18	P3	18 – 0 = 18	18 – 2 = 16
19	P4	19 – 0 = 19	19 – 1 = 18

Average turn around time = (1+6+16+18+19) / 5 = 60/5 = 12
Average waiting time = (6+0+16+18+1) / 5 = 8.2
Throughput = 5/19 = 0.26
Processor utilization = (30/30) * 100 = 100%

Priorities can be defined either internally or externally. Internally defined priorities use one measurable quantity or quantities to complete the priority of a process.

2.6 PERFORMANCE EVALUATION OF THE SCHEDULING ALGORITHMS

Performance of an algorithm for a given set of processes can be analysed if the appropriate information about the process is provided. But how do we select a CPU-scheduling algorithm for a particular system? There are many scheduling algorithms so the selection of an algorithm for a particular system can be difficult. To select an algorithm there are some specific criteria such as:

- Maximize CPU utilization with the maximum response time.
- Maximize throughput.

For example, assume that we have the following five processes arrived at time 0, in the order given with the length of CPU time given in milliseconds.

Process	**Processing time**
P1	10
P2	29
P3	03
P4	07
P5	12

First consider the FCFS scheduling algorithm for the set of processes.

For FCFS scheduling the Gantt chart will be:

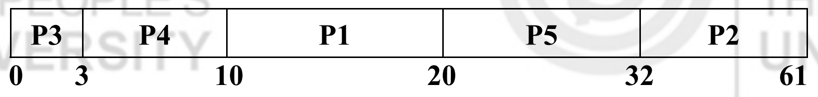
P1	**P2**	**P3**	**P4**	**P5**

0 10 39 42 49 61

Process	**Processing time**	**Waiting time**
P1	10	0
P2	29	10
P3	03	39
P4	07	42
P5	12	49

Average Waiting Time: (0+10+39+42+49) / 5 = 28 milliseconds.

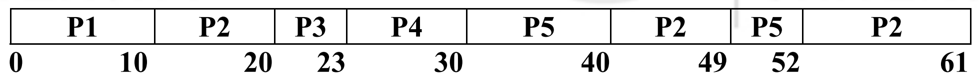
Now consider the SJF scheduling, the Gantt chart will be:



Process	Processing time	Waiting time
P1	10	10
P2	29	32
P3	03	00
P4	07	03
P5	12	20

Average Waiting Time: (10+32+0+3+20)/5 = 13 milliseconds.

Now consider the Round Robin scheduling algorithm with a quantum of 10 milliseconds. The Gantt chart will be:



Process	Processing time	Waiting time
P1	10	0
P2	29	32
P3	03	20
P4	07	23
P5	12	40

Average waiting time = (0+32+20+23+40) / 5 = 23 milliseconds

Now if we compare average waiting time above algorithms, we see that SJF policy results in less than one half of the average waiting time to that of FCFS scheduling; the RR algorithm gives us an intermediate value.

So performance of algorithm can be measured when all the necessary information is provided.

☞ Check Your Progress 3

1) Explain the difference between voluntary or co-operative scheduling and preemptive scheduling. Give two examples of preemptive and of non-preemptive scheduling algorithms.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2) Outline how different process priorities can be implemented in a scheduling algorithm. Explain why these priorities need to be dynamically adjusted.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3) Draw the Gantt chart for the FCFS policy, considering the following set of processes that arrive at time 0, with the length of CPU time given in milliseconds. Also calculate the Average Waiting Time.

Process	Processing Time
P1	13
P2	08
P3	83

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4) For the given five processes arriving at time 0, in the order with the length of CPU time in milliseconds:

Process	Processing Time
P1	10
P2	29
P3	03
P4	07

Consider the FCFS, SJF and RR (time slice= 10 milliseconds) scheduling algorithms for the above set of process which algorithm would give the minimum average waiting time?

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2.7 SUMMARY

A process is an instance of a program in execution. A process can be defined by the system or process is an important concept in modem operating system. Processes provide a suitable means for informing the operating system about independent activities that may be scheduled for concurrent execution. Each process is represented by a process control block (PCB). Several PCB's can be linked together to form a queue of waiting processes. The selection and allocation of processes is done by a scheduler. There are several scheduling algorithms. We have discussed FCFS, SJF, RR, SJRT and priority algorithms along with the performance evaluation.

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A process is an instance of an executing program and its data. For example, if you were editing 3 files simultaneously, you would have 3 processes, even though they might be sharing the same code.

A thread is often called a lightweight process. It is a "child" and has all the state information associated with a child process but does not run in a separate address space i.e., doesn't have its own memory. Since they share memory, each thread has access to all variables and any change made by one thread impacts all the others. Threads are useful in multi-client servers where each thread can service a separate client connection. In some stand-alone processes where you can split the processing such that one thread can continue processing while another is blocked e.g., waiting for an I/O request or a timer thread can implement a regular repaint of a graphics canvas.

This is how simple animations are accomplished in systems like NT and Windows95. Unix is a single-threaded operating system and can't do this at the operating system level although some applications e.g., Java can support it via software.

2) A process can be in one of the following states:

Ready i.e., in the scheduling queue waiting it's turn for the CPU
Running i.e., currently occupying the CPU
Blocked i.e., sleeping waiting for a resource request to be satisfied.
Halted i.e., process execution is over.

- A process moves from ready to running, when it is dispatched by the scheduler.
- A process moves from running to ready, when it is pre-empted by the scheduler.
- A process moves from running to blocked when it issues a request for a resource.
- A process moves from blocked to ready when completion of the request is signaled.

3) A context switch occurs whenever a different process is moved into the CPU. The current state of the existing process (register and stack contents, resources like open files etc.) must be stored and the state of the new process restored. There is a significant overhead in context switching. However, in a system with Virtual Memory, some processes and some of the state information will be swapped to disks. The overhead involved here is too great to undertake each quantum. Such systems use a two-level scheduler. The high-level scheduler is responsible for bring ready processes from disk into memory and adding them to ready queue. This is done regularly (e.g., every 10 secs) but not each quantum (e.g., < every 100 msecs). The low level scheduler is responsible for the context switching between CPU and ready queue.

Check Your Progress 2

1) Every process has a parent which invoked it. In the UNIX environment, the parent can wait for the child to complete after invoking it (foreground child process) or continue in a ready state (background child process).

2) For each process, the operating system needs to maintain

- id information - process id, parent process id
- summary status - blocked, ready, running, swapped to disk
- owner information - user id, group id
- scheduling info - priority, nice value, CPU usage
- location info - resident or not, memory areas allocated
- state info - register values (instruction pointer etc.), stack, resources like open files etc.

Check Your Progress 3

1) Preemptive multi-tasking means that the operating system decides when a process has had its quantum of CPU, and removes it from the CPU to the ready queue. This is present in UNIX, NT and Windows 95.
Voluntary and co-operative means that only the process itself decide to vacate the CPU e.g., when it finishes or blocks on a request or yields the CPU. This is what Windows3.x uses.

First Come First Served is a typical non-premptive algorithm.
Round Robin and Priority Queues are typical preemptive algorithms.
Shortest Job First can be either.

2) Some processes are more critical to overall operation than others e.g., kernel activity. I/O bound processes, which seldom use a full quantum, need a boost over compute bound tasks. This priority can be implemented by any combination of higher positioning in the ready queue for higher priority more frequent occurrences in the ready list for different priorities different quantum lengths for different priorities. Priority scheduling is liable to **starvation**. A succession of high priority processes may result in a low priority process never reaching the head of the ready list. Priority is normal a combination of static priority (set by the user or the class of the process) and a dynamic adjustment based on it's history (e.g., how long it has been waiting). NT and Linux also boost priority when processes return from waiting for I/O.

3) If the process arrives in the order P1, P2 and P3, then the Gantt chart will be as:

```
|    P1    |     P2      |               P3                |
0         13            21                               104
```

Process	Processing time	Waiting Time
P1	13	00
P2	08	13
P3	83	21

Average Waiting Time: (0+13+21)/3 = 11.33 ms.

4) For FCFS algorithm the Gantt chart is as follows:

```
| P1          | P2       | P3      | P4        | P5              |
0            10         39        42          49                51
```

Process	Processing Time	Waiting time
P1	10	0
P2	29	10
P3	3	39
P4	7	42
P5	12	49

Average Waiting Time = (0+10+39+42+49) / 5 = 5

2. With SJF scheduling algorithm, we have

```
| P3      | P4          | P1            | P5            | P2                      |
0         3            10              20              32                        61
```

Process	Processing Time	Waiting time
P1	10	10
P2	29	32
P3	3	00
P4	7	3
P5	12	20

Average Waiting Time = (10+32+00+03+20)/5 = 13 milliseconds.
With round robin scheduling algorithm (time quantum = 10 milliseconds)

P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	
0	**10**	**20**	**23**	**30**	**40**	**50**	**52**	**61**

Process	**Processing Time**	**Waiting time**
P1	10	0
P2	29	32
P3	03	20
P4	07	23
P5	12	40

Average Waiting Time = (0+32+20+23+40)/5 = 23 milliseconds.

From the above calculations of average waiting time we found that SJF policy results in less than one half of the average waiting time obtained from FCFS, while Round Robin gives intermediate result.

2.9 FURTHER READINGS

1) Abraham Silberschatz and James L, *Operating system Concepts*. Peterson, Addition Wesely Publishing Company.

2) Andrew S. Tanenbaum, *Operating System Design and Implementation,* PHI

3) D.M. Dhamdhere, *Operating Systems, A Concept-based Approach*, TMGH, 2002.

4) Harvay M. Deital, *Introduction to Operating Systems,* Addition Wesely Publishing Company

5) Madnick and Donovan, *Operating systems – Concepts and Design* Mc GrawHill Intl. Education.

6) Milan Milenkovic, *Operating Systems, Concepts and Design*, TMGH, 2000.