Unit 3 Process Management

Structure:

3.1 Introduction

Dear student, in the last unit you have studied the various architecture of the operating systems. In this unit you are going to study about process management. Current day computer system allows multiple programs to be loaded into memory and to be executed concurrently. This evolution requires more coordination and firmer control of the various programs. These needs resulted in the notion of a process. A Process can be simply defined as a program in execution. A process is created and terminated, and it allows some or all of the states of process transition; such as New, Ready, Running, Waiting and Exit.

Thread is single sequence stream which allows a program to split itself into two or more simultaneously running tasks. Threads and processes differ from one operating system to another but in general, a thread is contained inside a process and different threads in the same process share some resources while different processes do not. An operating system that has thread facility, the basic unit of CPU utilization, is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads share with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Objectives:

After studying this unit, you should be able to:

- explain the process state
- describe the process control block
- discuss process scheduling
- explain operations on processes and cooperating processes
- describe threads in operating systems

3.2 Process State

Dear student, a program in execution is a process. A process is executed sequentially, one instruction at a time. A program is a passive entity. Example: a file on the disk. A process on the other hand is an active entity. In addition to program code, it includes the values of the program counter, the contents of the CPU registers, the global variables in the data section and the contents of the stack that is used for subroutine calls. In reality, the CPU switches back and forth among processes.

A process being an active entity, changes state as execution proceeds. A process can be any one of the following states:

- New: Process being created.
- Running: Instructions being executed.
- Waiting (Blocked): Process waiting for an event to occur.
- Ready: Process waiting for CPU.
- Terminated: Process has finished execution.

Locally, the ‘Running’ and ‘Ready’ states are similar. In both cases the process is willing to run, only in the case of ‘Ready’ state, there is temporarily no CPU available for it. The ‘Blocked’ state is different from the ‘Running’ and ‘Ready’ states in that the process cannot run, even if the CPU is available.

These above states are arbitrary and very between operating systems. Certain operating systems also distinguish among more finely delineating process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting. A state diagram (Figure 3.1) is used to diagrammatically represent the states and also the events that trigger the change of state of a process in execution.

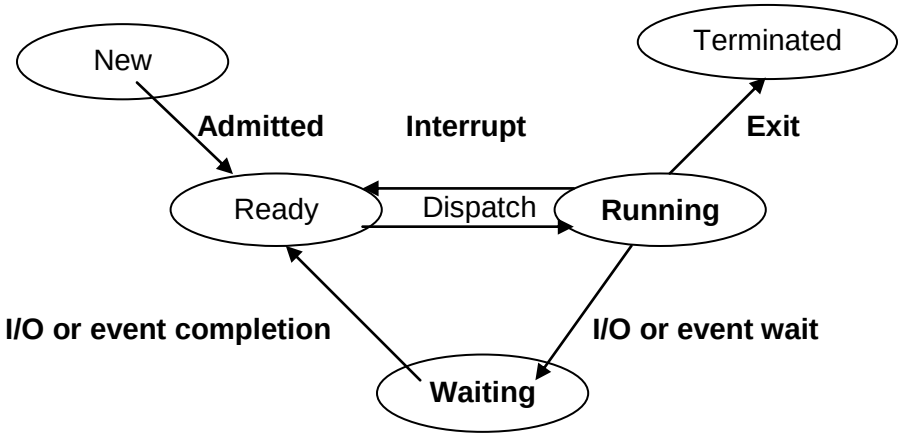

Fig. 3.1: Process state diagram

3.3 Process Control Block

Every process has a number and a process control block (PCB) represents a process in an operating system. The PCB serves as a repository of information about a process and varies from process to process. The PCB contains information that makes the process an active entity. A PCB is shown in Figure 3.2. It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, and waiting, halted and so on.
- **Program counter:** It is a register and it holds the address of the next instruction to be executed.
- **CPU Registers:** These registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers, plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.
- **CPU scheduling information:** This information includes a process priority, pointer to scheduling queues and other scheduling parameters.
- **Memory management information**: This information includes the value of the base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system.

- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The PCB simply serves as the repository of any information that may vary from process to process.

Pointer	Process State
Process number	
Program counter	
Registers	
Memory limits	
List of open files	
…	

Fig. 3.2: The process control block

Self Assessment Questions

1. A process is executed in parallel, multiple instructions at a time. (True / False)
2. A _____ serves as a repository of information about a process and varies from process to process.
3. When a process is waiting for CPU, we can say it is in _____ state.
   a) Ready
   b) New
   c) Running
   d) Waiting

3.4 Process Scheduling

The main objective of multiprogramming is to see that some process is always running so as to maximize CPU utilization whereas in the case of time sharing, the CPU is to be switched between processes frequently, so that users interact with the system while their programs are executing. In a uniprocessor system, there is always a single process running while the

other processes need to wait till they get the CPU for execution on being scheduled.

As a process enters the system, it joins a job queue that is a list of all processes in the system. Some of these processes are in the ready state and are waiting for the CPU for execution. These processes are present in a ready queue. The ready queue is nothing but a list of PCB's implemented as a linked list with each PCB pointing to the next.

There are also some processes that are waiting for some I/O operation like reading from a file on the disk or writing onto a printer. Such processes are present in device queues. Each device has its own queue.

A new process first joins the ready queue. When it gets scheduled, the CPU executes the process until-

1) An I/O occurs and the process gives up the CPU to join a device queue only to rejoin the ready queue after being serviced for the I/O.
2) It gives up the CPU on expiry of its time slice and rejoins the ready queue.

Every process is in this cycle until it terminates (Figure 3.3). Once a process terminates, entries for this process in all the queues are deleted. The PCB and resources allocated to it are released.

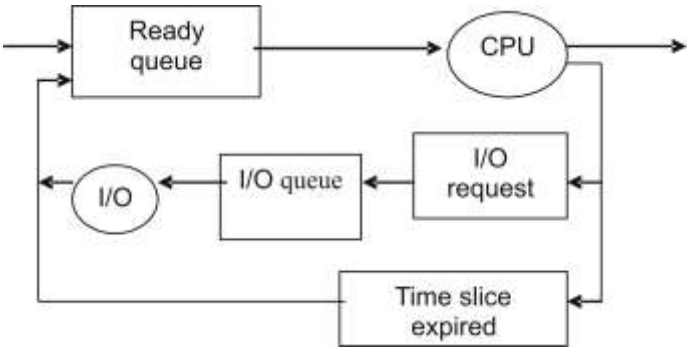

Fig. 3.3: Queuing diagram of process scheduling

Schedulers

At any given instant of time, a process is in any one of the new, ready, running, waiting or terminated state. Also a process moves from one state to another as long as it is active. The operating system scheduler schedules processes from the ready queue for execution by the CPU. The scheduler

selects one of the many processes from the ready queue based on certain criteria.

Schedulers could be any one of the following:

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

Many jobs could be ready for execution at the same time. Out of these more than one could be spooled onto the disk. The long-term scheduler or job scheduler as it is called picks and loads processes into memory from among the set of ready jobs. The short-term scheduler or CPU scheduler selects a process from among the ready processes to execute on the CPU.

The long-term scheduler and short-term scheduler differ in the frequency of their execution. A short-term scheduler has to select a new process for CPU execution quite often as processes execute for short intervals before waiting for I/O requests. Hence, short-term scheduler must be very fast or else, the CPU will be doing only scheduling work.

A long-term scheduler executes less frequently since new processes are not created at the same pace at which processes need to be executed. The number of processes present in the ready queue determines the degree of multi-programming. So the long-term scheduler determines this degree of multi-programming. If a good selection is made by the long-term scheduler in choosing new jobs to join the ready queue, then the average rate of process creation must almost equal the average rate of processes leaving the system. The long-term scheduler is thus involved only when processes leave the system to bring in a new process so as to maintain the degree of multi-programming.

Choosing one job from a set of ready jobs to join the ready queue needs careful selection. Most of the processes can be classified as either I/O bound or CPU bound depending on the time spent for I/O and time spent for CPU execution. I/O bound processes are those which spend more time executing I/O whereas CPU bound processes are those which require more CPU time for execution. A good selection of jobs by the long-term scheduler will give a good mix of both CPU bound and I/O bound processes. In this case, the CPU as well as the I/O devices will be busy. If this is not to be so,

then either the CPU is busy or I/O devices are idle or vice-versa. Time sharing systems usually do not require the services of a long-term scheduler since every ready process gets one time slice of CPU time at a time in rotation.

Sometimes processes keep switching between ready, running and waiting states with termination taking a long time. One of the reasons for this could be an increased degree of multi-programming meaning more number of ready processes than the system can handle. The medium-term scheduler (Figure 3.4) handles such a situation. When system throughput falls below a threshold, some of the ready processes are swapped out of memory to reduce the degree of multi-programming. Sometime later these swapped processes are reintroduced into memory to join the ready queue.

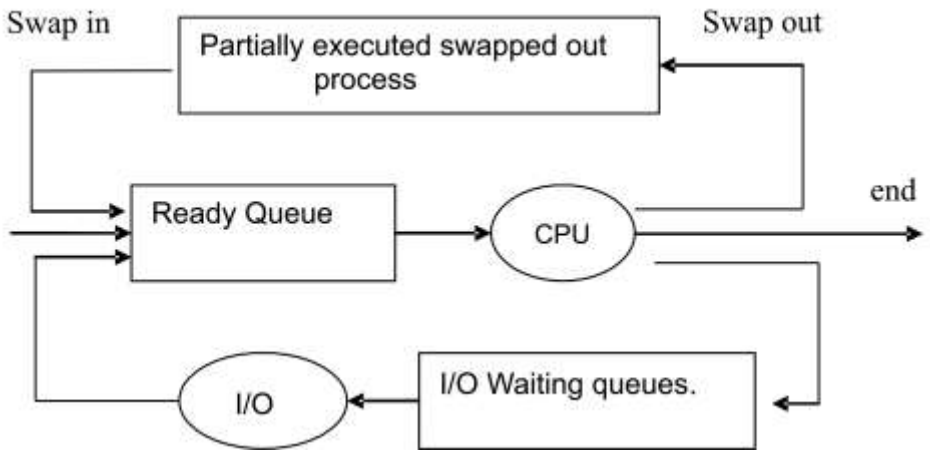

Fig. 3.4: Medium term scheduling

Context Switch

CPU switching from one process to another requires saving the state of the current process and loading the latest state of the next process. This is known as a Context Switch (Figure 3.5). Time that is required for a context switch is a clear overhead since the system at that instant of time is not doing any useful work. Context switch time varies from machine to machine depending on speed, the amount of saving and loading to be done, hardware support and so on.

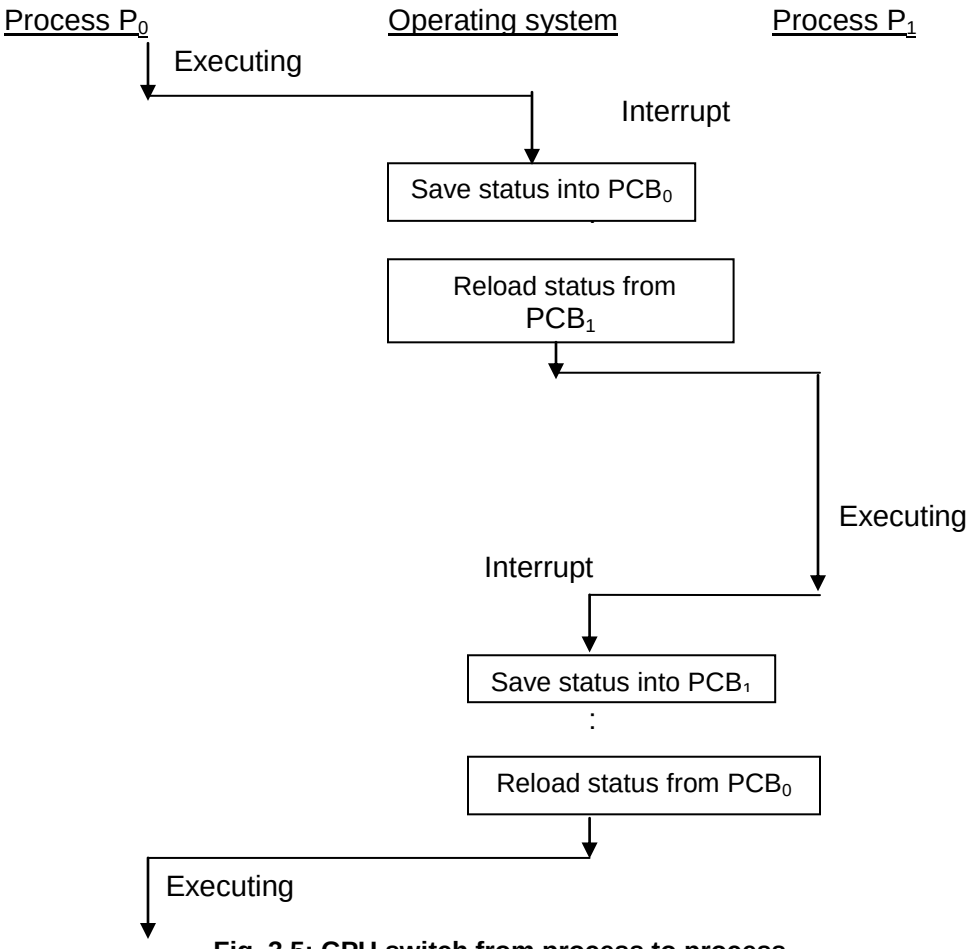

Fig. 3.5: CPU switch from process to process

Self Assessment Questions

4. The main objective of multiprogramming is to see that some process is always running so as to maximize CPU utilization. (True / False)
5. The operating system _____ schedules processes from the ready queue for execution by the CPU.
6. CPU switching from one process to another requires saving the state of the current process and loading the latest state of the next process. This is known as a _____.
   a) Program Switch
   b) Context Switch
   c) Process Switch
   d) OS Switch

3.5 Operation on Processes

The processes in the system can execute concurrently, and must be created and deleted dynamically. Thus the operating system must provide a mechanism for process creation and termination.

Process Creation

During the course of execution, a process may create several new processes using a create-process system call. The creating process is called a parent process and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

A process in general needs certain resources (CPU time, memory, I/O devices) to accomplish its task. When a process creates a sub process, the sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.

When a process creates a new process, two possibilities exit in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process, the first possibility is,

- The child process is a duplicate of the parent process.
  An example for this is UNIX operating system in which each process is identified by its process identifier, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

The second possibility is,

- The child process has a program loaded into it.

An example for this implementation is the DEC VMS operating system, it creates a new process, loads a specified program into that process, and starts it running. The Microsoft Windows/NT operating system supports both models: the parent's address space may duplicate, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

Process Termination

A process terminates when it finishes executing its last statement and asks the operating system to delete it by using the exit system call. At that time, the process should return the data (output) ti its parent process via the wait system call. All the resources of the process, including physical and virtual memory, open files, and I/O buffers, are de-allocated by the operating system.

Only a parent process can cause termination of its children via abort system call. For that a parent needs to know the identities of its children. When one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

To illustrate process execution and termination, let us consider UNIX system. In UNIX system a process may terminate by using the exit system call and its parent process may wait for that event by using the wait system call. The wait system call returns the process identifier of a terminated child, so the parent can tell which of the possibly many children has terminated. If the parent terminates, however all the children are terminated by the operating system. Without a parent, UNIX does not know whom to report the activities of a child.

3.6 Co-operating Processes

The processes executing in the operating system may be either independent processes or cooperating processes. A process is said to be independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share any data with any other process is independent. On the other hand, a process is co-operating if it can affect or be affected by the other processes executing in the system. Clearly any process that shares data with other processes is a co-operating process.

There are several advantages of providing an environment that allows process co-operation:

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

Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

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

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

Self Assessment Questions

7. An application program provides a mechanism for process creation and termination. (True / False)
8. A process terminates when it finishes executing its last statement and asks the operating system to delete it by using the _____.
9. Any process that shares data with other processes is a _____.
   a) Main process
   b) Independent process
   c) Co-operating process
   d) None of the above

3.7 Threads

Thread is a single sequence stream which allows a program to split itself into two or more simultaneously running tasks. It is a basic unit of CPU utilization, and consists of a program counter, a register set and a stack space. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. Threads are not independent of one other like processes. As a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Some of the similarities between processes and Threads are as follows.

- Like processes threads share CPU and only one thread is running at a time.
- Like processes, threads within processes execute sequentially.
- Like processes, thread can create children.
- And like process if one thread is blocked, another thread can run.

Some of the differences between processes and Threads are as follows.

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task.
- Unlike processes, threads are designed to assist one other. (processes might or might not assist one another because processes may originate from different users.)

Following are some reasons why we use threads in designing operating systems.

1) A process with multiple threads makes a great server. For example printer server.
2) Because threads can share common data, they do not need to use interprocess communication.
3) Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that:

1) They only need a stack and storage for registers. Therefore, threads are cheap to create.

2) Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3) Context switching is fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

Advantages of Threads over Multiprocesses

Let's see some of the advantages of threads over multiprocesses:

- ***Context Switching****:* Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- ***Sharing:*** Threads allow the sharing of a lot resources that cannot be shared in process. For example, sharing code section, data section, Operating System resources like open file etc.

Disadvantages of Threads over Multiprocesses

Here are the some of the disadvantages of threads over multiprocesses:

- ***Blocking:*** The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.
- ***Security:*** Since there is an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

Types of Threads

There are two main types of threads:

1) User-Level Threads (ULTs)
2) Kernel-Level Threads (KLTs)

1) User-Level Threads (ULTs):

User-level threads implement in user-level libraries, rather than via systems calls. So thread switching does not need to call operating system and to interrupt the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes. In

this level, all thread management is done by the application by using a thread library.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are:

- User-level threads do not require modification to operating systems.
- Simple Representation:
  Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management:
  This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient:
  Thread switching is not much more expensive than a procedure call.

Disadvantages:

Following points illustrate the disadvantages of ULTs:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads require non-blocking systems call i.e., a multi-threaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

2) Kernel-Level Threads (KLTs):

Kernel Threads (Threads in Windows) are threads supplied by the kernel. All thread management is done by kernel. No thread library exists. The kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

Here are some of the advantages of KLTs:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.
- kernel routines can be multi-threaded

Disadvantages:

The disadvantages of KLTs are as follows:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result, there is significant overhead and increase in kernel complexity.

Combined ULT/KLT Approaches:

Here the idea is to combine the best of both approaches

Solaris is an example of an OS that combines both ULT and KLT

Here are the features of combined ULT/KLT approaches:

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

Self Assessment Questions

10. Thread is a single sequence stream which allows a program to split itself into two or more simultaneously running tasks. (True / False)

11. As threads have some of the properties of processes, they are sometimes called _____ processes.
12. No thread library exists in _____ threads.
    a) User-Level
    b) Kernel-Level
    c) Either a) or b)
    d) None of the above

3.8 Summary

Dear student, let's summarize the important concepts:

- A process is a program in execution. As process executes, it changes it state. Each process may be in one of the following states: New, Ready, Running, Waiting, or Halted.
- The process in the system can execute concurrently. There are several reasons for allowing concurrent execution; information sharing, computation speedup, modularity, and convenience. The processes executing in the operating system may be either independent processes or co-operating processes. Co-operating processes must have the means to communicate with each other.
- Co-operating processes that directly share a logical address space can be implemented as lightweight processes or threads. A thread is a basic unit of CPU utilization, and it shares with peer threads its code section, data section, and operating system resources, collectively known as task.

3.9 Terminal Questions

1. Explain with diagram all possible states a process visits during the course of its execution.
2. What is PCB? What is the useful information available in PCB?
3. Discuss different types of schedulers.
4. Explain different operations on process.
5. What are Co-operating processes? What are the advantages of process co-operation?

3.10 Answers

Self Assessment Questions

1. False
2. Process Control Block (PCB)
3. a) Ready
4. True
5. Scheduler
6. b) Context Switch
7. False
8. Exit System Call
9. c) Co-operating Process
10. True
11. Lightweight
12. d) Kernel-level

Terminal Questions

1. A process can be any one of the following states:
   a) New: Process being created.
   b) Running: Instructions being executed.
   c) Waiting (Blocked): Process waiting for an event to occur.
   d) Ready: Process waiting for CPU.
   e) Terminated: Process has finished execution. (Refer Section 3.2)

2. Every process has a number and a process control block (PCB) represents a process in an operating system. The PCB serves as a repository of information about a process and varies from process to process. The PCB contains information that makes the process an active entity. (Refer Section 3.3)

3. The operating system scheduler schedules processes from the ready queue for execution by the CPU. The scheduler selects one of the many processes from the ready queue based on certain criteria. Schedulers could be any one of the following:
   - Long-term scheduler
   - Short-term scheduler
   - Medium-term scheduler (Refer Section 3.4)

4. The processes in the system can execute concurrently, and must be created and deleted dynamically. Thus the operating system must provide a mechanism for process creation and termination. (Refer Section 3.5)
5. A process is co-operating if it can affect or be affected by the other processes executing in the system. Clearly any process that shares data with other processes is a co-operating process. (Refer Section 3.6)