**5132 – OPERATING SYSTEM**

**MODULE 2**

**CO2 Describe Process & Process Management Contents**

Define process - process control block (PCB) and its general structure - different states of a process with the help of state diagram.- Schedulers – long, medium and short term difference between preemptive and non-preemptive scheduling- Various scheduling criteria- FCFS, SJF, Priority, and RR scheduling algorithms with Gantt charts –comparison of various scheduling algorithms-Resource allocation graph-deadlock-its causes-Process synchronization and critical section management.

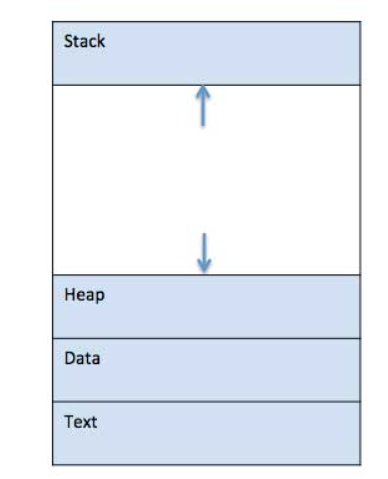
**Process**

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −



**Stack**

The process Stack contains the temporary data such as method/function parameters, return address and local variables.

**Heap**

This is dynamically allocated memory to a process during its run time.

**Text**

This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

**Data**

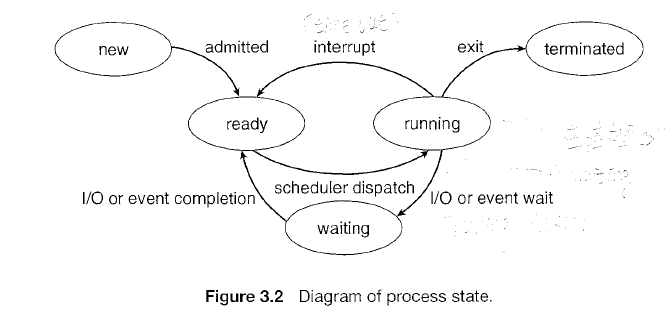
This section contains the global and static variables.

**Process Life Cycle**

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

|  |  |
| --- | --- |
| **S.N.** | **State & Description** |
| 1 | **Start or New**  This is the initial state when a process is first started/created. |
| 2 | **Ready**  The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process. |
| 3 | **Running**  Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions. |
| 4 | **Waiting**  Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available. |
| 5 | **Terminated or Exit**  Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory. |

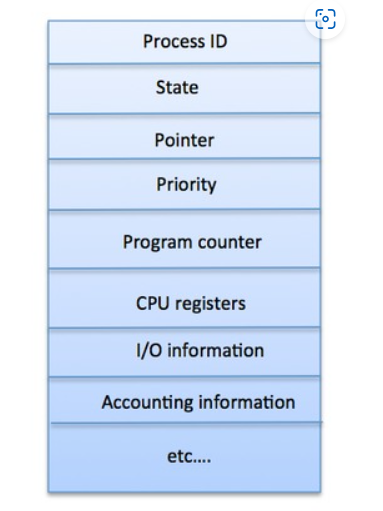


**Process Control Block (PCB)**

A Process Control Block or Task Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

|  |  |
| --- | --- |
| **S.N.** | **Information & Description** |
| 1 | **Process State**  The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**  This is required to allow/disallow access to system resources. |
| 3 | **Process ID**  Unique identification for each of the process in the operating system. |
| 4 | **Pointer**  A pointer to parent process. |
| 5 | **Program Counter**  Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**  Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**  Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**  This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**  This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**  This includes a list of I/O devices allocated to the process. |

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

# Operating System - Process Scheduling

## Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing. The process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

## Process Scheduling Queues

The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

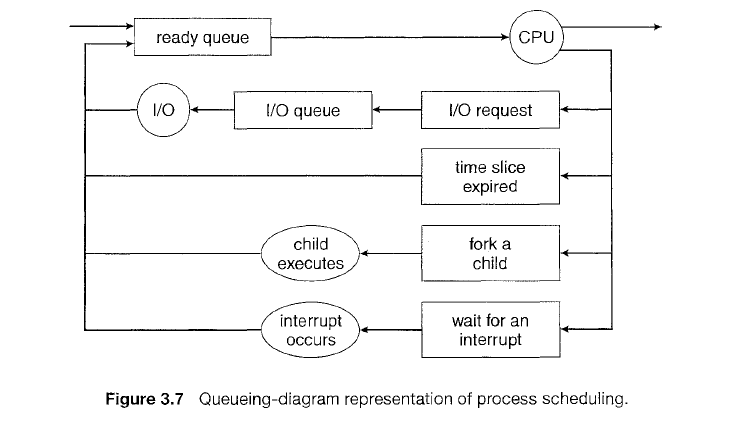
The Operating System maintains the following important process scheduling queues −

* **Job queue** − This queue keeps all the processes in the system.
* **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues or I/O queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

* The process could issue an I/0 request and then be placed in an I/0 queue. The process could create a new sub-process and wait for the sub-process's termination.
* The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources de-allocated.

****

## Categories of Scheduling

There are two categories of scheduling:

1. **Non-preemptive:** Here the resource can’t be taken from a process until the process completes execution. The switching of resources occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

**Two-State Process Model**

Two-state process model refers to running and non-running states which are described below −

|  |  |
| --- | --- |
| **S.N.** | **State & Description** |
| 1 | **Running**  When a new process is created, it enters into the system as in the running state. |
| 2 | **Not Running**  Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute. |

**Schedulers**

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler

**Long Term Scheduler**

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

Most processes can be described as either **I/ 0 bound or CPU bound**. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/0 requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/0 bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/0 waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

**Short Term Scheduler**

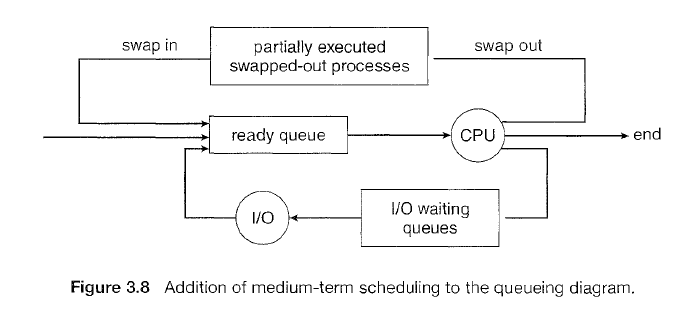
It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

**Medium Term Scheduler**

Some operating systems such as time-sharing systems may use an intermediate level of scheduling called medium-term scheduling. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramrning. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out.



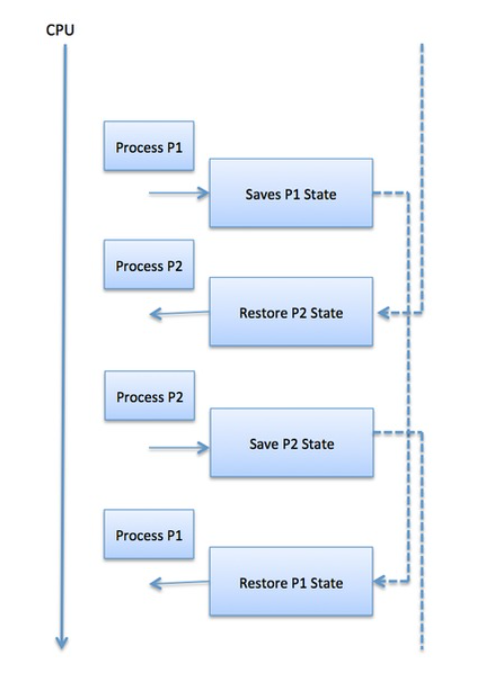
**Comparison among Scheduler**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Long-Term Scheduler** | **Short-Term Scheduler** | **Medium-Term Scheduler** |
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

**Context Switching**

A context switching is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

When the process is switched, the following information is stored for later use.

* Program Counter
* Scheduling information
* Base and limit register value
* Currently used register
* Changed State
* I/O State information
* Accounting information

# CPU Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/0 request ).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, at completion of I/0).
4. When a process terminates.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive. Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.

**Scheduling Criteria**

There are six popular process scheduling algorithms:

* First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-Next (SJN) Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin(RR) Scheduling
* Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include the following:

**CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

**Throughput**. The number of processes that are completed per time unit, called *throughput.* **Turnaround time.** The interval from the time of submission of a process to the time of completion is the *turnaround time.*

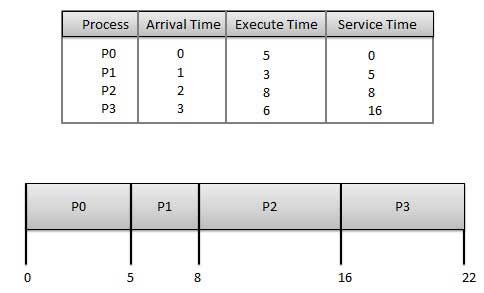
**Waiting time**. *Waiting time* is the sum of the periods spent waiting in the ready queue.

**Response time**. It is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minirnize turnaround time, waiting time, and response time.

**First Come First Serve (FCFS)**

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive scheduling algorithm. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.



The above bar chart is called **Gantt chart**, which illustrates a particular schedule, including the start and finish times of each of the participating processes:

**Wait time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Wait Time : Service Time - Arrival Time** |
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 8 - 2 = 6 |
| P3 | 16 - 3 = 13 |

Average Wait Time: (0+4+6+13) / 4 = 5.75

There is a **convoy effect** in FCFS as all the other processes wait for the one big process to get off the CPU.

**Example:** Find the average waiting time for processes p1, p2 and p3 with burst times (execute time) 24, 3 and 3 respectively, if they arrive in the order p1 p2 p3. Also find the average waiting time if they arrive in the order p2 p3 p1.

**Shortest Job Next (SJN)**

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive scheduling algorithm.
* Best approach to minimize waiting time.
* The processer should know in advance how much time process will take.

Given: Table of processes and their Execution time(Burst time)

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P0 | 5 |
| P1 | 3 |
| P2 | 8 |
| P3 | 6 |

The Gantt chart is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P0 | P3 | P2 |
|  |  |  |  |

0 3 8 14 22

**Waiting time** of each process is as follows −

Average Wait Time: (0 + 3 + 8 + 14)/4 = 25 / 4 = 6.25

Example: consider the following set of processes, with the length of the CPU burst given in milliseconds:

**Process Burst Time**

Pl6

P2 8

P3 7

P4 3

Schedule the above processes Using SJF scheduling, draw Gantt chart and find average waiting time.

The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes. The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

A preemptive SJF algorithmwill preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. **Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.**

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process Arrival Time Burst Time

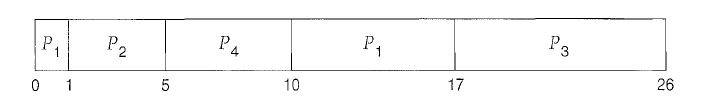
Pl 0 8

P2 1 4

P3 2 9

P4 3 5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in the queue. Process *P2* arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process *P2* (4 milliseconds), so process P1 is preempted, and process *P2* is scheduled. The average waiting time for this example is [(10- 1) + (1 - 1) + (17- 2) +(5-3)]/ 4 = 26/4 = 6.5 milliseconds.

Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

**Priority Based Scheduling**

* Priority scheduling can be either preemptive or non-preemptive.
* Each process is assigned a priority. Process with highest priority is to be executed first and so on.
* Processes with same priority are executed on first come first served basis.
* Priority can be decided based on memory requirements, time requirements or any other resource requirement.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, *P2,* · · ·, *P5,* with the length of the CPU burst given in milliseconds:

Process Burst Time Priority

Pl 10 3

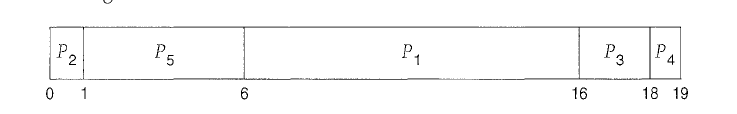
P2 1 1

P3 2 4

P4 1 5

P5 5 2

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



The average waiting time is 8.2 milliseconds.

When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

A major problem with priority scheduling algorithms is indefinite blocking, or **starvation**. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

**Round Robin Scheduling**

* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a **quantum**.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the **tail** of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

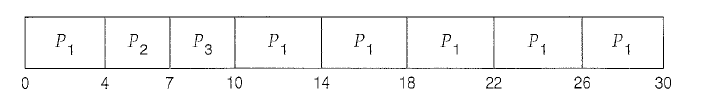
Process Burst Time

P1 24

P2 3

P3 3

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 .* Process *P2* does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process *P3.* Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



P1 waits for 6 millisconds (10- 4), *P2* waits for 4 millisconds, and *P3* waits for 7 millisconds.

Thus, the average waiting time is 17/3 = 5.66 milliseconds.

**Example**

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process Burst Time Priority

P1 10 3

P2 1 1

P3 2 3

P4 1 4

P5 5 2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum= 1).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

**DEADLOCKS**

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called

a deadlock.

**Necessary Conditions for a deadlock**

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

**Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

**Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

**No** **preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

**Circular wait.** A set { P0 , Pl, ... , Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

**Resource-Allocation Graph**

Deadlocks can be described more precisely in terms of a directed graph called a **system resource allocation graph**. This graph consists of a set of vertices *V* and a set of edges *E.* The set of vertices *V* is partitioned into two different types of nodes: *P* = { P1, P2, ... , Pn}, the set consisting of all the active processes in the system, and *R* = {R1, R2, ... , *Rm},* the set consisting of all resource types in the system.

A directed edge from process Pi to resource type *Rj* is denoted by Pi 🡪Rj ;it signifies that process Pihas requested an instance of resource type *Rj* and is currently waiting for that resource. A directed edge from resource type *Rj* to process Piis denoted byRj 🡪 Pi;it signifies that an instance of resource type Rjhas been allocated to process Pi*.* A directed edge *Pi 🡪* *Rj* is called a request edge; a directed edge *Rj🡪* *Pi* is called an assignment edge.

Pictorially we represent each process *Pi* as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *Ri,* whereas an assignment edge must also designate one of the dots in the rectangle.

When process *Pi* requests an instance of resource type *Rj,* a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

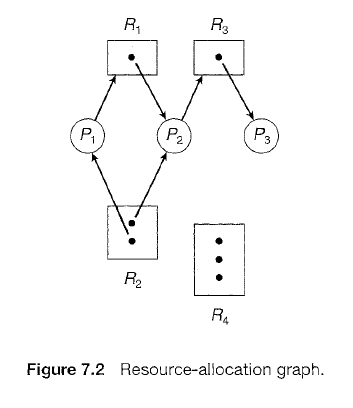
The resource-allocation graph shown in Figure 7.2 depicts the following situation.

The sets *P,* R and *E:*

* *P* = {P1, P2, P3}
* R= {R1, R2, R3, R4}
* *E* ={Pl🡪 Rl, p2🡪 *R3,* Rl🡪P2, R2🡪 P2, R2🡪 Pl, R3🡪 P3}

Resource instances:

* One instance of resource type R1
* Two instances of resource type R2
* One instance of resource type R3
* Three instances of resource type R4



Process states:

* Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .
* Process P2 is holding an instance of R1 and an instance of R2 and iswaiting for an instance of R3.
* Process *P3* is holding an instance of R3 .

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

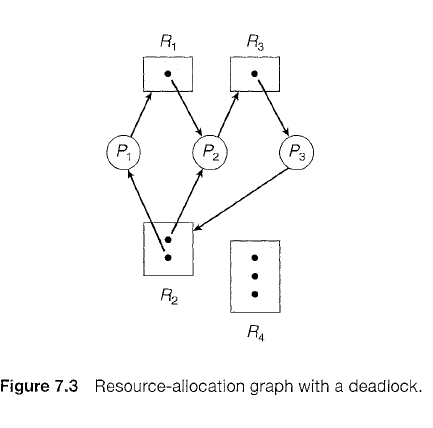
If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 7.2. Suppose that process *P3* requests an instance of resource type R2. Since no resource instance is currently available, a request edge *P3* 🡪 R2 is added to the graph (Figure 7.3). At this point, two minimal cycles exist in the system:

P1 🡪 *R* 1 🡪 P2 🡪 R3 🡪 *P3* 🡪 R2 🡪 P1

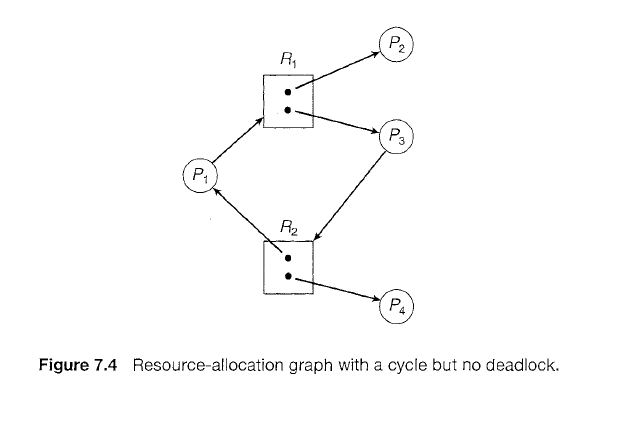
P2 🡪 R3 🡪 *P3* 🡪 R2 🡪 P2



Processes *P1, P2,* and *P3* are deadlocked. Process *P2* is waiting for the resource *R3,* which is held by process *P3.* Process *P3* is waiting for either process *P1* or process *P2* to release resource R2. In addition, process *P1* is waiting for process *P2* to release resource R1.

Now consider the resource-allocation graph in Figure 7.4. In this example, we also have a cycle:

P1🡪R1🡪P3🡪R2🡪P1



However, there is no deadlock. Observe that process *P4* may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

**PROCESS SYNCHRONIZATION**

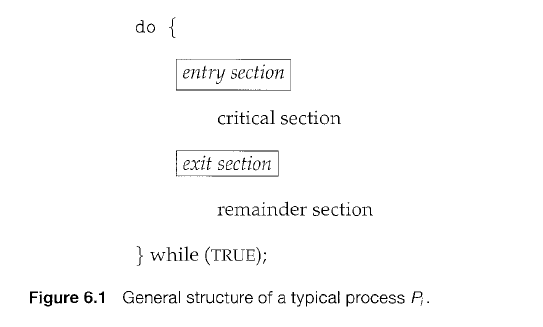
A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency. Process synchronization techniques use various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Solutions of the **critical-section problem**, can be used to ensure the consistency of shared data.

**The critical-section problem**

Consider a system consisting of *n* processes {P0, P1 , ... , *Pn-1*}. Each process has a segment of code, called a ***critical section*** in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the *entry section*. The critical section may be followed by an *exit section*. The remaining code is the *remainder section*. The general structure of a typical process *Pi* is shown in Figure 6.1.



The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.