Unit 2 : The Process

A process is a program in execution. A process is different from a program. A program is a passive entity whereas a process is an active entity. A program is also known as the text section. A process includes the current activity, as represented by the value of the program counter and the contents of the processor's register. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, return address, and local variables), and a data section, which contains global variables.

PROCESS STATE

The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

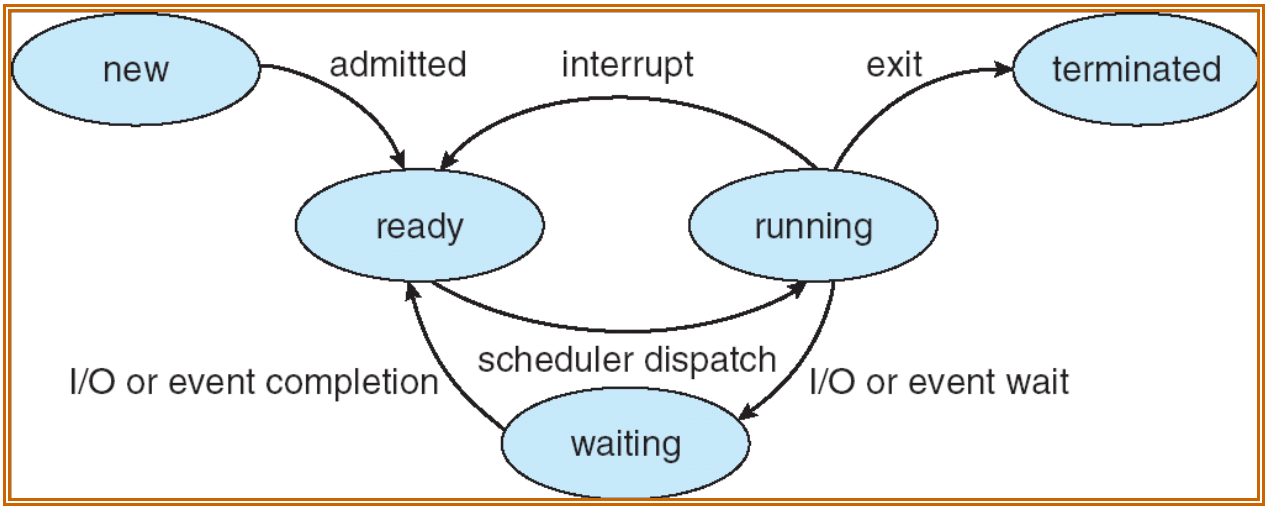


Diagram of Process State

- **New:**

  The process is being created.
- **Ready:**

  The process is waiting to be allocated to a processor. Processor comes to this state immediately after it is created. All ready processes (kept in queue) keeps waiting for CPU time to be allocated by operating system in order to run. A program called **scheduler** which is a part of operating system, pick-ups one ready process for execution by passing a control to it.
- **Running:**

Instructions are being executed. When a process gets a control from CPU plus other resources, it starts executing. The running process may require some I/O during execution. Depending on the particular scheduling policy of operating system, it may pass its control back to that process or it (operating system) may schedule another process if one is ready to run.

- **Waiting:**

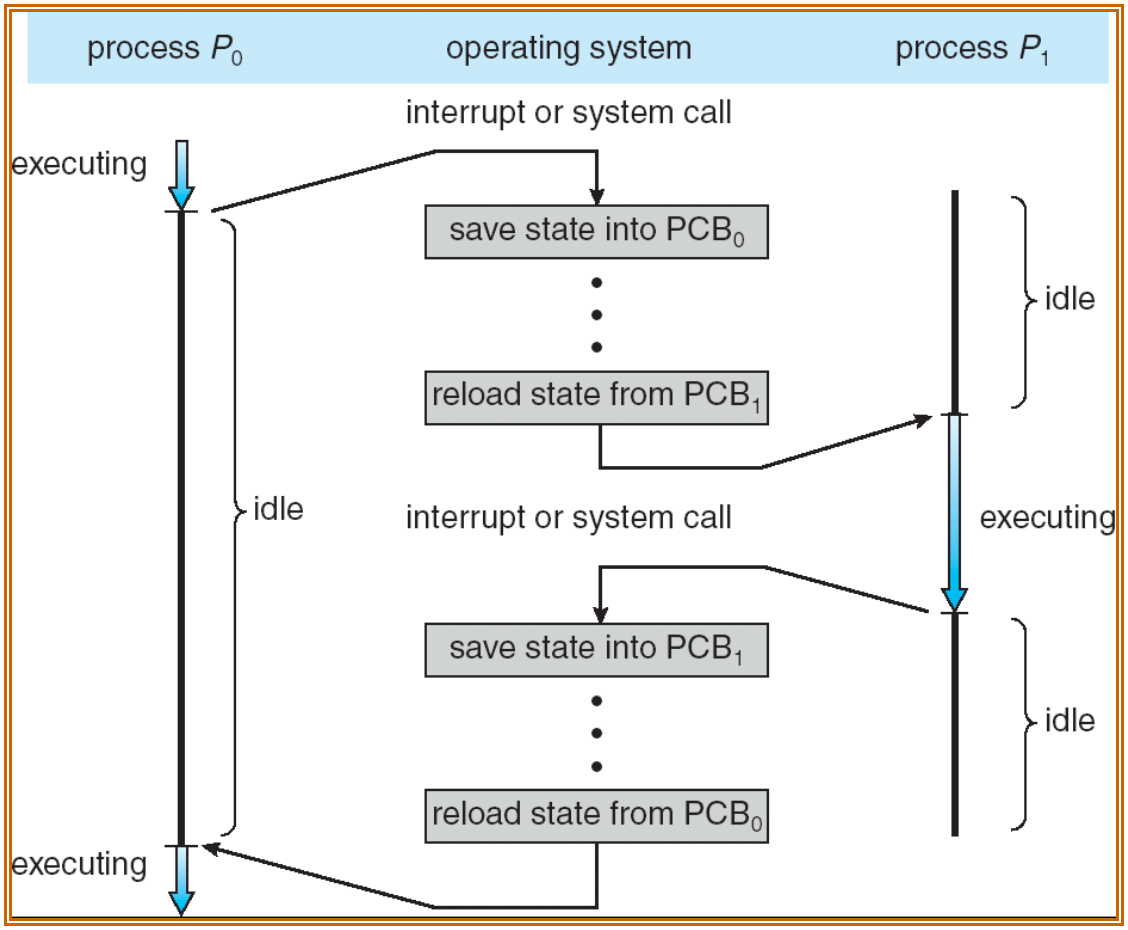
  A suspended process lacks some resource other than the CPU. Such processes are normally not considered for execution until the related suspending conditions is fulfilled. The running process become suspended by invoking I/O routines whose result it needs in order to proceed.

- **Terminated:**

  When the process finally stops. A process terminates when it finishes executing its last statement. At that point in time, the process may return some data to its parent process. Sometimes there are additional circumstances when termination occurs. A process can cause the termination of another process via an appropriate system call.

PROCESS CONTROL BLOCK (PCB)

The operating system groups all information that it needs about a particular process into a data structure called a Process Control Block (PCB). It simply serves as the storage for any information for processes. When a process is created, the operating system creates a corresponding PCB and when it terminates, its PCB is released to the pool of free memory locations from which new PCBs are drawn. A process is eligible to compete for system resources only when it has an active PCB associated with it. A PCB is implemented as a record containing many pieces of information associated with a specific process, including:



- **Process state:**

  The state may be new, ready, running, waiting or terminated.
- **Program counter:**

  The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:**

  They include accumulator, general purpose registers, index registers etc. Whenever a processor switches over from one process to another process, information about current status of the old process is saved in the register along with the program counter so that the process be allowed to continue correctly afterwards.
- **CPU-scheduling information:**

  This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

- **Memory-management Information:**
  This information may include information as 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 this process, a list of open files and so on.

PROCESSOR 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 utilization 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. This mechanism improves the overall efficiency of the computer system by getting more work done in less time.

Types of Schedulers

- **Long term schedulers**
- **Medium term schedulers**
- **Short term schedulers**

Long-term schedulers:

This is also called job scheduler. This determines which job shall be admitted for immediate processing. There are always more processes than it can be executed by the CPU. These processes are kept in large storage devices like disk for later processing. The long term scheduler selects processes from this pool and loads them into memory. In memory these processes belong to a ready queue. The short term scheduler (also called the CPU scheduler) selects from among the processes in memory which are ready to execute and assigns the CPU to one of them. The long term scheduler executes less frequently. The long-term scheduler 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 longer time taken by CPU during execution, the long term scheduler can afford to take more time to decide which process should be selected for execution.

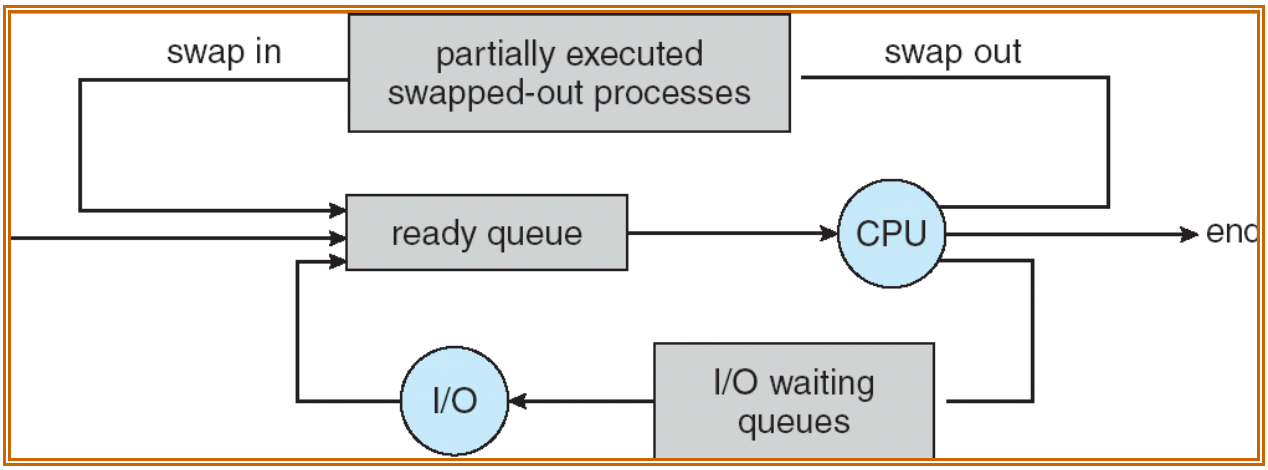
It may also be very important that the long-term scheduler should take a careful selection of processes, i.e. processes should be a combination of CPU and I/O bound types. Generally, most processes can be put into any of the two categories: CPU bound or I/O bound. If all processes are I/O bound, the ready queue will always be empty and the short-term scheduler will have nothing to do. If all processes are CPU bound, no process will be waiting for I/O operation and again the system will be unbalanced. Therefore, the long-term scheduler provides good performance by selecting a combination of CPU bound and I/O bound processes.

Medium-term schedulers:

Most of the processes require some I/O operation. In that case, it may become suspended for I/O operation after running a while. It is beneficial to remove these process (suspended) from main memory to hard disk to make room for other processes. At some later time these process can be reloaded into memory and

continued where from it was left earlier. Saving of the suspended process is said to be swapped out or rolled out. The process is swapped in and swapped out by the medium term scheduler.

The medium term scheduler has nothing to do with the suspended processes. But the moment the suspending condition is fulfilled, the medium term scheduler get activated to allocate the memory and swapped in the process and make it ready for execution. The medium-term scheduler is also helpful for reducing the degree of multiprogramming, when the long term-term scheduler is absent or minimal.



(fig: long-term scheduler, medium-term scheduler, short-term scheduler)

Short-term scheduler:

This is also called CPU scheduler. When ever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler. It allocates processes belonging to ready queue to CPU for immediate processing. It's main objective is to maximize CPU utilization. Compared to the other two schedulers this is more frequent. It must select a new process for execution quite often because a CPU executes a process only for few milliseconds before it goes for I/O operation. Often the short term scheduler executes at least once in every 10 milliseconds. If it takes 1 millisecond to decide to execute a process for 10 milliseconds, the 1/(10+1) = 9% of the CPU is being wasted simply for scheduling the work. Therefore. it must be very fast.

Scheduling and Performance Criteria:

Different CPU-scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. The following criteria have been suggested for comparing CPU-scheduling algorithms.

- **CPU utilization**
- **Throughput:**
- **Turnaround time**
- **Waiting time**
- **Response time**

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 may range from 0 to 100 percent.

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 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 the process with similar requirements.

Turnaround time:

From the point of view of a particular process, the important criterion is how long it takes to execute that process. Turnaround time may be defined as the 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, executing on the CPU and I/O operations.

Waiting time:

In a 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 the actual processing time, i.e. 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 (the CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O). Thus rather than looking at turnaround time waiting time is the sum of the periods spent waiting in the ready queue.

Response time:

It is most frequently considered in time sharing and real time operating systems. However it's characteristics differs 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 signaled to the time the first instruction of the respective service routine is executed.

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 utilization. Throughput and CPU utilization may be increased by executing large number of processes, but then response time may suffer. Therefore, the design of a scheduler usually requires a balance of all the different requirements and constraints.

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are several scheduling algorithms exists. A major division among scheduling algorithms is that whether they support **pre-emptive or non-preemptive scheduling** discipline. A scheduling discipline is non-preemptive if once a process has been given the CPU, the CPU cannot be taken away from that process. A scheduling discipline is pre-emptive if the CPU can be taken away. Preemptive scheduling is more useful in high priority process which requires immediate response. In non-preemptive systems, jobs are made to wait by longer jobs, but the treatment of all processes is fairer. 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.

First-Come-First-Served Scheduling:

This is one of the simplest scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

A FCFS scheduling is **non preemptive** which usually results in poor performance. As a consequence of **non preemption**, there is a low rate of component utilization and system throughput. Short jobs may suffer considerable turnaround delays and waiting times when CPU has been allocated to longer jobs.

Consider the following two processes:

Process	Execution time
P1	20
P2	4

0 20 24

If both processes P1 and P2 arrive in order P1 -P2 in quick succession, the turnaround times are 20 and 24 units of time respectively (P2 must wait for P1 to complete) thus giving an average of 22 (20+24)/2) units of time. The corresponding waiting times are 0 and 20 units of time with average of 1 0 time units.

0 4 24

However, when the same processes arrive in P2-P 1 order, the turn around times are 4 and 24 units of time respectively giving an average of 14 (4+24)/2) units of time and the average waiting time is(0+4)/2)=2. This is a substantial reduction. This shows how short jobs may be delayed in FCFS scheduling algorithm.

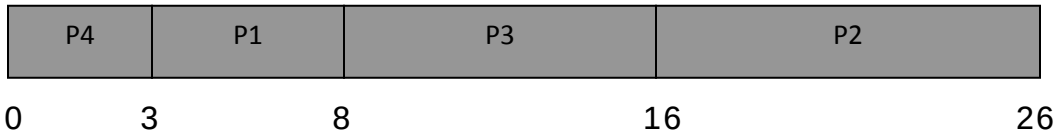
The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is particularly troublesome for time-sharing, where each user needs to get a share of the CPU at regular intervals.

Shortest-Job-First Scheduling:

A different approach to CPU scheduling is the Shortest-Job-First where the scheduling of a job or a process is done on the basis of its having shortest next CPU burst (execution time). When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Consider the following set processes:

Process	Execution time
P1	5
P2	10
P3	8
P4	3



Using SJF scheduling, these processes would be scheduled in the P4 - P1 - P3 - P2 order. Waiting time is 0 + 3 + 8 + 16 / 4 = 27/4 = 6.75 units of time. If we were using the FCFS scheduling then the average waiting time will be = (0 + 5 + 15 + 23)/4 = 43/4 = 10.75 units of time.

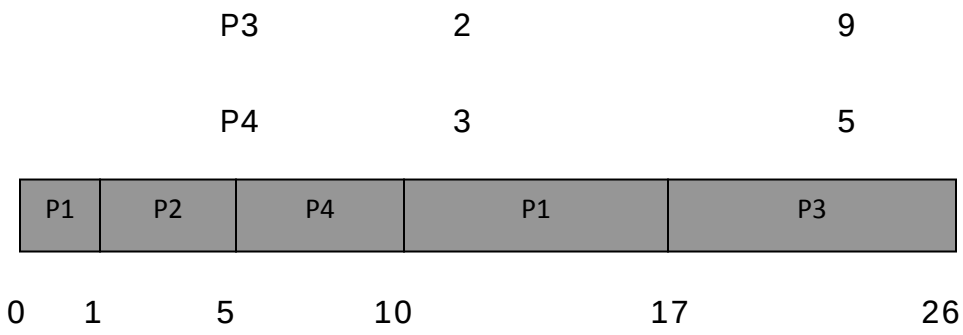
The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of shot-term CPU scheduling. There is no way to know the length of the next CPU burst. SJF scheduling is used frequently in long-term scheduling. For long-term scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.

SJF may be implemented in either non-preemptive or preemptive varieties. When a new process arrives at the ready queue with a shorter next CPU burst than what is left of the currently executing process, then a preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also called **shortest-remaining-time-first** scheduling.

Consider the following processes:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4



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 is larger than the time required by P2, so P1 is preempted, and P2 is scheduled. The average waiting time for this example is

((10 – 1) + (1 – 1) + (17 – 2) + (5 – 3)) / 4 = 26 / 4 = 6.5. But a non-preemptive SJF scheduling would result in an average waiting time of 7.75.

Priority 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 in FCFS order. An SJF algorithm is simply a priority algorithm where the priority is the inverse of the next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. The level of priority may be determined on the basis of resource requirements, processes characteristics and its run time behavior.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with 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 non-preemptive priority-scheduling algorithm will simply put the new process at the head of the ready queue.

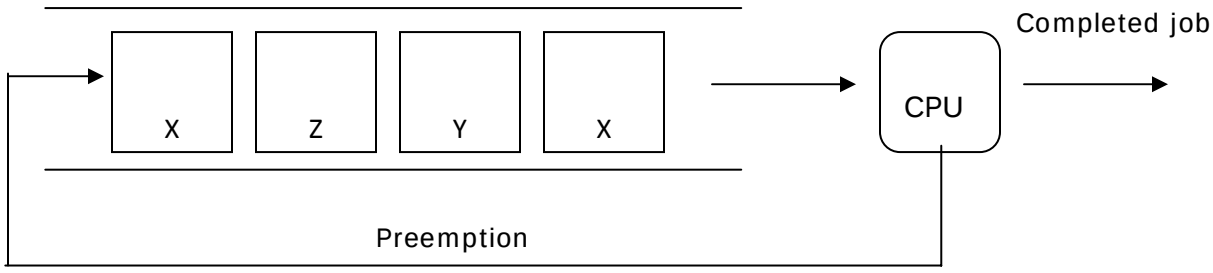
A major problem with priority-scheduling algorithms in **indefinite blocking** (or **starvation**). A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from

ever getting the CPU. 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 processes is aging. Aging is a technique of gradually increasing the priority of processes 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.

Round-Robin Scheduling (RR):

This is one of the oldest, simplest and widely used algorithm. The round-robin scheduling algorithm is primarily used in a time-sharing and a multi-user system where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** (or **time slice**), is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

Each process is allocated a small time-slice (from 10-100 millisecond) while it is running. No process can run for more than one time slice when there are others waiting in the ready queue. If a process needs more CPU time to complete after exhausting one time slice, it goes to the end of ready queue to await the next allocation. Otherwise, if the running process releases a control to operating system voluntarily due to I/O request or termination, another process is scheduled to run.



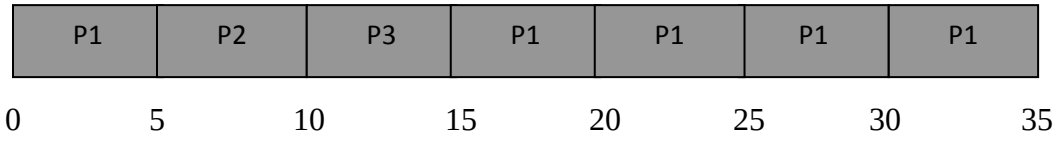
(Fig: Round Robin Scheduling)

Round robin scheduling utilizes the system resources in an equitable manner. Small process may be executed in a single time-slice giving good response time whereas long processes may require several time slices and thus be forced to pass through ready queue a few times before completion.

Consider the following processes:

Process	Burst time/Execution time
P1	25
P2	5
P3	5

If we use a time-slice of 5 units of time, then P1 gets the first 5 units of time. Since it requires another 20 units of time, it is pre-empted after the first time slice and the CPU is given to the next process i.e. P2. Since P2 just needs 5 units of time, it terminates as time-slice expires. The CPU is then given to the next process P3. Once each process has received one time slice, the CPU is returned to P1 for an additional time-slice. Thus the resulting round robin schedule is:



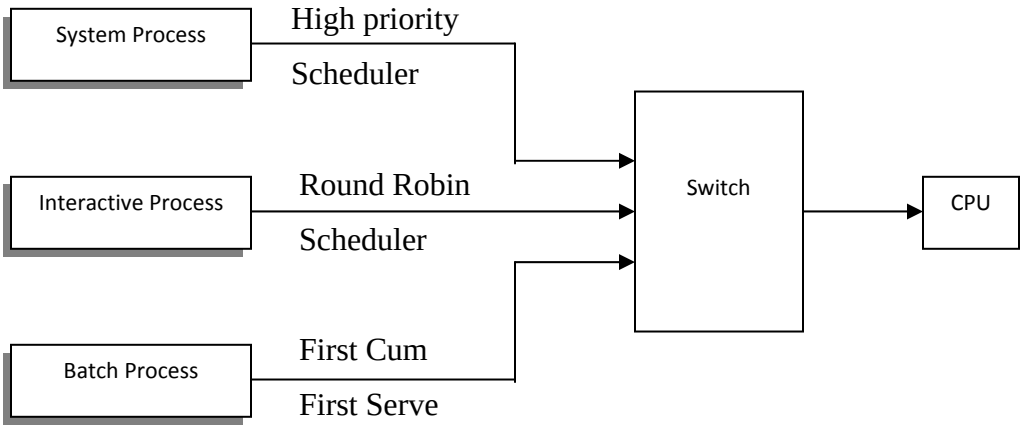
Implementation of round robin scheduling requires the support of a dedicated timer. The timer is usually set to interrupt the operating system wherever a time slice expires and thus force the scheduler to be involved. Processing the interrupt to switch the CPU to another process requires saving all the registers for the old process and then loading the registers for the new process. This task is known as **context switching**. Whenever the running process surrenders control to the operating system before expiration of its time-slice, the scheduler is again invoked to dispatch a new process to CPU.

Multiple-Level-Queue (MLQ) Scheduling:

In MLQ, processes are classified into different groups. For example, interactive processes (foreground) and batch processes (background) could be considered as two types of processes because of their different response time requirements, scheduling needs and priorities.

A multi-level-queue scheduling algorithm partitions the ready queue into several separate queues. Processes are permanently assigned to each queue, usually based upon properties such as memory size or process type. Each queue has its own scheduling algorithm. The interactive queue might be scheduling by a round-robin algorithm while batch queue follows FCFS.

As an example of multiple-level-queues scheduling one simple approach to partitioning of the ready queue into system processes, interactive processes and batch processes which creates a three ready queues.



(fig : **Multiple-Level-Queue Scheduling**)

In addition, there must be scheduling among the queues. There are different possibilities to manage queues. One possibility is to assign a time slice to each queue, which it can schedule among different processes in its queue. Foreground

processes can be assigned 80% of CPU whereas background processes are given 20% of the CPU time.

The second possibility is to execute the high priority queue first. For example, no process in the batch queue could run unless the queue for system processes and interactive processes were all empty. If an interactive process entered the ready queue while a batch process is running, the batch process would be pre-empted.

Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other process is a cooperating process.

Reasons for providing an environment that allows process cooperation:

- **Information Sharing**: Since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to such information.
- **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 or threads.
- **Convenience**: Even an individual user may work on may tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. Inter Process Communication is a capability supported by operating system that allows one process to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another.

- Critical Resource

  It is a resource shared with constraints on its use(e.g., memory, files, printers, etc)

- Critical Section

  It is code that accesses a critical resource.

- Mutual Exclusion

  At most one process may be executing a critical section with respect to a particular critical resource simultaneously.

There are two fundamental models of interprocess communication:

- **Shared Memory**

  In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

- **Message Passing**

  In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

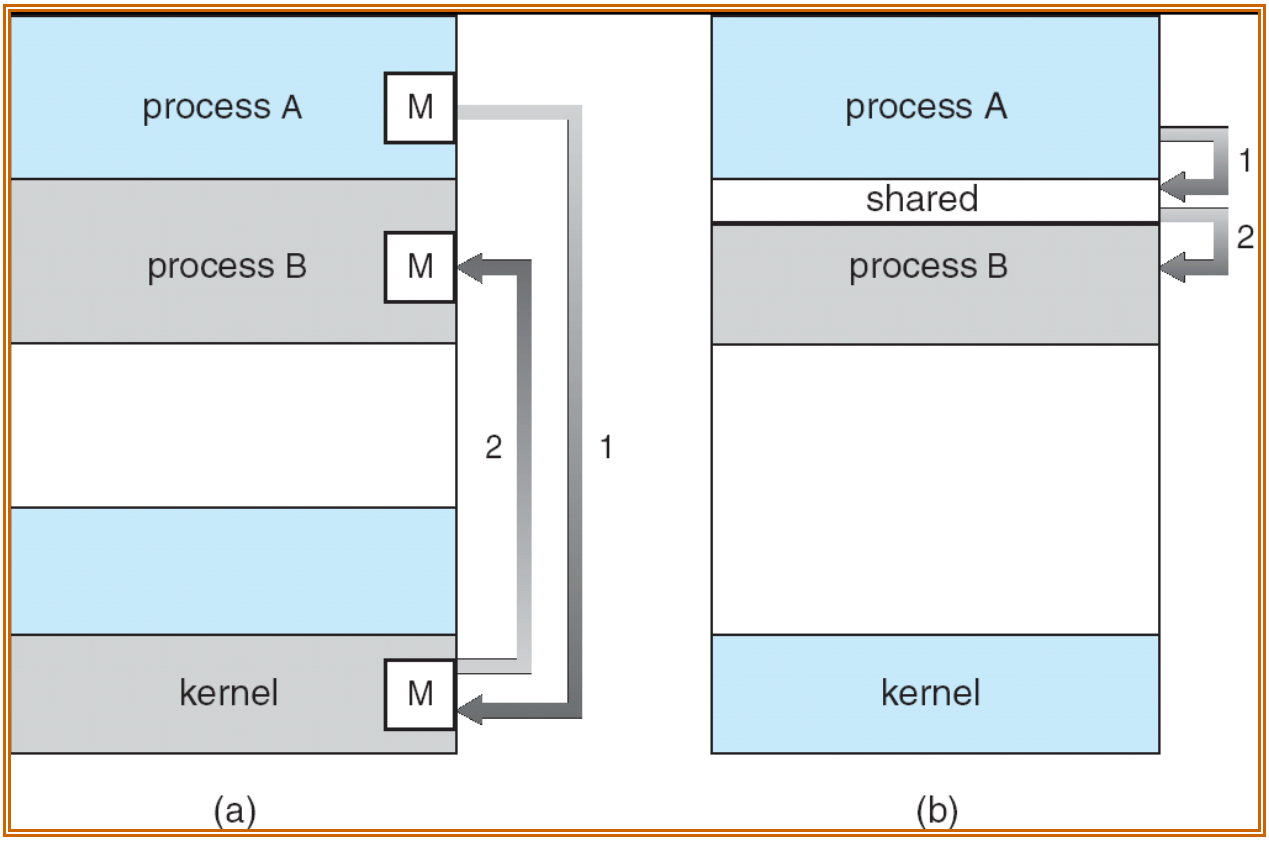


Fig: Communication models. (a) Message passing. (b) Shared memory.

Message passing is useful for exchanging smaller amounts of data. Message passing is also easier to implement than shared memory for inter-computer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Shared Memory System requires communication processes to share some variables. The processes are expected to exchange information through the use of these shared variables.

Responsibility for providing communication rests with the application programmer. The OS only needs to provide shared memory.

Message Passing System allows communication processes to exchange messages – responsibility for providing communication rest with OS.

The function of a MPS is to allow processes to communicate with each other without the need to resort to shared variables.

An Inter Process Communication facility basically provides two operations:

- send(message)
- receive(message)

In order to send and to receive messages, a communication link must exist between the two involved processes.

Methods for logically implementing a communication link and the send/receive operations are classified into:

- Naming - Consisting of direct and indirect communication.
- Buffering - Consisting of capacity and message properties.

Direct Communication

Each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme the send and receive primitives are:

- send(P, message) – send a message to process P
- receive(Q, message) – receive a message from process Q

  Here a link is established automatically between every pair of processes that want to communicate. Exactly one link exists between each pair of processes.

Symmetry in addressing

- send(P, message) – send a message to process P
- receive(Q, message) – receive a message from process Q

This scheme shows the symmetry in addressing; that is both the sender and the receiver processes must name the other to communicate.

Asymmetry in addressing

- send(P, message) – send a message to process P
- receive(id, message) – receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

Only the sender names the recipient; the recipient is not required to name the sender.

Indirect Communication

With indirect communication, the message are sent to and receive from a *mailbox*. It is an object into which messages may be placed and from which messages may be removed by processes. Each mailbox owns a unique identification. A process may communicate with some other process by a number of different mailboxes.

- send(A, message) – send a message to mailbox A
- receive(A, message) – receive a message from mailbox A

Mailbox owned by process:

Mailboxes may be owned by either by a process or by the system.

If the mailbox is owned by a process, then the owner who can only receive from this mailbox and the user who can only send message to the mailbox are to be distinguished. When a process that owns a mailbox terminates, its mailbox disappears.

Mailbox owned by the OS:

It has an existence of its own, i.e., it is independent and not attached to any particular process.

The OS provides a mechanism that allows a process to:

- create a new mailbox
- send and receive message through the mailbox
- destroy a mailbox

Buffering:

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. This queue can be implemented in three ways:

- Zero capacity
- Bounded capacity
- Unbounded capacity

Zero capacity

The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message. The zero-capacity link is referred to as a message-passing system with no buffering.

Bounded capacity

The queue has finite length n; thus, at most n messages can reside in it. If a new message is sent, and the queue is not full, it is placed in the queue either by copying the message or by keeping a pointer to the message and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue.

Unbounded capacity

The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks. Bounded and Unbounded capacity link is referred to as message-passing system with automatic buffering.