UNIT-II
Process, Threads, CPU scheduling

1.Process

A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program).

An active program which running now on the Operating System is known as the process. Basically, a process is a simple program.

The structure of a process in memory is shown in Figure

A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

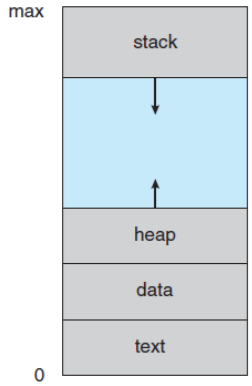


Figure 3.1 Process in memory.

Stack

The process stack stores temporary information such as method or function arguments, the return address, and local variables.

Heap

This is the memory where a process is dynamically allotted while it is running.

Text

This consists of the information stored in the processor's registers as well as the most recent activity indicated by the program counter's value.

Data

In this section, both global and static variables are discussed.

Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

• **New**. The process is being created.

• **Running**. Instructions are being executed.

• **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• **Ready**. The process is waiting to be assigned to a processor.

• **Terminated**. The process has finished execution.

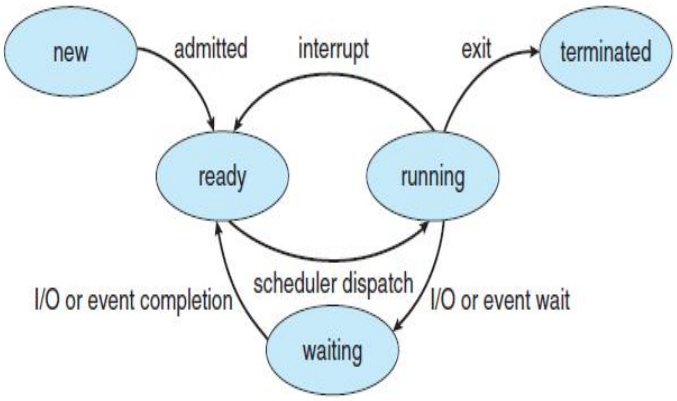


Figure 3.2 Diagram of process state.

Process Control Block

It contains many pieces of information associated with a specific process, including these:

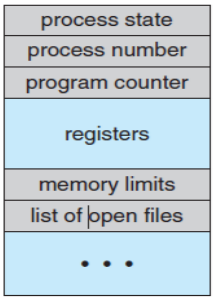


Figure 3.3 Process control block (PCB).

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The 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 afterward

- **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 such items as the value of the base and limit registers and 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**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Thread

- A thread refers to a single sequential flow of activities being executed in a process; it is also known as the thread of execution or the thread of control. Now, thread execution is possible within any OS's process.

2.Process Scheduling

- The act of determining which process is in the ready state, and should be moved to the running state is known as Process Scheduling.
- The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fall into one of the two general categories:

- **Non-Pre-emptive Scheduling**: Non-preemptive scheduling, any new process has to wait until the running process finishes its CPU cycle
- **Pre-emptive Scheduling**: Preemptive scheduling allows a running process to be interrupted by a high priority process.

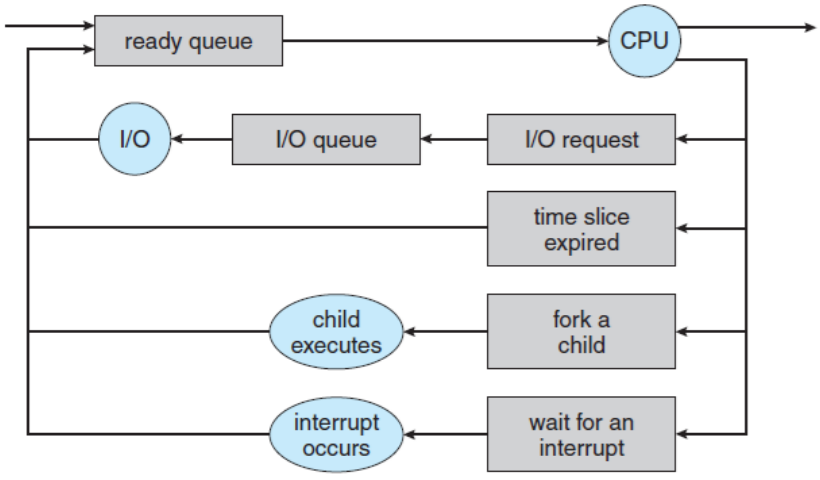


Figure 3.6 Queueing-diagram representation of process scheduling.

Schedulers

The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**

Types of Schedulers

- There are three types of schedulers available:
- 1. Long Term Scheduler
- 2. Short Term Scheduler
- Medium Term Scheduler

Long Term Scheduler

- Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution.
- Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

Short Term Scheduler

- This is also known as CPU Scheduler and runs very frequently.
- The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

- This scheduler removes the processes from memory (and from active contention for the CPU),and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van 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.
- Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is descripted in the below diagram:

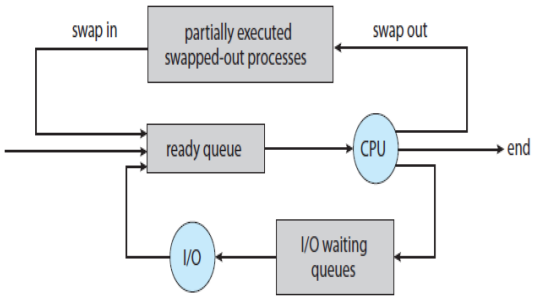
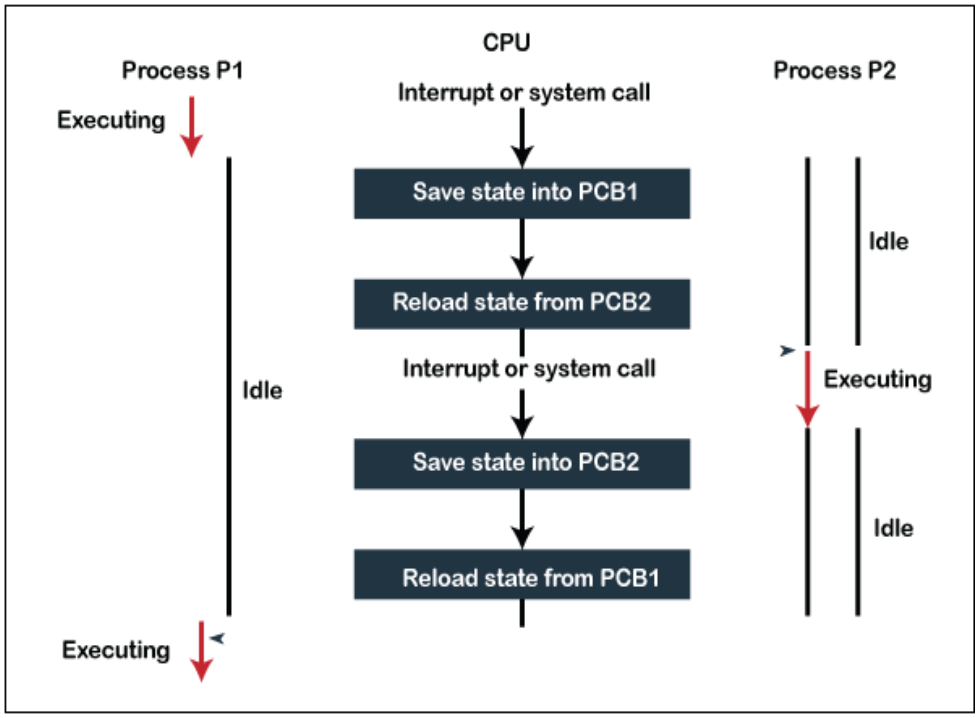


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Context Switch

- 1. Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a Context Switch.
- 2. The context of a process is represented in the Process Control Block(PCB) of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- 3. Context switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions(such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.
- 4. Context Switching has become such a performance bottleneck that programmers are using new structures(threads) to avoid it whenever and wherever possible.



3.Operations on Processes

There are many operations that can be performed on processes. Some of these are process creation, process preemption, process blocking, and process termination. These are given in detail as follows −

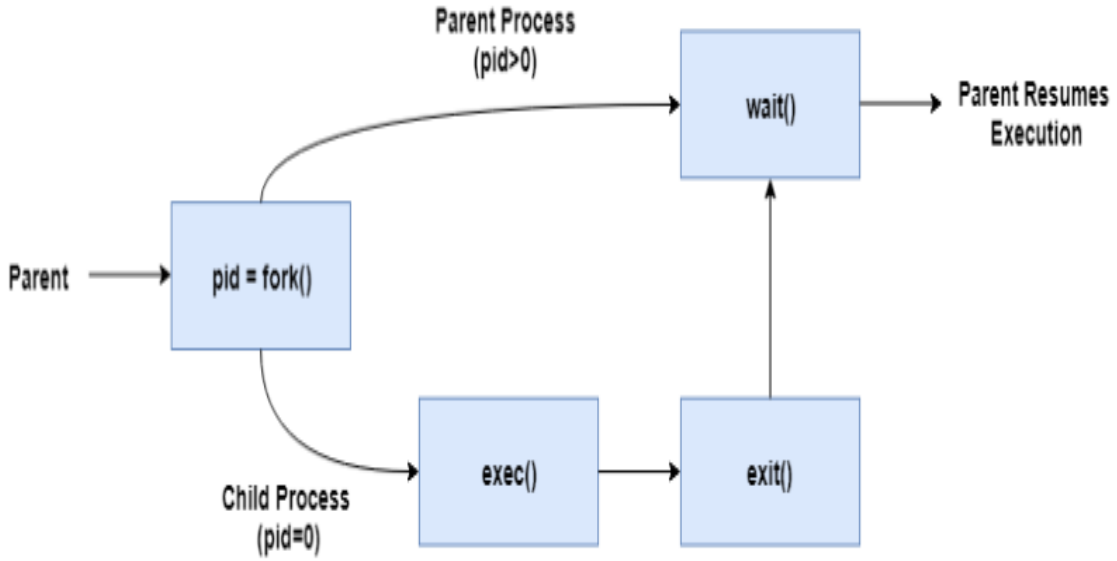
Process Creation

Processes need to be created in the system for different operations. This can be done by the following events −

- User request for process creation

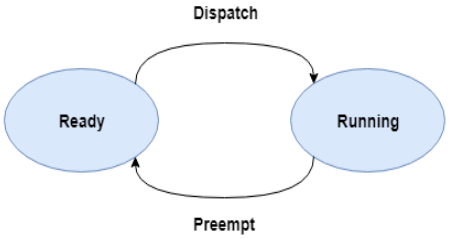
- System initialization
- Execution of a process creation system call by a running process
- Batch job initialization

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.
A diagram that demonstrates process creation using fork() is as follows –



Process Creation Using fork()

Process Preemption
An interrupt mechanism is used in preemption that suspends the process executing currently and
the next process to execute is determined by the short-term scheduler. Preemption makes sure that all processes get some CPU time for execution.
A diagram that demonstrates process preemption is as follows –

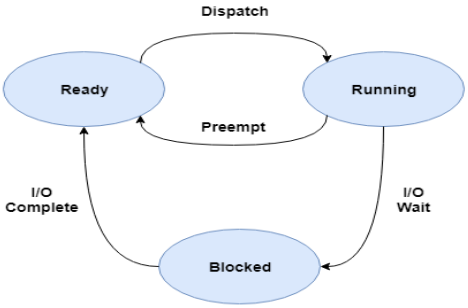


Process Preemption

Process Blocking

The process is blocked if it is waiting for some event to occur. This event may be I/O as the I/O
events are executed in the main memory and don't require the processor. After the event is complete, the process again goes to the ready state.

A diagram that demonstrates process blocking is as follows –



Process Blocking

Process Termination

After the process has completed the execution of its last instruction, it is terminated. The resources held by a process are released after it is terminated.

A child process can be terminated by its parent process if its task is no longer relevant. The child
process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated.

4.Inter Process Communication (IPC)

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a cooperating
process can be affected by other executing processes. Though one can think that
those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.

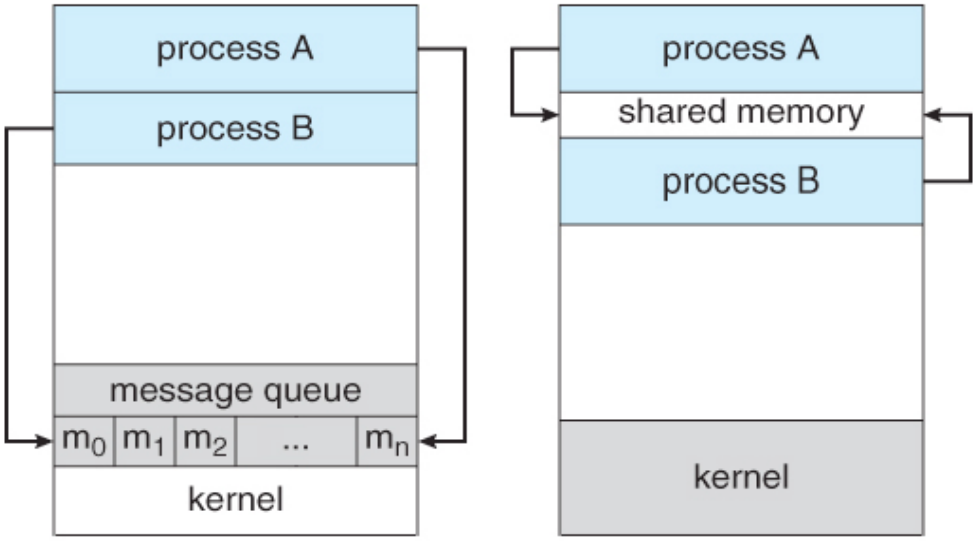
Basics of Inter Process Communication:

- **Information sharing:** Since some users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to that information.
- **Computation speedup:** If you want a particular work to run fast, you must break it into subtasks
where each of them will get executed in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
- **Modularity:** You may want to build the system in a modular way by dividing the system functions into split processes or threads.
- **Convenience:** Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.Working together with multiple processes, require an inter process communication (IPC) method which will allow them to exchange data along with various information. There are two primary models of inter process communication:

1. Shared memory and
2. Message passing.

In the shared-memory model, a region of memory which is shared by cooperating processes gets established. Processes can be then able to exchange information by reading and writing all the data to the shared region. In the message-passing form, communication takes place by way of messages exchanged among the cooperating processes.
The two communications models are contrasted in the figure below:
Threads in Operating System

- A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control.
- There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process.
- Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks.
- Thread is often referred to as a lightweight process.



(a) Message Passing (b) Shared Memory

Shared memory:

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process.

```
#define BUFFER SIZE 10
typedef struct {
. . .
}item;
item buffer[BUFFER SIZE];
int in = 0;
int out = 0;


item next produced;
while (true) { /* produce an item in next produced */
while (((in + 1) % BUFFER SIZE) == out)
; /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
}
```

The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in ==out; the buffer is full when ((in + 1) % BUFFER SIZE) == out.

```
item next consumed;
while (true) { while (in == out)
; /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment.

- A message-passing facility provides at least two operations:
  send(message) receive(message)
- Here are several methods for logically implementing a link and the send()/receive() operations:

• Direct or indirect communication
• Synchronous or asynchronous communication
• Automatic or explicit buffering

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 defined as:
• send(P, message)—Send a message to process P.
• receive(Q, message)—Receive a message from process Q.

- In this scheme, the send() and receive() primitives are defined as follows:

• 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

indirect communication, the messages are sent to and received from ***mailboxes***, or ***ports***. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

- The send() and receive() primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.
• receive(A, message)—Receive a message from mailbox A.

Synchronization

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.
- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.

N**onblocking receive**. The receiver retrieves either a valid message or a null

Automatic or explicit buffering
Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
Bounded capacity. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

5. Communication in Client–Server Systems

- we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems.
- we explore three other strategies for communication in client–server systems: sockets, remote procedure calls (RPCs), and pipes.

Sockets

- A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number.
- In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services

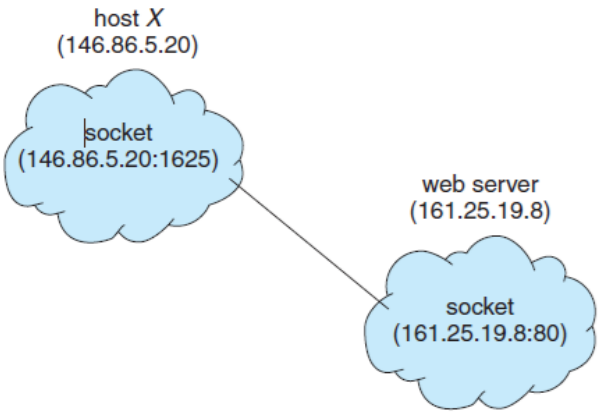


Figure 3.20 Communication using sockets.

Remote Procedure Calls

- The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections.
- The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a **stub(proxy)** on the client side.
- Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and **marshals** the parameters.
- Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

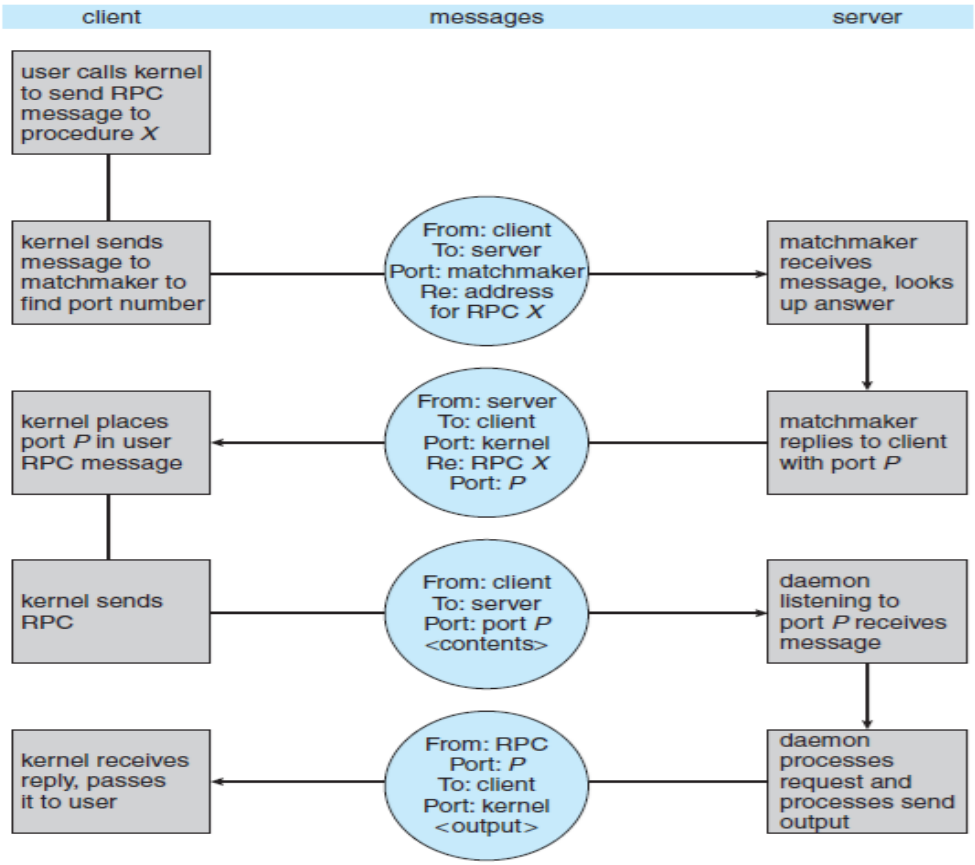


Figure 3.23 Execution of a remote procedure call (RPC).

Pipes

- A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as ***parent–child***) exist between the communicating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

Ordinary Pipes

- Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the **write-end**) and the consumer reads from the other end (the **read-end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- On UNIX systems, ordinary pipes are constructed using the function pipe(int fd[]) This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end.

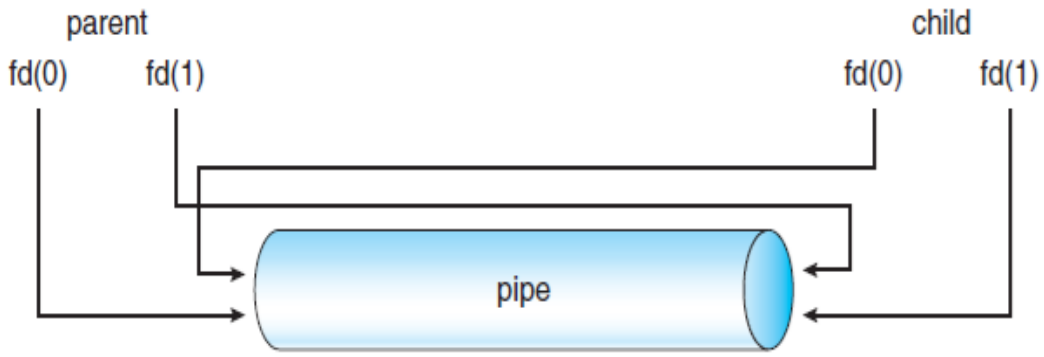


Figure 3.24 File descriptors for an ordinary pipe.

- **Named Pipes**
- Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication.
- Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(), and close() system calls. It will continue to exist until it is explicitly deleted from the file system.

6. Threads in Operating System

- A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control.
- There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process.
- Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks.
- Thread is often referred to as a lightweight process.
- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

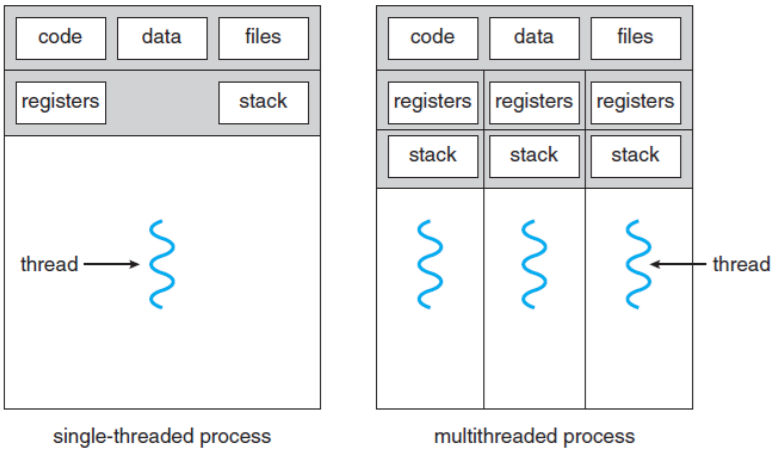


Figure 4.1 Single-threaded and multithreaded processes.

- **Need of Thread**:

o It takes far less time to create a new thread in an existing process than to create a new process.
o Threads can share the common data, they do not need to use Inter- Process communication.
o Context switching is faster when working with threads.
o It takes less time to terminate a thread than a process.

- **Types of Threads**

In the operating system, there are two types of threads.

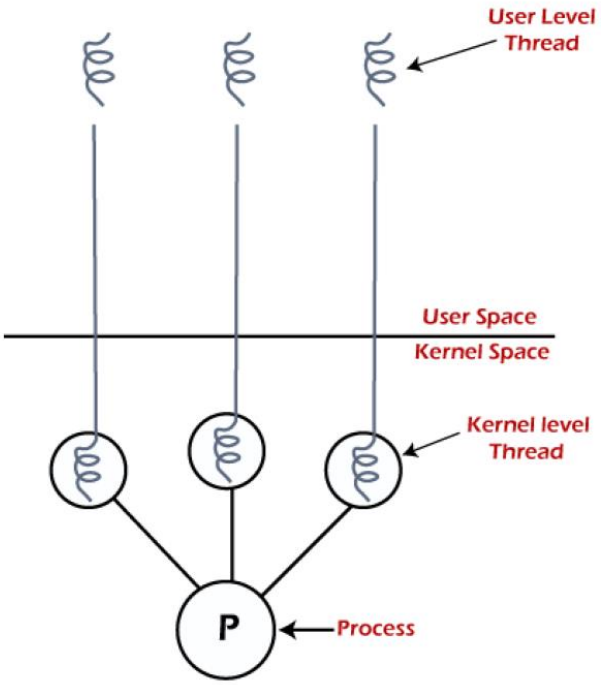
- 1. Kernel level thread.
- 2. User-level thread.

User-level thread

- The operating system does not recognize the user-level thread.
- User threads can be easily implemented and it is implemented by the user.
- If a user performs a user-level thread blocking operation, the whole process is blocked.
- The kernel level thread does not know nothing about the user level thread.
- The kernel-level thread manages user-level threads as if they are single-threaded processes?

Kernel level thread

- The kernel thread recognizes the operating system.
- There is a thread control block and process control block in the system for each thread and process in the kernel-level thread.
- The kernel-level thread is implemented by the operating system.
- The kernel knows about all the threads and manages them.
- The kernel-level thread offers a system call to create and manage the threads from user space.
- The implementation of kernel threads is more difficult than the user thread.
- Context switch time is longer in the kernel thread.



7.CPU Scheduling in Operating System

- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.
- Load store add store CPU burst read from file I/O burst store increment index CPU burst write to file I/O burst load store add store CPU burst read from file I/O burst Alternating sequence of CPU and I/O burst.

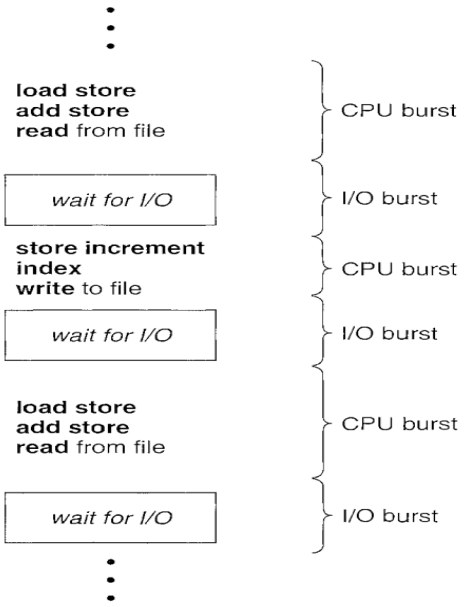


Figure 5.1 Alternating sequence of CPU and I/O bursts.

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 I/O request or invocation of wait for the termination of one of the child processes).
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,completion of I/O).
4. When a process **terminates**.

- **CPU Scheduling: Scheduling Criteria:**There are many different criteria to check when considering the **"best"** scheduling algorithm,they are:

- **Arrival time (AT)** − Arrival time is the time at which the process arrives in ready queue.
- **Burst time (BT) or CPU time of the process** − Burst time is the unit of time in which a particular process completes its execution.
- **Example**: If a process needs 5 milliseconds of CPU time to complete its execution, its burst time is 5 ms.
- **Completion time (CT)** − Completion time is the time at which the process has been terminated
- **Example**: If a process starts at time 2 ms and completes at time 12 ms, its completion time is 12 ms.

- **Turn-around time (TAT)** − The total time from arrival time to completion time is known as turn-around time. TAT can be written as,

Turn-around time (TAT) = Completion time (CT) – Arrival time (AT) or, **TAT = Burst time (BT) + Waiting time (WT)**

- **Waiting time (WT)** − Waiting time is the time at which the process waits for its allocation while the previous process is in the CPU for execution. WT is written as,
- ▶ **Waiting time (WT) = Turn-around time (TAT) – Burst time (BT)**
- **Response time (RT)** − Response time is the time at which CPU has been allocated to a particular process first time.
- In case of non-preemptive scheduling, generally Waiting time and Response time is same.
- **Gantt chart** − Gantt chart is a visualization which helps to scheduling and managing particular tasks in a project. It is used while solving scheduling problems, for a concept of how the processes are being allocated in different algorithms.

8.Scheduling Algorithms

There are 6 basic scheduling algorithms totally:

1. FCFS(First Come First Serve) Scheduling
2. SJF(Shortest Job First) Scheduling
3. Round Robin Scheduling
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues.

- ▶ The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- ▶ Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:
- ▶ **1.** System processes
- ▶ **2.** Interactive processes
- ▶ **3.** Interactive editing processes
- ▶ **4.** Batch processes

5. Student processes

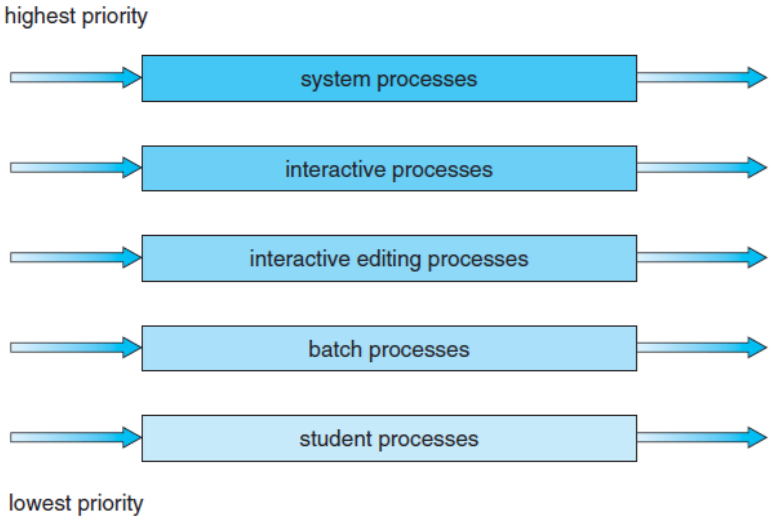


Figure 6.6 Multilevel queue scheduling.

- **Multilevel Feedback Queue Scheduling**

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues

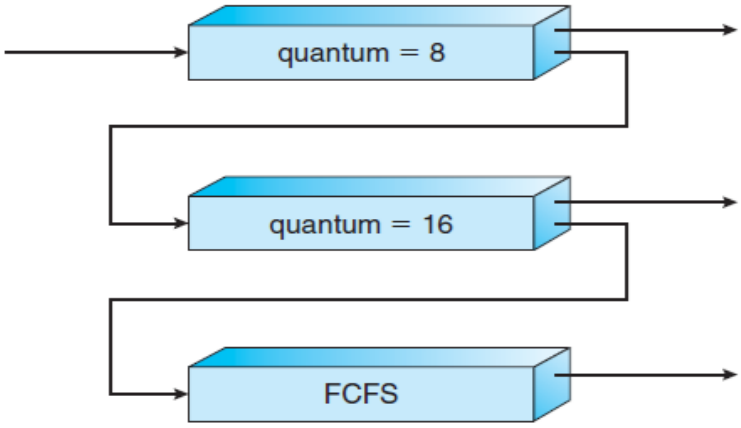


Figure 6.7 Multilevel feedback queues.

- consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 .The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving For queue 0.
- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
- • The number of queues
- • The scheduling algorithm for each queue
- • The method used to determine when to upgrade a process to a higher priority queue
- • The method used to determine when to demote a process to a lower priority queue
- • The method used to determine which queue a process will enter when that process needs service

9. Algorithm Evaluation

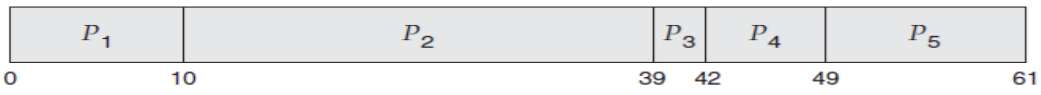
various evaluation methods we can use

- **Deterministic modeling** is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



- The waiting time is 0 milliseconds for process *P*1, 10 milliseconds for process *P*2, 39 milliseconds for process *P*3, 42 milliseconds for process *P*4, and 49 milliseconds for process *P*5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = **28milliseconds.**
- **SJF scheduling**, we execute the processes as The waiting time is 10 milliseconds for process *P*1, 32milliseconds for process *P*2, 0 milliseconds for process *P*3, 3 milliseconds for process *P*4, and 20 milliseconds for process *P*5. Thus, the average waiting time is **(10 + 32 + 0 + 3 + 20)/5 = 13milliseconds.**

- With the **RR algorithm**, we execute the processes as The waiting time is 0 milliseconds for process *P*1, 32 milliseconds for process *P*2, 20 milliseconds for process *P*3, 23 milliseconds for process *P*4, and 40 milliseconds for process *P*5. Thus, the average waiting time is **(0 + 32 + 20 + 23 + 40)/5 = 23milliseconds**.
- the average waiting time obtained with the **SJF** policy is less than half that obtained with **FCFS** scheduling; the **RR algorithm** gives us an intermediate value. Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms.

- **Queueing Models**
- The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.
- As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,
- $n = \lambda \times W$.
- This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

- **Simulations**
- A simulator in scheduling refers to a tool or program designed to emulate the behavior of different CPU scheduling algorithms. The primary goal of such a simulator is to model how various scheduling algorithms manage processes over time and allocate CPU resources. It helps visualize and analyze key metrics like CPU utilization, process turnaround time, waiting time, and response time.

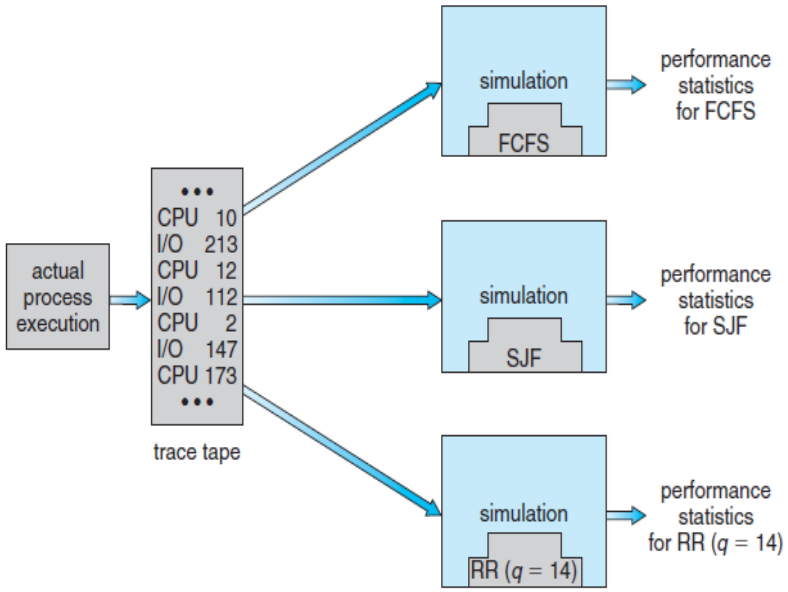


Figure 6.25 Evaluation of CPU schedulers by simulation.

Time	Event	CPU	I/O	Comment ------
0	Start CPU	P1	-	Process P1 starts on CPU for 10 units
10	I/O Start	-	P1 (213)	Process P1 moves to I/O
223	Resume CPU	P1	-	P1 resumes on CPU for 12 units
235	I/O Start	-	P1 (112)	Process P1 moves to I/O
347	Resume CPU	P1	-	P1 resumes on CPU for 2 units
349	I/O Start	-	P1 (147)	Process P1 moves to I/O
496	Resume CPU	P1	-	P1 resumes on CPU for 173 units
669	Process End	-	-	Process P1 completes

Scheduling algorithms

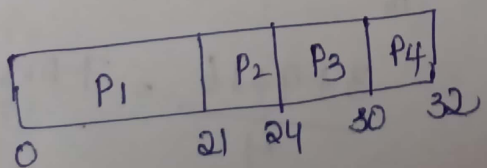
1) FCFS (First Come first Serve) scheduling
2) SJF (shortest Job first)
3) Round Robin.
4) priority Scheduling
5) Multilevel queue
6) Multilevel feedback queue

1) FCFS: As the name suggests, the process which arrives first, gets executed first. (or) we can say that the process which requests the CPU first, gets the CPU allocated first.

Ex:

process	Burst time
P_1	21
P_2	3
P_3	6
P_4	2

Gantt chart for above processes

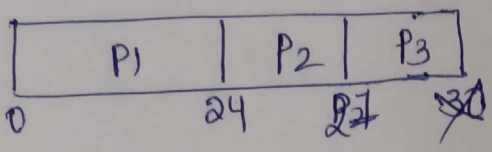


average waiting time will be = $\frac{0+21+24+30}{4}$

= 18.75ms

Ex: 2

Proces	Burst time
P_1	24
P_2	3
P_3	3



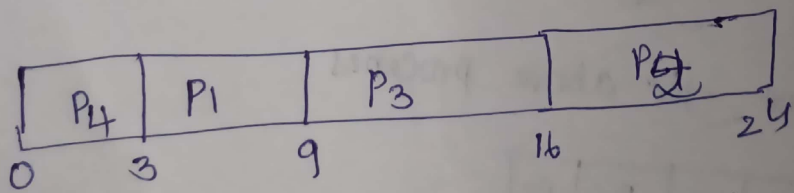
Average waiting time $= \frac{0+24+27+30}{3} = 17$ ms.

SJF ; It 1st scheduling works on proces with the shortest burst time or duration

- Preemptive
- Non-preemptive

Ex: 1)

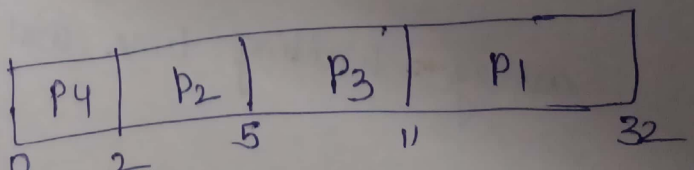
Proces	Burst time
P_1	6
P_2	8
P_3	7
P_4	3



Avg waiting time $= \frac{3+9+16}{4} = 7$ ms

Ex: 2

P_1	21
P_2	3
P_3	6
P_4	2



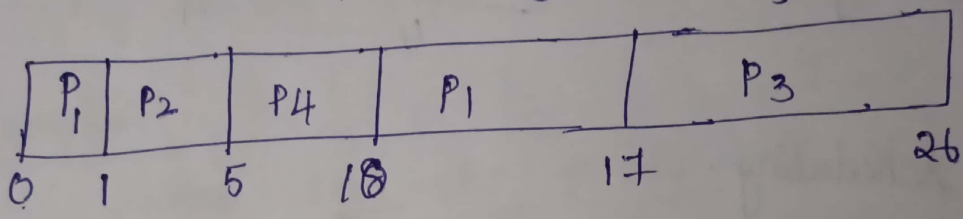
Avg waiting time $= \frac{2+5+11}{4} = \frac{18}{4}$

$= 4.5$ ms

Preemptive → It will preempt the currently executing executing process. It sometimes called as Shortest - remaining - time - first Scheduling

Ex:

Process	Arrival time	Burst time
P_1	0.	8
P_2	1.	4.
P_3	2	9
P_4	3	5



⇒ Avg Waiting time = ~~N+5+N0+~~

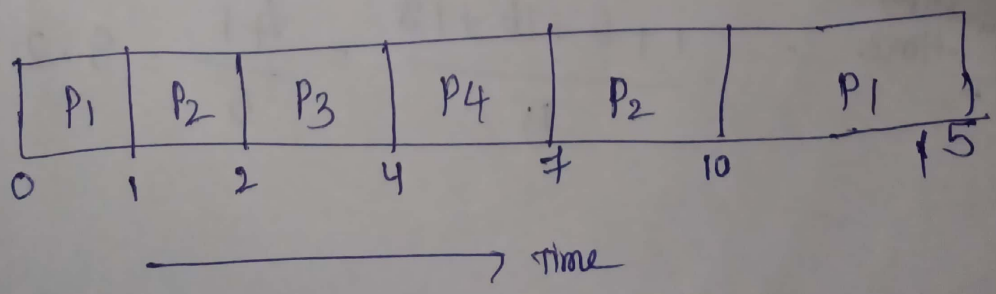
⇒ [(10-1) + (1-1) + (17-2) + (5-3)] / 4

= 26/4

= 6.5 ms

Ex:2)

Process	Arrival time	CPU burst time
P_1	0	6
P_2	1	4
P_3	2	2
P_4	3	3



Avg Waiting time =

Waiting time = Completion time - Burst time - Arrival time

P_1 Waiting time = (15-6-0) = 9ms

P_2 Waiting time = (7-4-1) = 2ms

P_3 Waiting time = (4-2-2) = 0ms

P_4 Waiting time = (10-3-3) = 4ms

Avg waiting time is $\frac{9+2+0+4}{4} = \frac{15}{4} = 3.75$ ms

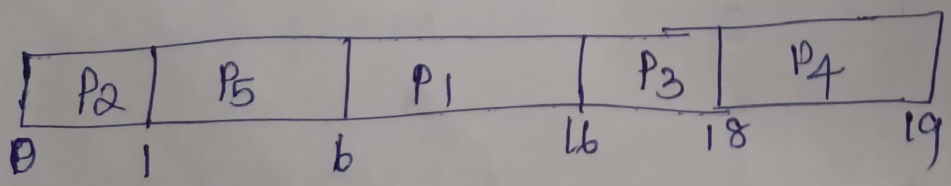
Priority Scheduling:

→ where each process associated with priority. The CPU is allocated to process with highest priority.

→ Equal-priority processes are scheduled in FCFS order

Ex.

Process	Burst time	priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

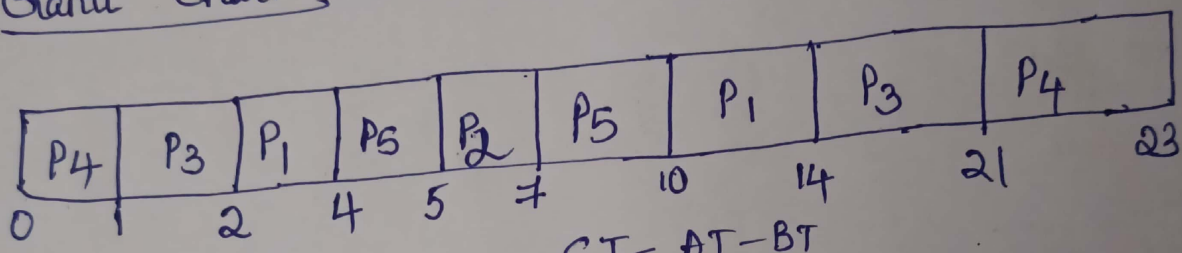


Avg Waiting time = $\frac{1+6+16+18}{5} = \frac{41}{5} = 8.2$ ms.

Pre-emptive priority Scheduling

Process	BT	AT	Priority
P_1	6	2	3
P_2	2	5	1
P_3	8	1	4
P_4	3	0	5
P_5	4	4	2

Gantt Chart



CT - AT - BT

Waiting time : $P_1 \rightarrow 14-2-6 = 6$

$P_2 \rightarrow 7-2-5 = 0$

$P_3 \rightarrow 21-1-8 = 12$

$P_4 \rightarrow 23-3-0 = 20$

$P_5 \rightarrow 10-4-4 = 2$

$$\text{Avg waiting time} = \frac{6+12+20+2+0}{5}$$

$$= \frac{40}{5}$$

$$= 8\text{ms}$$

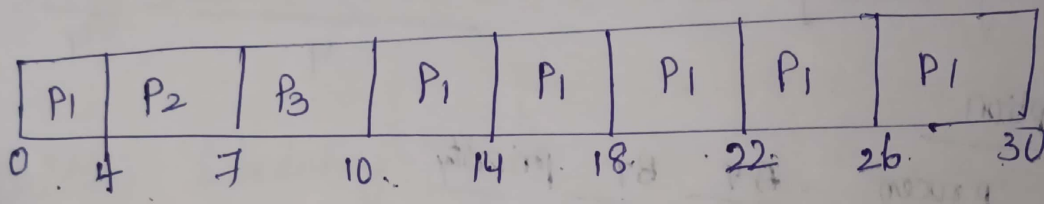
Round - Robin Scheduling

→ It is designed for time sharing systems.

→ A small unit of time, called a time quantum or timeslice is defined. It is generally from 10 to 100 ms in length

Ex: Time quantum is 4 ms

Process	Burst time
P_1	24
P_2	3
P_3	3



complete

Avg waiting time:

Waiting time = Turnaround time - Burst time

P_1 → waits for (10 − 4) = 6ms ⇒ 30 − 24 = 6

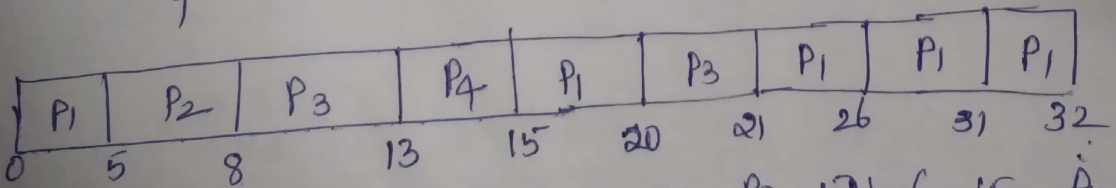
P_2 → " " = 4 ms = 7 − 3 = 4

P_3 → " " = 7 ms = 10 − 3 = 7

$$\Rightarrow \frac{6+4+7}{3} = \frac{17}{3} = 5.66 \text{ ms}$$

Ex: 2 Time Value = 5

Process	Burst time
P_1	21
P_2	3
P_3	6
P_4	2



$P_1 = 32 - 21 = 11$, $P_2 = 8 - 3 = 5$, $P_3 = 21 - 6 = 15$, $P_4 = 15 - 2 = 13$

$$\text{Avg waiting time} = \frac{11+5+15+13}{4} = \frac{44}{4} = 11 \text{ ms}$$