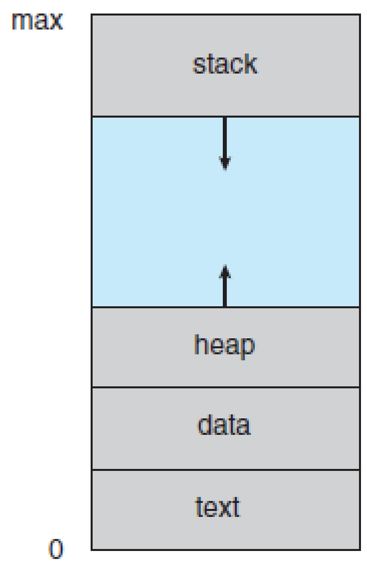
UNIT -2

PROCESS

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

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 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.

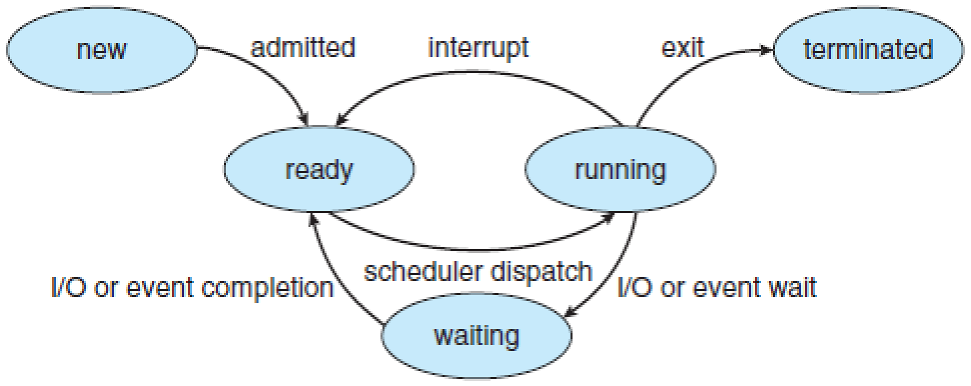
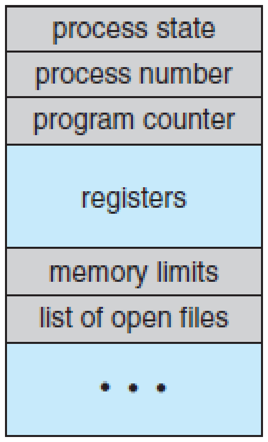


Diagram of process state.

PROCESS CONTROL BLOCK

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCB is shown in Figure



Process control block (PCB).

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.

PROCESS SCHEDULING

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process for program execution on the CPU.

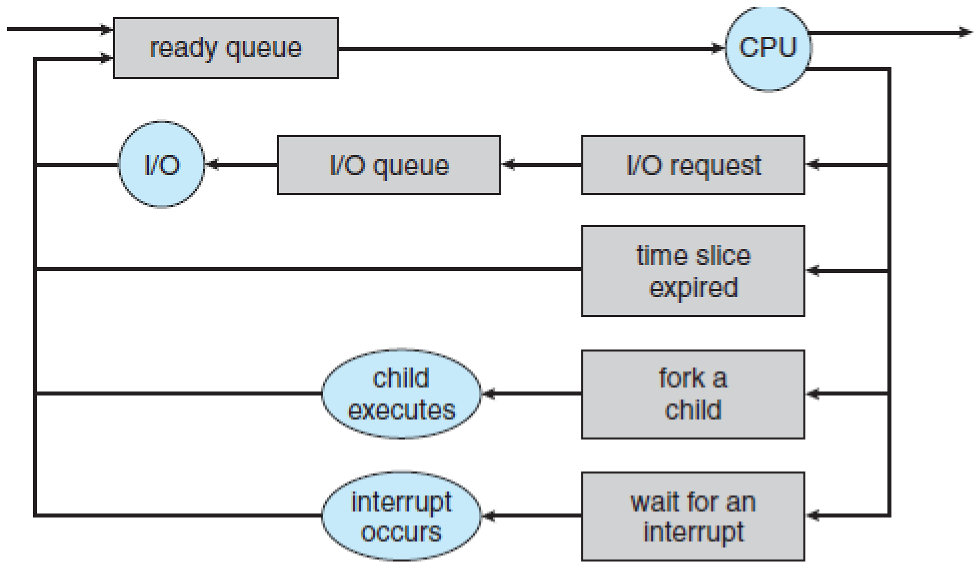
Scheduling Queues

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** − the processes which are blocked due to unavailability of an I/O device constitute this queue.

A common representation of process scheduling is a **queueing diagram**, such as that in Figure. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.



Queueing-diagram representation of process scheduling.

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

i) Long Term Scheduler

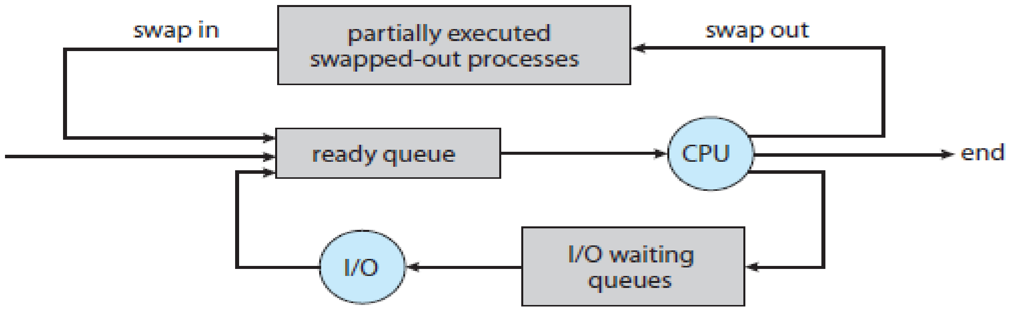
It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming.**

ii)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. The short-term scheduler must select a new process for the CPU frequently.

iii) Medium Term Scheduler

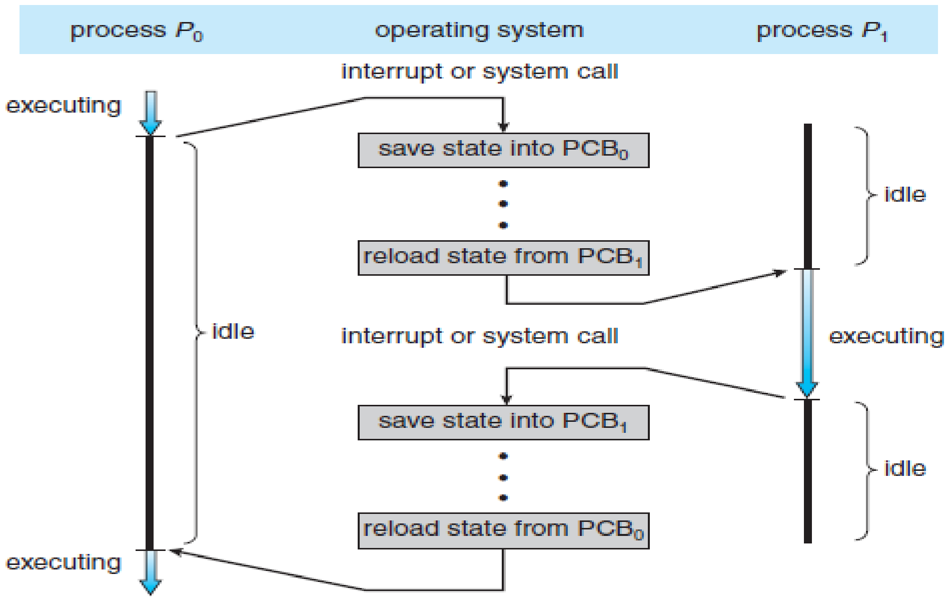
Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.



Context Switch

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.



OPERATIONS ON PROCESSES

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

Process Creation

During the course of execution, a process may create several new processes and the creating process is called a parent process, and the new processes are called the children of that process. Most operating systems identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system.

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

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it.

In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
}else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
}
return 0;
}
```

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are de allocated by the operating system. Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows).

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

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanismto inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

INTERPROCESS COMMUNICATION

Inter process communication is the mechanism provided by the **operating system** that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), 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. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

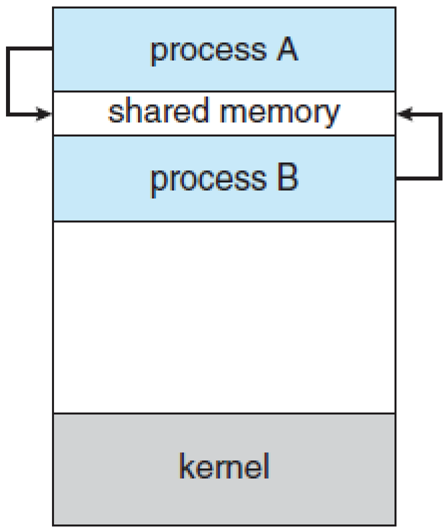
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

There are two fundamental models of interprocess communication:

- **shared memory**
- **Message passing**.

i) Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. They can then exchange information by reading and writing data in the shared areas.



Example:-

Producer–consumer problem

A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

ii) Message-Passing Systems

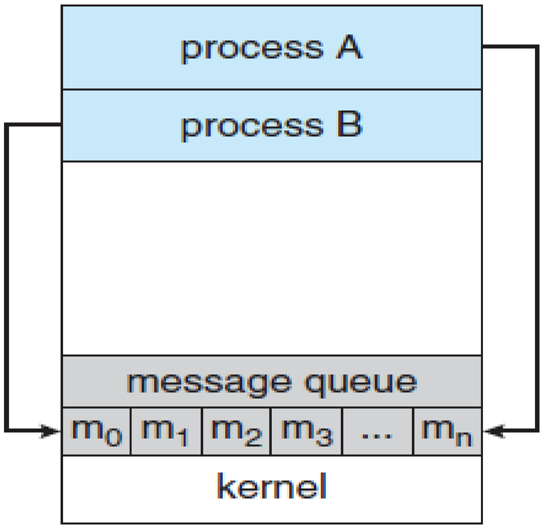
Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

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



Naming;-

.

Under **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.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

With ***indirect communication***, the messages are sent to and received from ***mailboxes***, or ***ports***.
A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. 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.

In this scheme, a communication link has the following properties:
• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

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.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Buffering

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

- **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, 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.

COMMUNICATION IN CLIENT–SERVER SYSTEMS

Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.

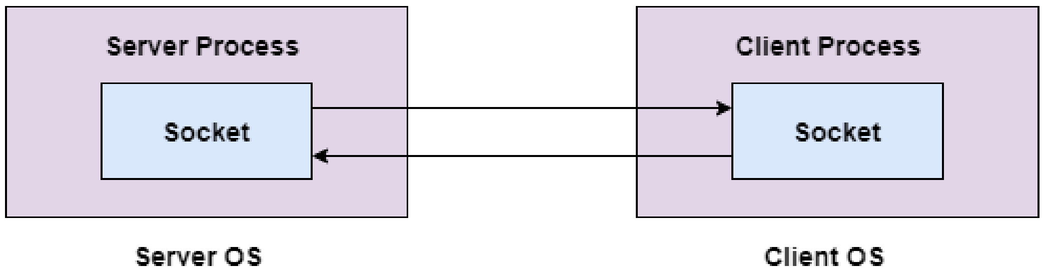
There are three main methods to client/server communication. These are given as follows −

Sockets

Sockets facilitate communication between two processes on the same machine or different machines. They are used in a client/server framework and consist of the IP address and port number. Many application protocols use sockets for data connection and data transfer between a client and a server.

Socket communication is quite low-level as sockets only transfer an unstructured byte stream across processes. The structure on the byte stream is imposed by the client and server applications.

A diagram that illustrates sockets is as follows −

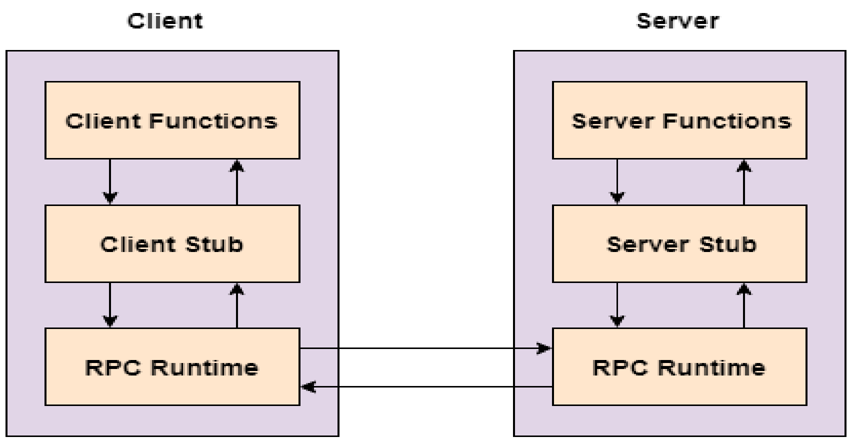


Remote Procedure Calls

These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.

A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.

A diagram that illustrates remote procedure calls is given as follows −



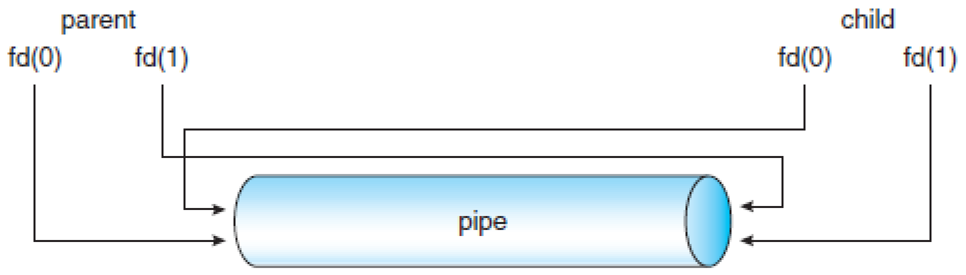
Pipes

Pipes are a powerful inter-process communication (IPC) mechanism in operating systems, particularly useful for client-server communication on the same machine. They allow processes to communicate by sending data through a unidirectional or bidirectional channel.
The two different types of pipes are ordinary pipes and named pipes.

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 fromthe 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.



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. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished.

MULTITHREADING

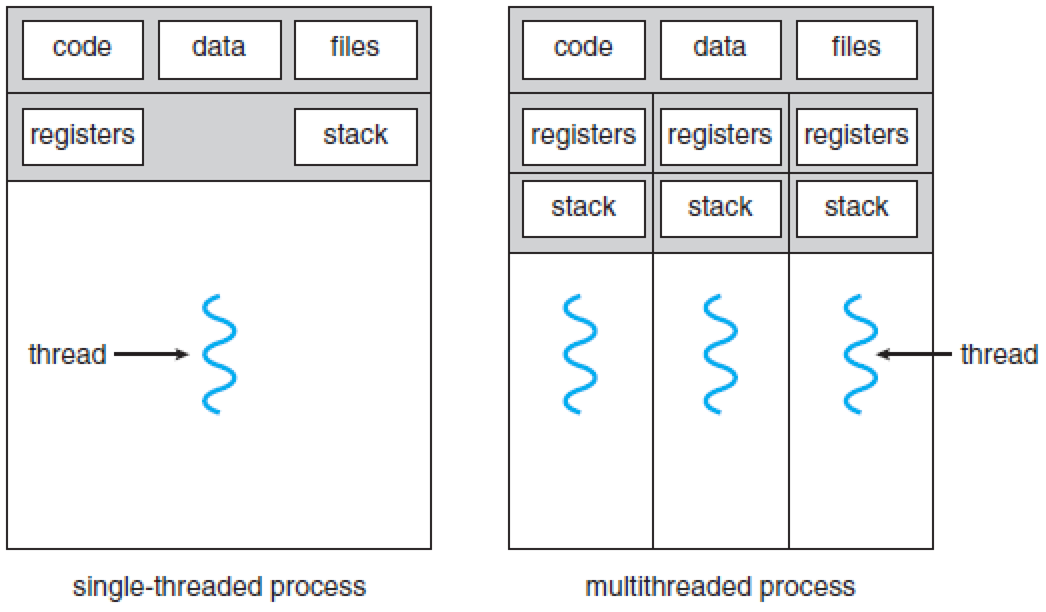
Thread

A thread is also called a **lightweight process**. A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

Multithreading

Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. With the use of multithreading, multitasking can be achieved.



All threads inside a process will have to share compute resources such as code, data, files, and memory space with its peer thread, but stacks and registers will not be shared, and each new thread will have its own stacks and registers.

BENEFITS OF MULTITHREADED PROGRAMMING

The benefits of multithreaded programming can be broken down into four major categories:
1. Responsiveness. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces.

2. Resource sharing. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. Economy. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

4. Scalability. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

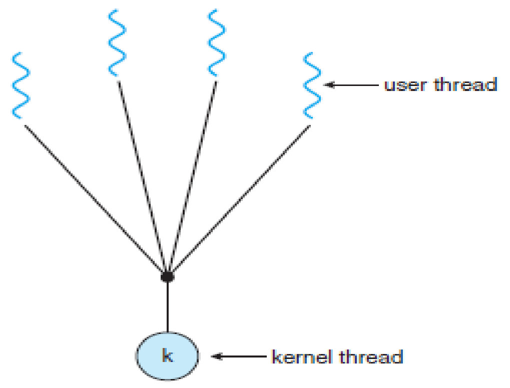
MULTITHREADING MODELS

However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
There are three sorts of models in multithreading.

- Many-to-many models
- Many-to-one model
- One-to-one model

1. Many-to-One: There will be a many-to-one relationship model between threads, as the name implies. Multiple user threads are linked or mapped to a single kernel thread in this case.

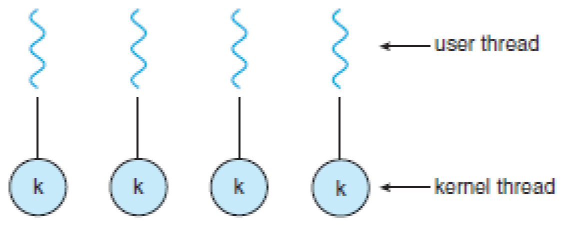
Management of threads is done on a user-by-user basis, which makes it more efficient. It converts a large number of user-level threads into a single Kernel-level thread.



The following issues arise in many to one model:

- A block statement on a user-level thread stops all other threads from running.
- Use of multi-core architecture is inefficient.
- There is no actual concurrency.
-

2. One-to-One: We can deduce from the name that one user thread is mapped to one separate kernel thread. The user-level thread and the kernel-level thread have a one-to-one relationship. The many-to-one model for thread provides less concurrency than this architecture. When a thread performs a blocking system call, it also allows another thread to run.



One-to-one model.

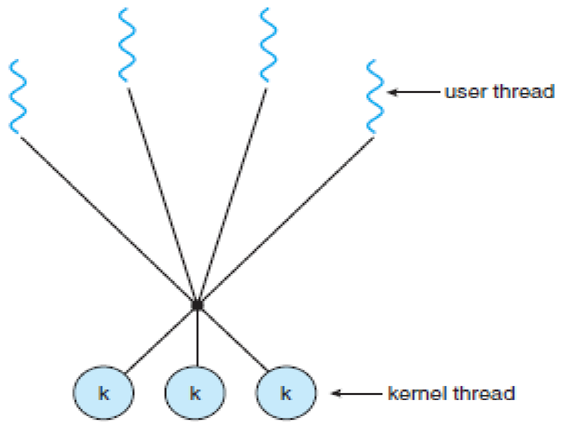
It provides the following benefits over the many-to-one model:

- A block statement on one thread does not cause any other threads to be blocked.
- Concurrency in the true sense.
- Use of a multi-core system that is efficient.

3. Many-to-Many: From the name itself, we may assume that there are numerous user threads mapped to a lesser or equal number of kernel threads. This model has a version called a two-level model, which incorporates both many-to-many and one-to-one relationships. When a thread makes a blocked system call, the kernel can schedule another thread for execution.

The particular application or machine determines the number of kernel threads. The many-to-many model multiplexes any number of user threads onto the same number of kernel threads or fewer kernel threads. On the utilization of multiprocessor architectures, developers may build as many user threads as they need, and the associated kernel threads can execute in parallel.

So, in a multithreading system, this is the optimal paradigm for establishing the relationship between user thread and the kernel thread.



Many-to-many model.

THREAD LIBRARIES

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.
Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***.

- Pthread library can be implemented either at the user space or **kernel space.** The Pthread is often implemented at the Linux, unix and solaris, and it is highly portable as its code written in pthread can typically be compiled and run on different unix without much modifications.
- Pthread program always have **pthread.h** header file. Windows doesn't support pthread standard. It support **C** and **C++** languages.

Key Functions:

- pthread_create(): Create a new thread.
- pthread_join(): Wait for a thread to finish.
- pthread_exit(): Exit from a thread.

Win32 Thread

- Win32 thread is a part of Windows operating system and it is also called as Windows Thread. It is a kernel space library.
- In this thread we can also achieve parallelism and concurrency in same manner as in pthread.
- Win32 thread are created with the help of createThread() function. Window thread support Thread Local Storage (TLS) as allow each thread to have its own unique data, and these threads can easily share data as they declared globally.
- They providing native and low level support for multi-threading. It means they are tightly integrated with window OS and offer efficient creation and thread management.

Header File: <windows.h>

Key Functions:

- CreateThread(): Create a new thread.
- WaitForSingleObject(): Wait for a thread to complete.
- ExitThread(): Exit from a thread.

Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and Mac OS X. The Java thread API is available for Android applications as well.

There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the Thread class and to override its run() method. An alternative—and more commonly used— technique is to define a class that implements the Runnable interface.

THREADING ISSUES

1. The fork() and exec() System Calls

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
The exec() system call typically works in the same way. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.
Which of the two versions of fork() to use depends on the application. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

2. Signal Handling

Signal is easily directed at the process in single threaded applications. However, in relation to multithreaded programs, the question is which thread of a program the signal should be sent.
Suppose the signal will be delivered to:

- Every line of this process.
- Some special thread of a process.
- thread to which it applies

A signal may be handled by one of two possible handlers:
1. A default signal handler
2. A user-defined signal handler
Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-defined signal handler that is called to handle the signal.

3. Thread Cancellation

The process of prematurely aborting an active thread during its run is called ‘thread cancellation’. So, let’s take a look at an example to make sense of it. Suppose, there is a multithreaded program whose several threads have been given the right to scan a database for some information. The other threads however will get canceled once one of the threads happens to return with the necessary results.

The target thread is now the thread that we want to cancel. Thread cancellation can be done in two ways:

- **Asynchronous Cancellation:** The asynchronous cancellation involves only one thread that cancels the target thread immediately.
- **Deferred Cancellation:** In the case of deferred cancellation, the target thread checks itself repeatedly until it is able to cancel itself voluntarily or decide otherwise.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation.

4. Thread Pool

The server develops an independent thread every time an individual attempts to access a page on it. The establishment of a fresh thread is another thing that worries us. The creation of a new thread should not take more than the amount of time used up by the thread in dealing with the request and quitting after because this will be wasted CPU resources.

Hence, thread pool could be the remedy for this challenge. The notion is that as many fewer threads as possible are established during the beginning of the process. A group of threads that forms this collection of threads is referred as a thread pool. There are always threads that stay on the thread pool waiting for an assigned request to service.

A new thread is spawned from the pool every time an incoming request reaches the server, which then takes care of the said request. Having performed its duty, it goes back to the pool and awaits its second order.

5. Thread-Local Storage

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data.We will call such data **thread-local storage** (or **TLS**.)

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. In some ways, TLS is similar to static data. The difference is that TLS data are unique to each thread. Most thread libraries—including Windows and Pthreads—provide some form of support for thread-local storage; Java provides support as well.

CPU SCHEDULING

CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer.

Preemptive Scheduling Algorithms

Preemptive scheduling allows the operating system to interrupt and suspend a currently running process to allocate the CPU to another process.
Some common preemptive scheduling algorithms include:

- Round Robin (RR)
- Shortest Remaining Time First (SRTF):
- Priority Scheduling (Preemptive):
- Multilevel Queue Scheduling

Non-Preemptive Scheduling Algorithms

Non-preemptive scheduling algorithms do not allow a running process to be interrupted. Once a process starts execution, it runs to completion before another process is scheduled. Common non-preemptive scheduling algorithms include:

1. First-Come, First-Served (FCFS):
2. Shortest Job Next (SJN) or Shortest Job First (SJF):
3. Priority Scheduling (Non-Preemptive):
4. Multi-level Feedback Queue (MLFQ):

SCHEDULING CRITERIA

In operating systems (OS), scheduling criteria refer to the various metrics and objectives used to manage the execution of processes or tasks. The goal is to optimize the performance and responsiveness of the system. Here are some common scheduling criteria:

1. **CPU Utilization**: The percentage of time the CPU is actively working on processes. High CPU utilization means the CPU is busy most of the time.
2. **Throughput**: The number of processes completed per unit of time. Higher throughput indicates that more processes are being completed.
3. **Turnaround Time**: The total time taken to execute a process, from its arrival in the ready queue to its completion. This includes waiting time, execution time, and any time spent in I/O operations.

4. **Waiting Time**: The total time a process spends waiting in the ready queue before it gets CPU time. Lower waiting time is desirable as it means processes get to execute faster.
5. **Response Time**: The time between submitting a request and receiving the first response. This is particularly important in interactive systems where users expect quick feedback.

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 many different CPU-scheduling algorithms.

1. First-Come, First-Served (FCFS) Scheduling

FCFS is considered as simplest CPU-scheduling algorithm. In FCFS algorithm, the process that requests the CPU first is allocated in the CPU first. The implementation of FCFS algorithm is managed with FIFO (First in first out) queue. FCFS scheduling is non-preemptive. Non preemptive means, once the CPU has been allocated to a process, that process keeps the CPU until it executes a work or job or task and releases the CPU, either by requesting I/O.

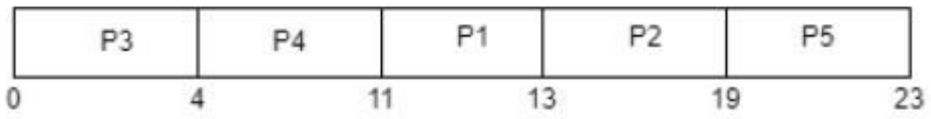
Problem 1

Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	5	6
P3	0	4
P4	0	7
P5	7	4

Solution:-

Gantt chart



Process ID	Arrival Time(AT)	Burst Time(BT)	Completion Time (CT)	Turn-Around Time TAT=CT-AT	Waiting Time WT=TAT-BT	Response Time RT
P1	2	2	13	13-2= 11	11-2= 9	9
P2	5	6	19	19-5= 14	14-6= 8	8
P3	0	4	4	4-0= 4	4-4= 0	0
P4	0	7	11	11-0= 11	11-7= 4	4
P5	7	4	23	23-7= 16	16-4= 12	12

Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)

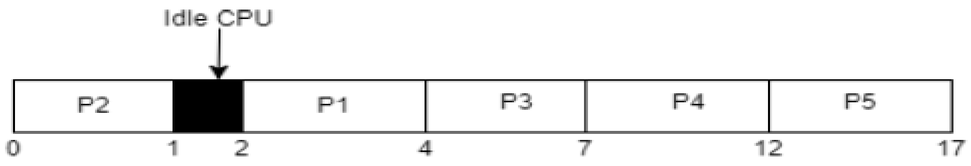
Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds)

Problem 2

Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	0	1
P3	2	3
P4	3	5
P5	4	5

<u>Solution</u>
Gantt chart −



For this problem CT, TAT, WT, RT is shown in the given table −

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	4	4-2= 2	2-2= 0	0
P2	0	1	1	1-0= 1	1-1= 0	0
P3	2	3	7	7-2= 5	5-3= 2	2
P4	3	5	12	12-3= 9	9-5= 4	4
P5	4	5	17	17-4= 13	13-5= 8	8

Average Waiting time = (0+0+2+4+8)/5 = 14/5 = 2.8
Average Turn-around time = (2+1+5+9+13)/5 = 30/5 = 6

Shortest-Job-First Scheduling

SJF scheduling algorithm schedules the processes according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.

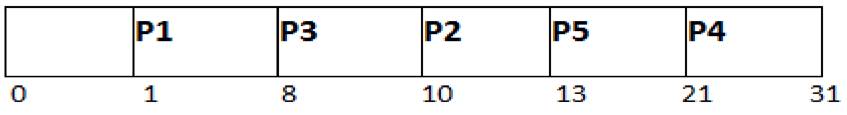
However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

Example

In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

PID	Arrival Time	Burst Time
1	1	7
2	3	3
3	6	2
4	7	10
5	9	8

Solution:-



Since, No Process arrives at time 0 hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives).

According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.

PID	Arrival Time(AT)	Burst Time(BT)	Completion Time(CT)	Turn Around Time TAT=CT-AT	Waiting Time WT=TAT-BT
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

Average Waiting Time = 27/5

<u>Priority Scheduling</u>

In priority scheduling algorithm a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. Priority scheduling algorithm can be implemented in two ways i.e non-preemptive and preemptive

i. Preemptive Priority Scheduling

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

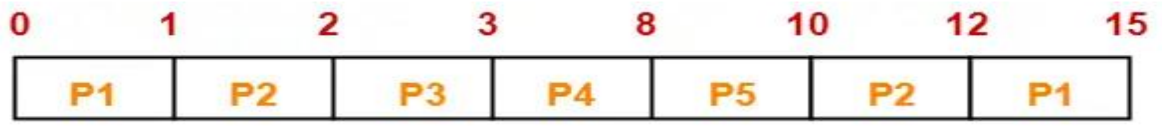
<u>Problem:</u>

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

Solution-

Gantt Chart-



Gantt Chart

PID	Arrival Time(AT)	Burst Time(BT)	Completion Time(CT)	Turn Around Time TAT=CT-AT	Waiting Time WT=TAT-BT
1	0	4	15	15	11
2	1	3	12	11	8
3	2	1	3	1	0
4	3	5	8	5	0
5	4	2	10	6	4

Now,

- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

i. **Non-Preemptive Priority Scheduling**

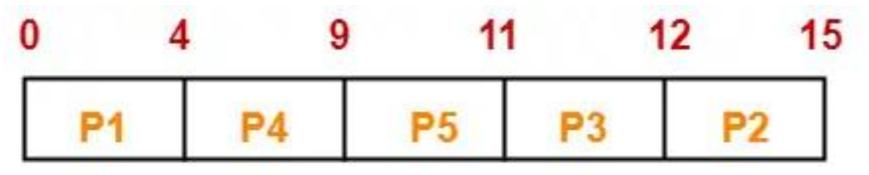
Problem:

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

Solution-

Gantt Chart-



Gantt Chart

PID	Arrival Time(AT)	Burst Time(BT)	Completion Time(CT)	Turn Around Time TAT=CT-AT	Waiting Time WT=TAT-BT
1	0	4	4	4	0
2	1	3	15	14	11
3	2	1	12	10	9
4	3	5	9	6	1
5	4	2	11	7	5

Now,

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined.

First, the processes which are eligible to enter the ready queue enter the ready queue. After entering the first process in Ready Queue is executed for a Time Quantum chunk of time. After execution is complete, the process is removed from the ready queue. Even now the process requires some time to complete its execution, then the process is added to Ready Queue.

Example:-

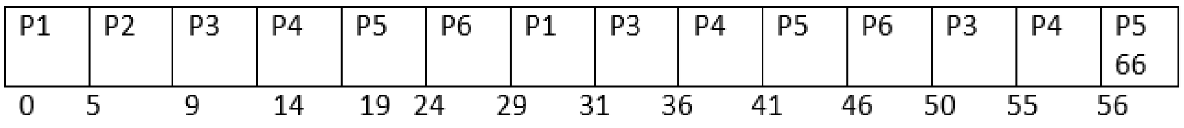
SNO	PROCESS ID	ARRIVAL TIME	BURST TIME
1	P1	0	7
2	P2	1	4
3	P3	2	15
4	P4	3	11
5	P5	4	20
6	P6	4	9

Assume Time Quantum TQ = 5

Ready Queue:

P1, P2, P3, P4, P5, P6, P1, P3, P4, P5, P6, P3, P4, P5

Gantt chart:



Process Id	Arrival Time(AT)	Burst Time (BT)	Completion Time (CT)	Turn Arround Time TRT=CT-AT	Waiting Time WT=(TRT-BT)
P1	0	7	31	31	24
P2	1	4	9	8	4
P3	2	15	55	53	38
P4	3	11	56	53	42
P5	4	20	66	62	42
P6	4	9	50	46	37

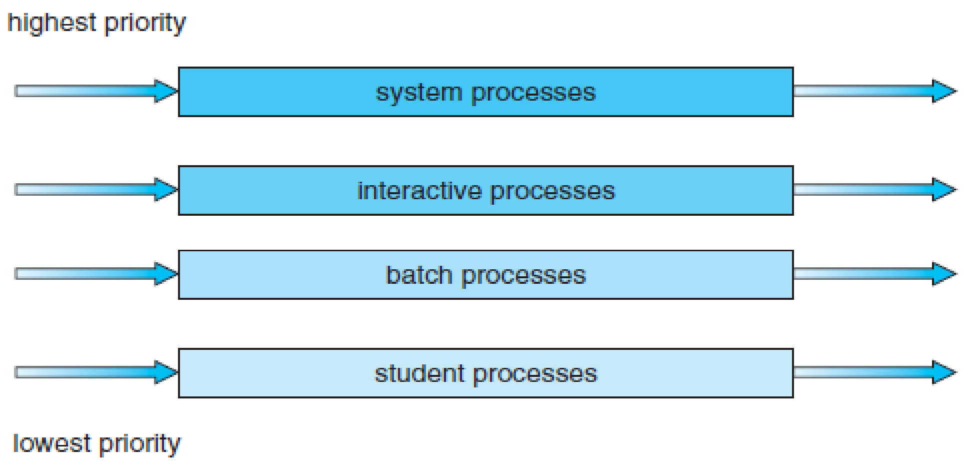
Average waiting time = (24+4+38+42+42+37)/5 = 187/5 = 37.4

Multilevel 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. Each queue has its own scheduling algorithm.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. System processes
2. Interactive processes (Fore ground)
3. Batch processes (Back ground)
4. Student processes



- **System Processes** − System processes refer to operating system tasks or processes that are essential for the proper functioning of the system. These processes handle critical operations such as memory management, process scheduling, I/O handling, and other system-level functions. System processes typically have the highest priority in the MLQ architecture and are executed before other types of processes.

- **Interactive Processes** − Interactive processes are user-oriented processes that require an immediate or near-real-time response. These processes are typically initiated by user interaction, such as keyboard input or mouse clicks.

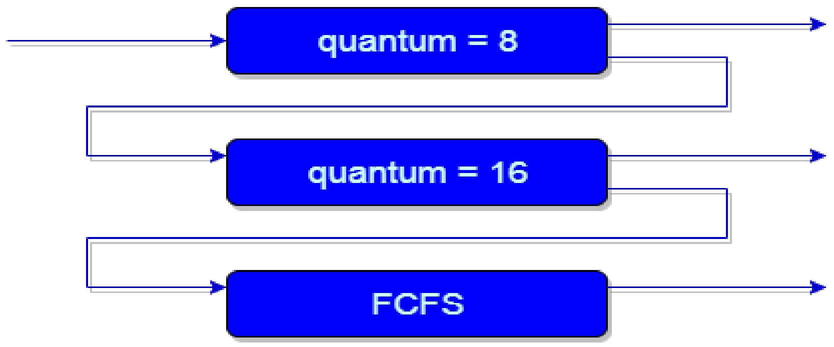
- **Batch Processes** − Batch processes are non-interactive processes that are executed in the background without direct user interaction. These processes often involve executing a series of tasks or jobs that can be automated and executed without user intervention.

- **Student Process**
  The system process always gets the highest priority while the student processes always get the lowest priority.

Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
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.



First of all, Suppose that queues 1 and 2 follow round robin with time quantum 8 and 16 respectively and queue 3 follows FCFS. One of the implementations of Multilevel Feedback Queue Scheduling is as follows:

1. If any process starts executing then firstly it enters queue 1.
2. In queue 1, the process executes for 8 unit and if it completes in these 8 units or it gives CPU for I/O operation in these 8 units unit than the priority of this process does not change, and if for some reasons it again comes in the ready queue than it again starts its execution in the Queue 1.
3. If a process that is in queue 1 does not complete in 8 units then its priority gets reduced and it gets shifted to queue 2.
4. Above points 2 and 3 are also true for processes in queue 2 but the time quantum is 16 units. Generally, if any process does not complete in a given time quantum then it gets shifted to the lower priority queue.
5. After that in the last queue, all processes are scheduled in an FCFS manner.

6. It is important to note that a process that is in a lower priority queue can only execute only when the higher priority queues are empty.
7. Any running process in the lower priority queue can be interrupted by a process arriving in the higher priority queue.

INTER-PROCESS COMMUNICATION

Race condition

Race condition occurs when multiple processes read and write the same variable i.e. they have access to some shared data and they try to change it at the same time. In such a scenario threads are “racing” each other to access/change the data.

Example:-

Producer-Consumer problem:-

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

The code for the producer process can be modified as follows:

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

The code for the consumer process can be modified as follows:

```
while (true)
{ while (counter == 0); /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
counter--;/* consume the item in next consumed */
}
```

suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements “counter++” and “counter--”.

Note that the statement “counter++” may be implemented in machine language (on a typical machine) as follows:
register1 = counter
register1 = register1 + 1
counter = register1

Similarly, the statement “counter--” is implemented as follows:
register2 = counter
register2 = register2 − 1
counter = register2

One such interleaving is the following:
T0: producer execute register1 = counter {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = counter {register2 = 5}
T3: consumer execute register2 = register2 − 1 {register2 = 4}
T4: producer execute counter = register1 {counter = 6}
T5: consumer execute counter = register2 {counter = 4}

Notice that we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at *T*4 and *T*5, we would arrive at the incorrect state “counter == 6”.

CRITICAL SECTION

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

The general structure of a typical process

```
do {
      entry section
            critical section
      exit section
            remainder section
} while (true);
```

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.

MUTEX LOCKS

Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.) We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock.
A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.
The definition of acquire() is as follows:

```
acquire()
{
while (!available); /* busy wait */
available = false;;
}
```

The definition of release() is as follows:

```
release()
{
available = true;
}
```

SEMAPHORES

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**
  The wait operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed.

```
wait(S)
 {
while (S <= 0); // busy wait
S--;
}
```

- **Signal**
  The signal operation increments the value of its argument **S**.

```
signal(S)
{
 S++;
}
```

Mutual Exclusion implementation:

```
do
{
wait(mutex);
critical section
signal(mutex);
remainder section
}
```

Here initially mutex value is 1

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**
  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores**
  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

MONITORS

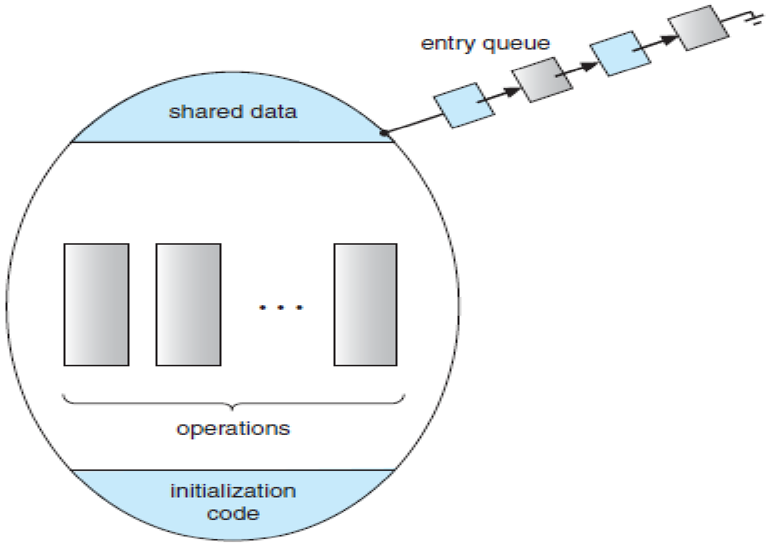
Monitors are a programming language component that aids in the regulation of shared data access. The Monitor is a package that contains shared data structures, operations, and synchronization between concurrent procedure calls. Therefore, a monitor is also known as a synchronization tool. Java, C#, Visual Basic, Ada, and concurrent Euclid are among some of the languages that allow the use of monitors.

Syntax:-

```
monitor monitor name
{ /* shared variable declarations */
function P1 ( . . . )
{ . . .
}
function P2 ( . . . )
{ . . .
}
.
.
.
function Pn ( . . . )
{ . . .
}
initialization code ( . . . )
 { . . .
 }
}
```

A **monitor type** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

Schematic view of a monitor



Condition Variables

There are two sorts of operations we can perform on the monitor's condition variables:

1. Wait
2. Signal

Consider a condition variable (y) is declared in the monitor:

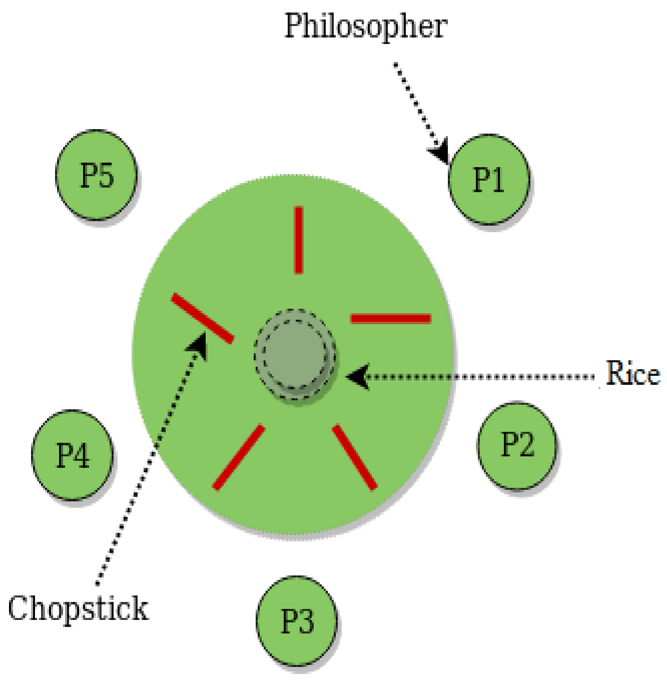
y.wait(): The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.

y.signal(): If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

CLASSIC PROBLEMS OF SYNCHRONIZATION

1. The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



Problems:-

a) Adjacent philosophers can't eat simultaneously
b) Suppose that all philosophers become hungry simultaneously and each grab left chopstick.
c) When each philosopher try to grab right chopstick, each philosopher will wait for infinite amount of time and because of starvation (Deadlock) a philosopher may die.

Solution:-

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick [5];

```
do { wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
. . .
/* eat for awhile */
. . .
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
. . .
/* think for awhile */
. . .
} while (true);
```

Several possible remedies to the deadlock problem are replaced by:

• Allow at most four philosophers to be sitting simultaneously at the table.

• Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

• Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

2. Readers Writers Problem

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

The solution of readers and writers can be implemented using binary semaphores.

In the solution to the first readers–writers problem, the reader processes share the following data structures:
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;

The structure of a writer process is .

```
do {
 wait(rw mutex);
. . .
/* writing is performed */
. . .
signal(rw mutex);
} while (true);
```

The structure of a reader process is

```
do {                                    // entry section
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex);
signal(mutex);
. . .
/* reading is performed */
. . .
wait(mutex);                    // exit section
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```