# Operating System

## UNIT – II

Process and CPU Scheduling - Process concepts and scheduling, Operations on processes, Cooperating Processes,

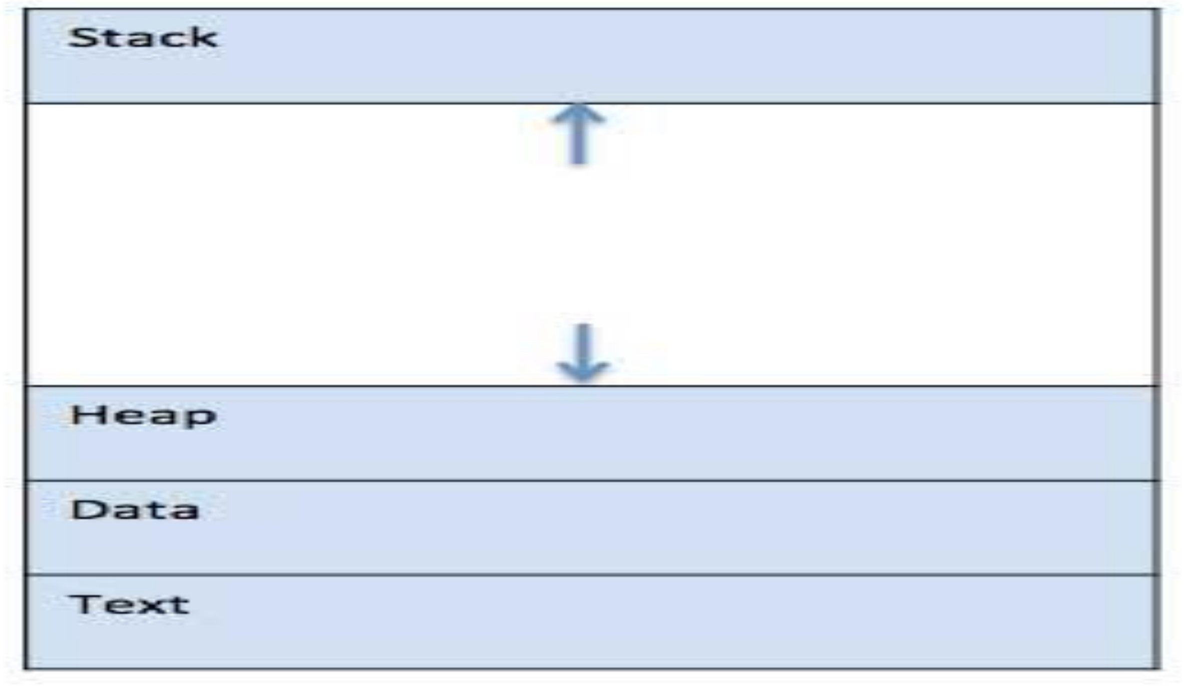
Threads, and Interposes Communication, Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling.

System call interface for process management-fork, exit, wait, waitpid, exec

## Process

- **A process is basically a program in execution. The execution of a process must progress in a sequential fashion.**
- **A process is defined as an entity which represents the basic unit of work to be implemented in the system.**
- To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

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



| S.N. | Component & Description |
|---|---|
| 1 | **Stack**<br><br>The process Stack contains the temporary data such as method/function parameters, return address and local variables. |
| 2 | **Heap**<br><br>This is dynamically allocated memory to a process during its run time. |
| 3 | **Text**<br><br>This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. |
| 4 | **Data**<br><br>This section contains the global and static variables. |

## <u>**Program**</u>

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language.

For example, here is a simple program written in C programming language −

```
#include <stdio.h>

int main() {
  printf("Hello, World! \n");
  return 0;
}
```

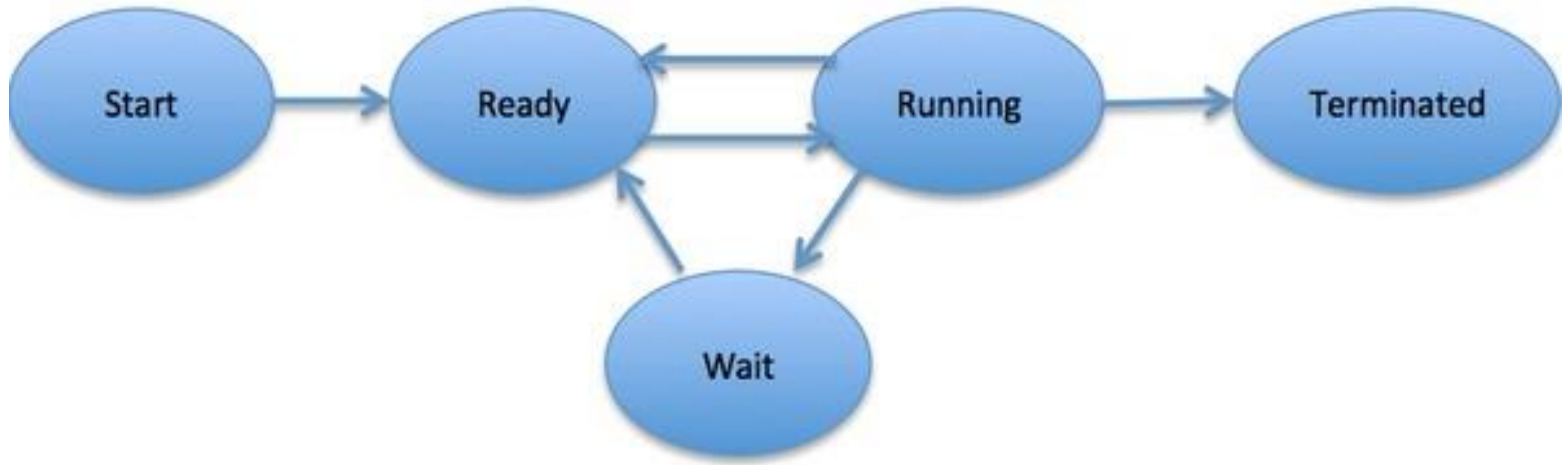
A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

## Process Life Cycle

- When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.
- In general, a process can have one of the following five states at a time.

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

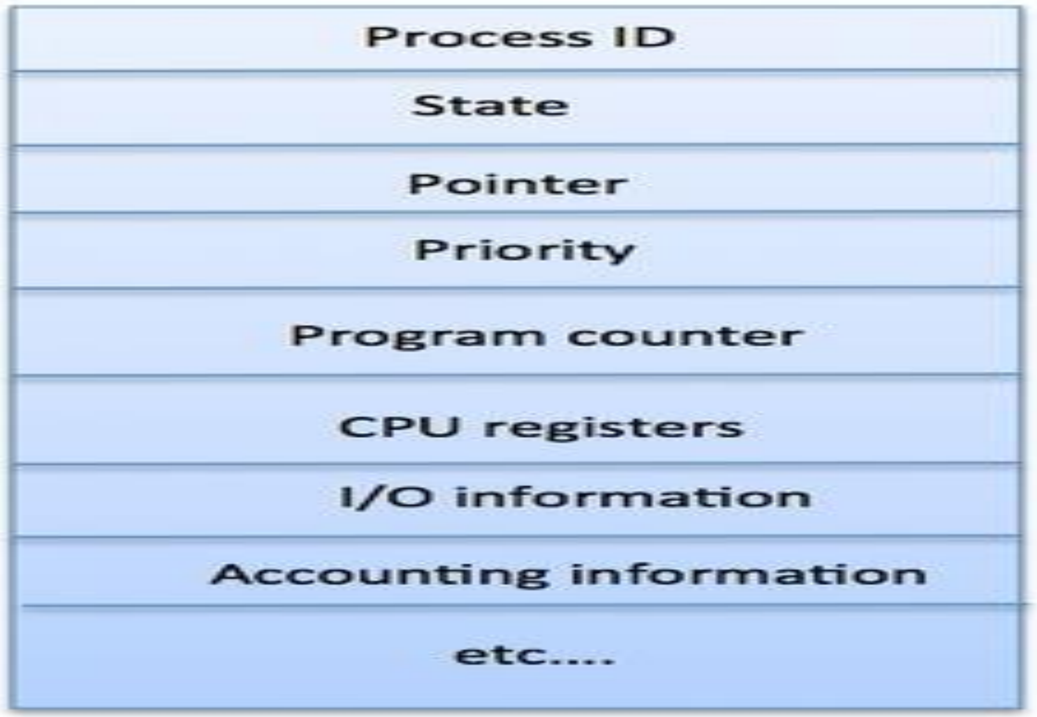

Start
Ready
Running
Terminated
Wait

Process Control Block (PCB)

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

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

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



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

## Operating System - Process Scheduling

**<u>Definition</u>**

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

**Process Scheduling Queues**

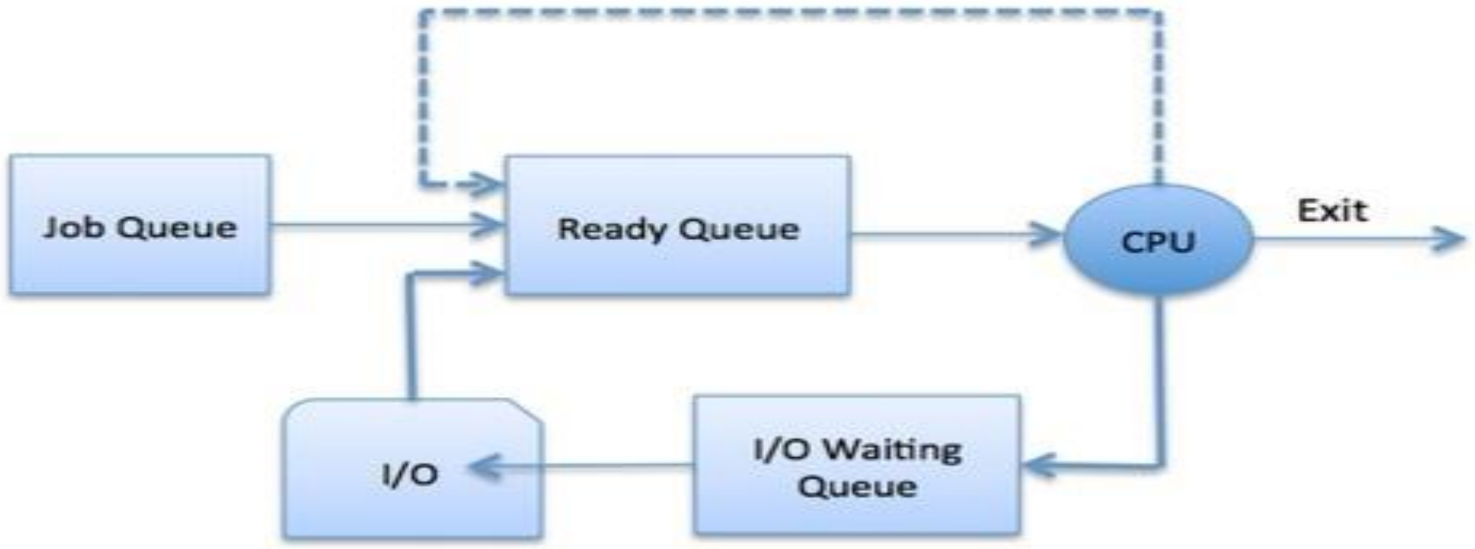
The OS maintains all PCBs in Process Scheduling Queues.

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.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

**Schedulers**

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

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

## Long Term Scheduler

- It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

## Short Term Scheduler

- It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

## Medium Term Scheduler

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

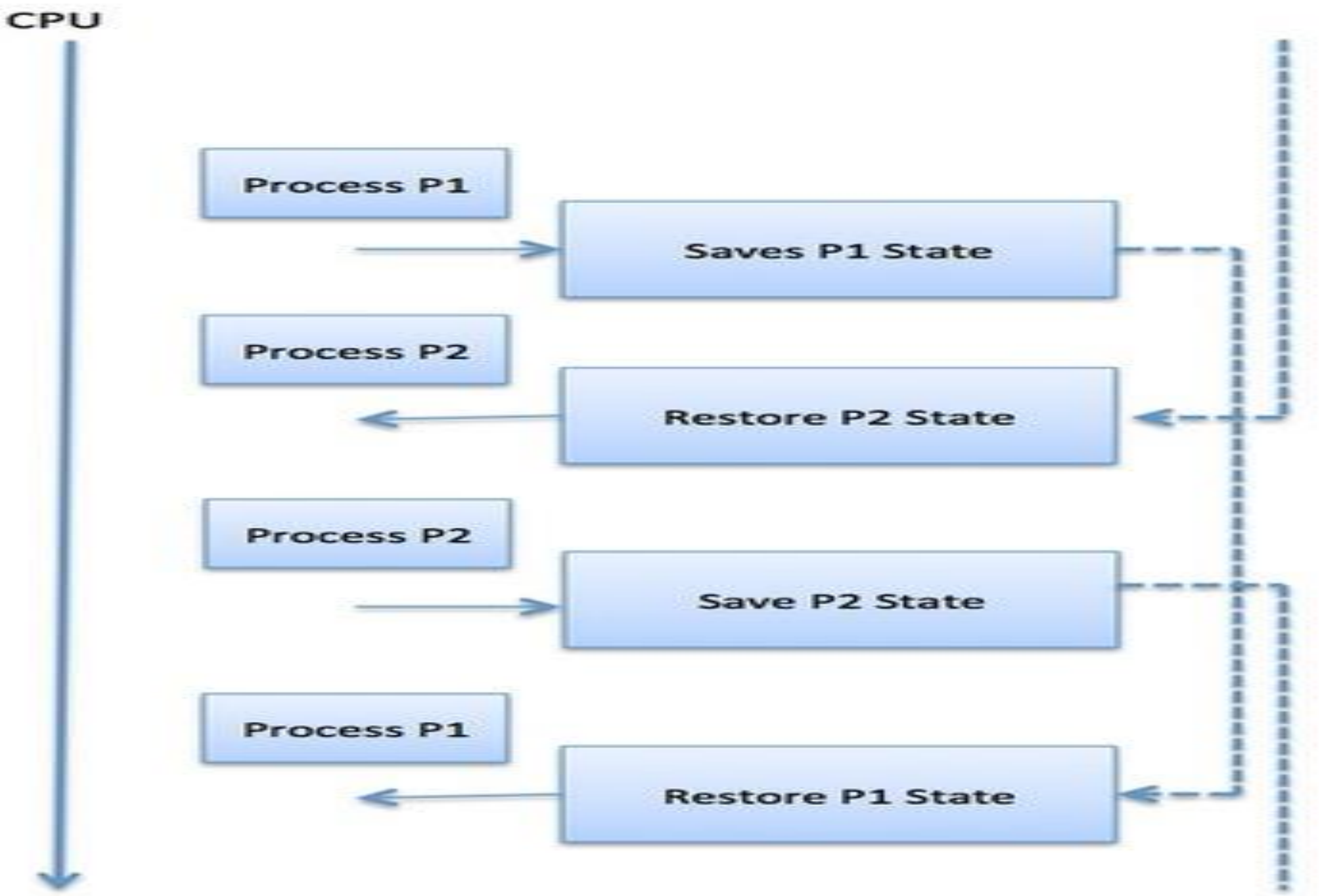
| S. N. | Long-Term Scheduler | Short-Term Scheduler | Medium-Term Scheduler |
|---|---|---|---|
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short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

## Context Switch

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.
- When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.
- Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

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

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


CPU
Process P1
Saves P1 State
Process P2
Restore P2 State
Process P2
Save P2 State
Process P1
Restore P1 State

## Operating System Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

There are six popular process scheduling algorithms which we are going to discuss in this chapter −

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

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

## Operating System Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

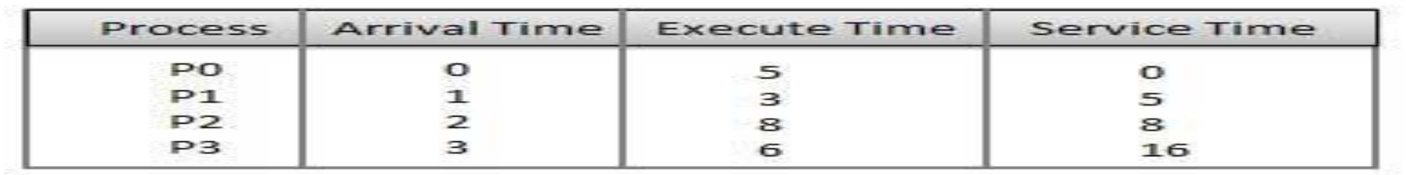
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**.

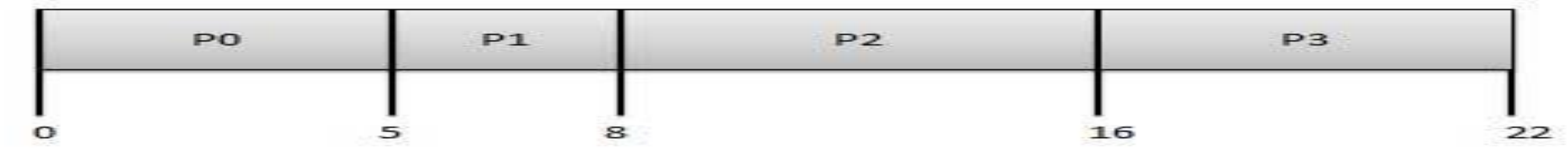
**Non-preemptive algorithms** are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the **preemptive scheduling** is based on priority where a scheduler may preempt a low priority running process anytime when a high priority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

## First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



| Process | Arrival Time | Execute Time | Service Time |
|---|---|---|---|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



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

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

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

## Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time :

| Process | Arrival Time | Execution Time | Service Time |
|---|---|---|---|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 14 |
| P3 | 3 | 6 | 8 |

| Process | Waiting Time |
|---|---|
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 14 - 2 = 12 |
| P3 | 8 - 3 = 5 |

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

## Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

| Process | Arrival Time | Execution Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 0 |
| P1 | 1 | 3 | 2 | 11 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 5 |

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

| Process | Waiting Time |
|---|---|
| P0 | 0 - 0 = 0 |
| P1 | 11 - 1 = 10 |
| P2 | 14 - 2 = 12 |
| P3 | 5 - 3 = 2 |

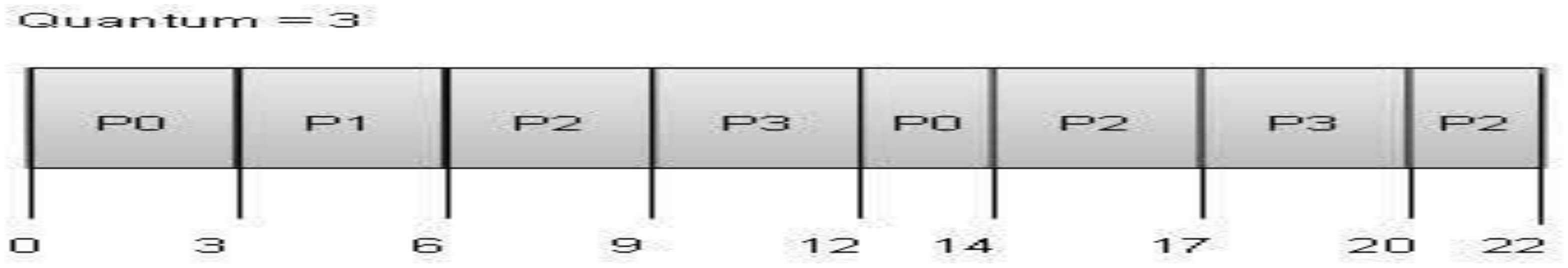
Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

## Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

## Round Robin Scheduling

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



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

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

## <u>Multiple-Level Queues Scheduling</u>

- Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.
- For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

## Different 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 −

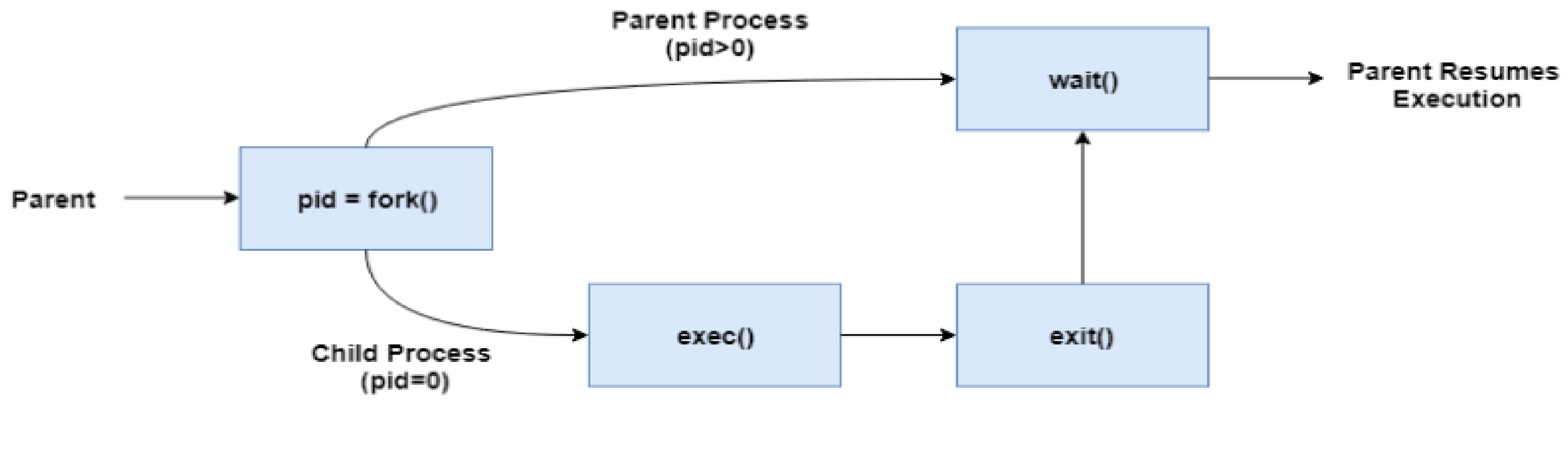
**Process Creation**

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()

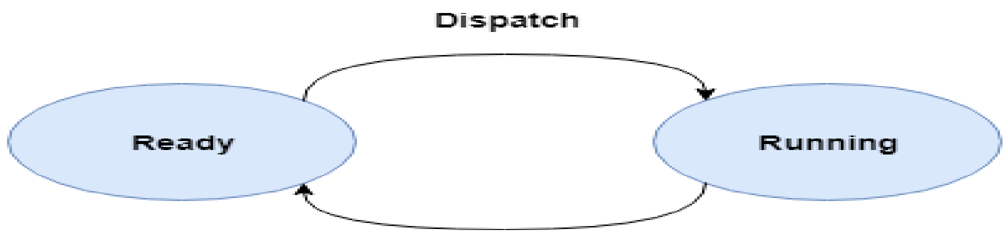
**Scheduling/Dispatching:**

- The event or activity in which the state of the process is changed from ready to running. It means the operating system puts the process from ready state into the running state.
- Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process. There are various other cases in which the process in running state is preempted and process in ready state is dispatched by the operating system.

**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 –

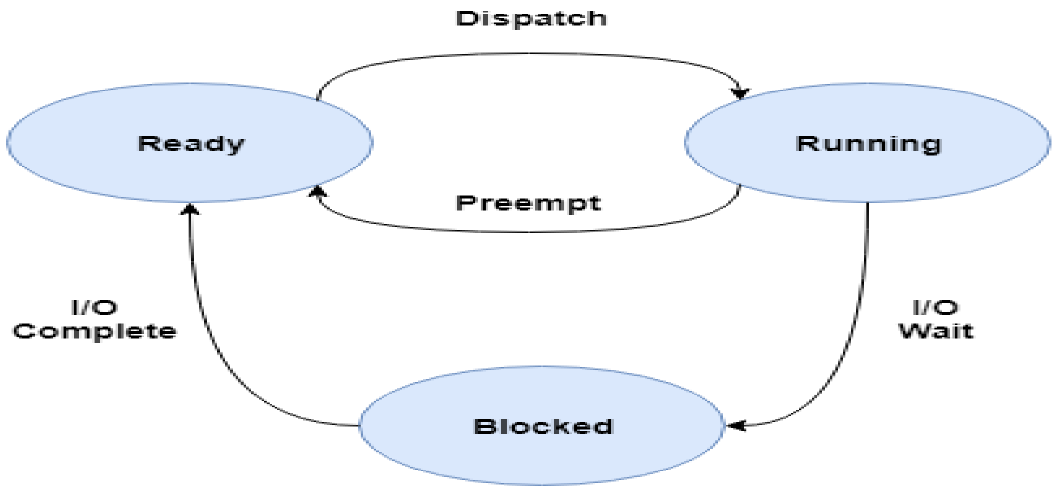


Process Preemption

**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.

## Cooperating Process

Cooperating processes are those that can affect or are affected by other processes running on the system. Cooperating processes may share data with each other.

**Reasons for needing cooperating processes**

There may be many reasons for the requirement of cooperating processes. Some of these are given as follows −

**Modularity**

- Modularity involves dividing complicated tasks into smaller subtasks. These subtasks can completed by different cooperating processes. This leads to faster and more efficient completion of the required tasks.

**Information Sharing**

- Sharing of information between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.

**Convenience**

- There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.

**Computation Speedup**

- Subtasks of a single task can be performed parallely using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

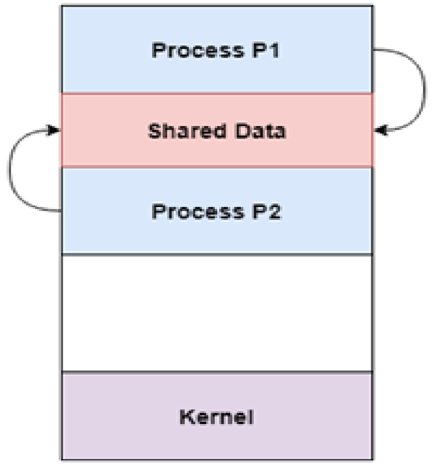
**Methods of Cooperation**

- Cooperating processes can coordinate with each other using shared data or messages. Details about these are given as follows −

**Cooperation by Sharing**

- The cooperating processes can cooperate with each other using shared data such as memory, variables, files, databases etc. Critical section is used to provide data integrity and writing is mutually exclusive to prevent inconsistent data.

A diagram that demonstrates cooperation by sharing is given as follows −

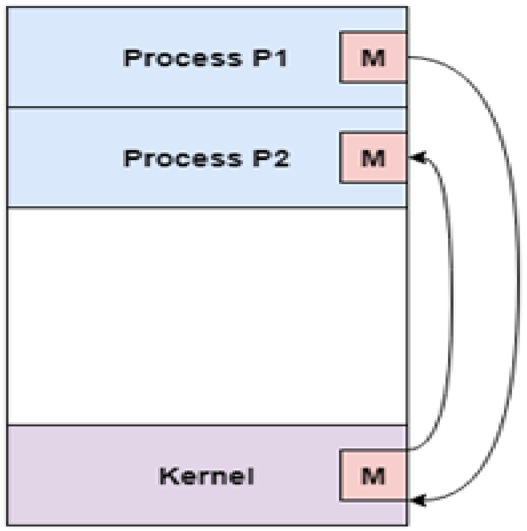


In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

**Cooperation by Communication**

- The cooperating processes can cooperate with each other using messages. This may lead to deadlock if each process is waiting for a message from the other to perform a operation. Starvation is also possible if a process never receives a message.

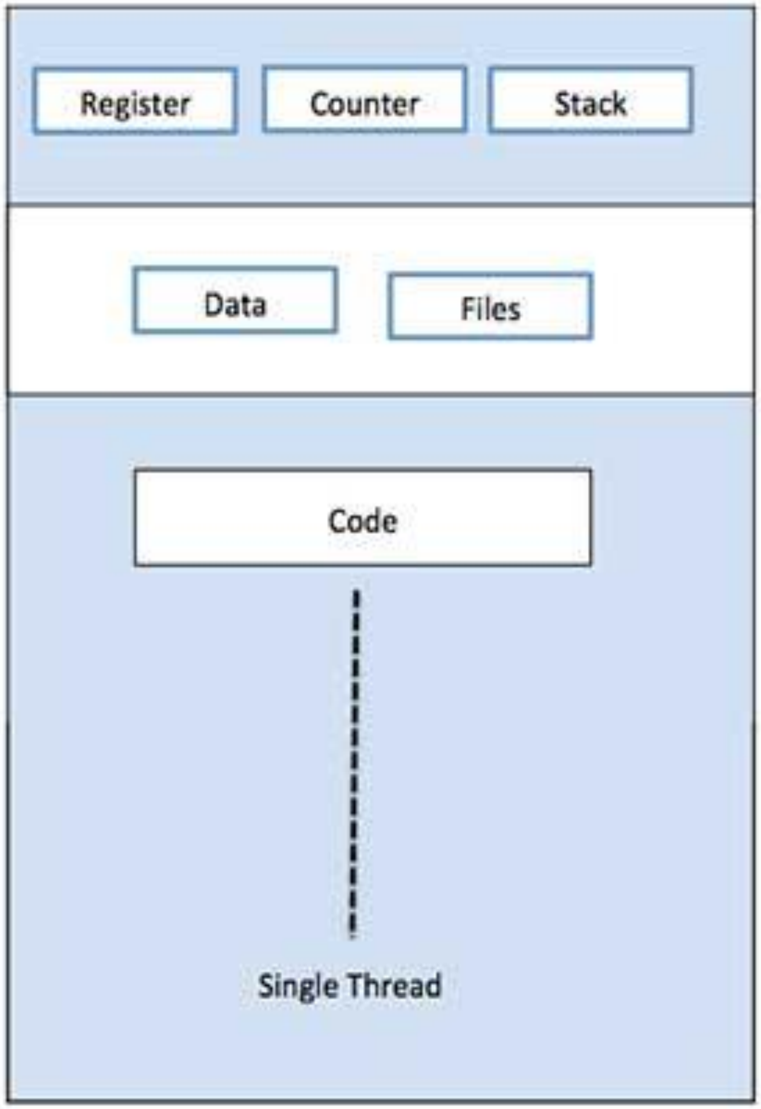
A diagram that demonstrates cooperation by communication is given as follows −



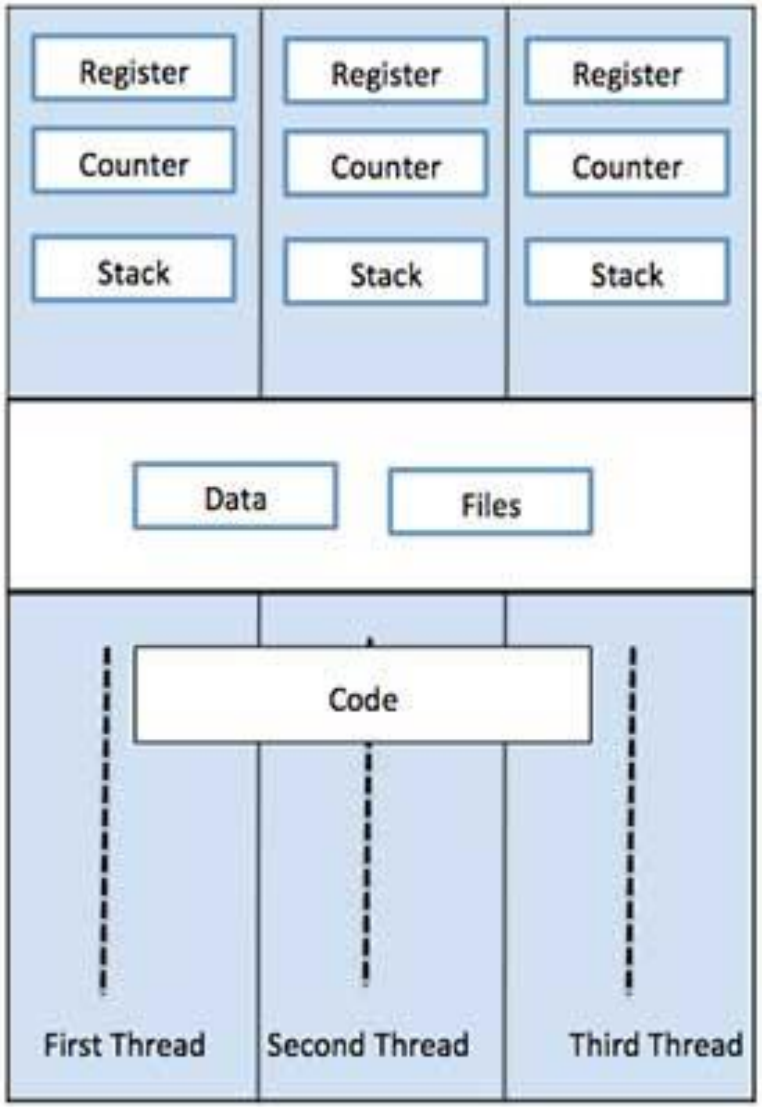
In the above diagram, Process P1 and P2 can cooperate with each other using messages to communicate.

## Thread

- 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.
- A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.
- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

| S. N. | Process | Thread |
|---|---|---|
| 1 | Process is heavy weight or resource intensive. | Thread is light weight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

## Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

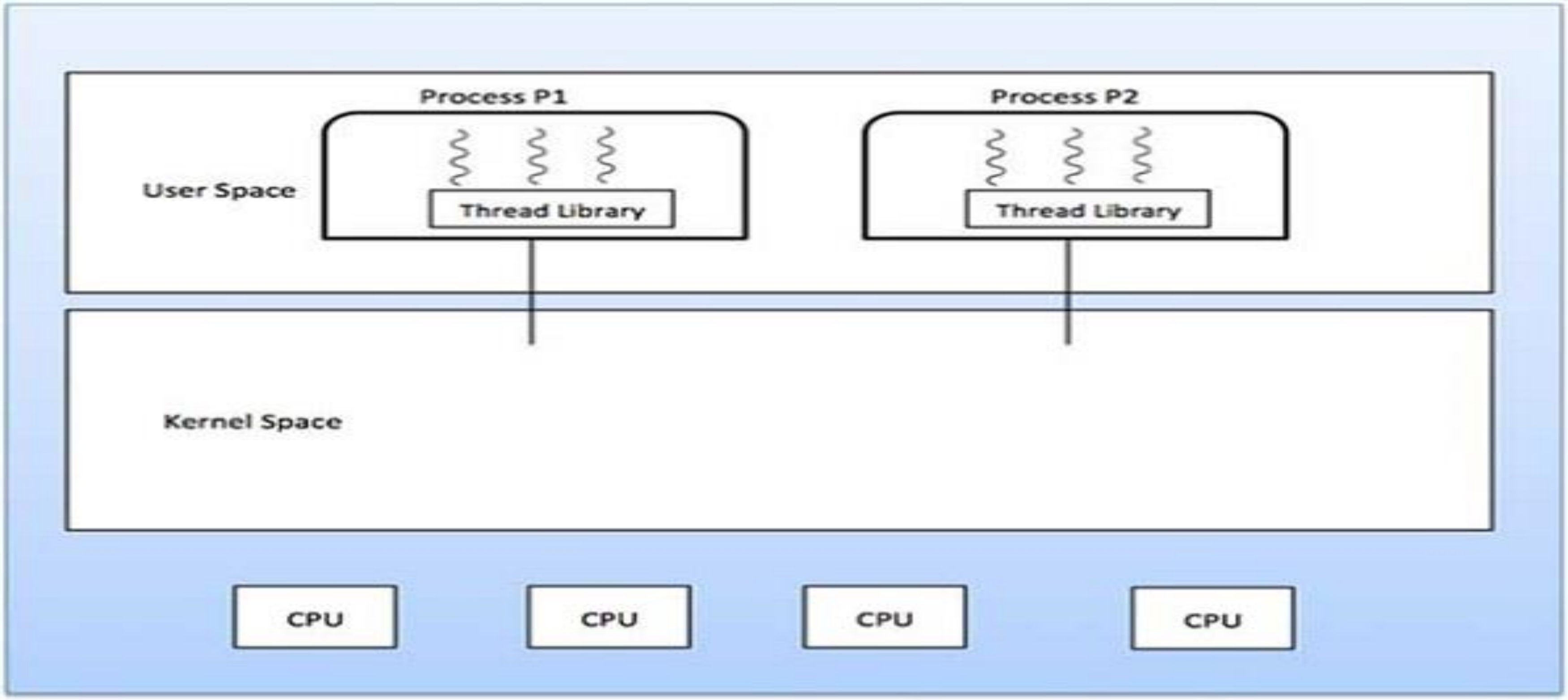
## Types of Thread

Threads are implemented in following two ways –

- **User Level Threads** – User managed threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.

## User Level Threads

- In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

- Thread switching does not require Kernel mode privileges and fast to create and manage.
- User level thread can run on any operating system  Scheduling can be application specific.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

**Kernel Level Threads**

- In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## **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 co-operating 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.

Processes can communicate with each other through both:

- Shared Memory
- Message passing

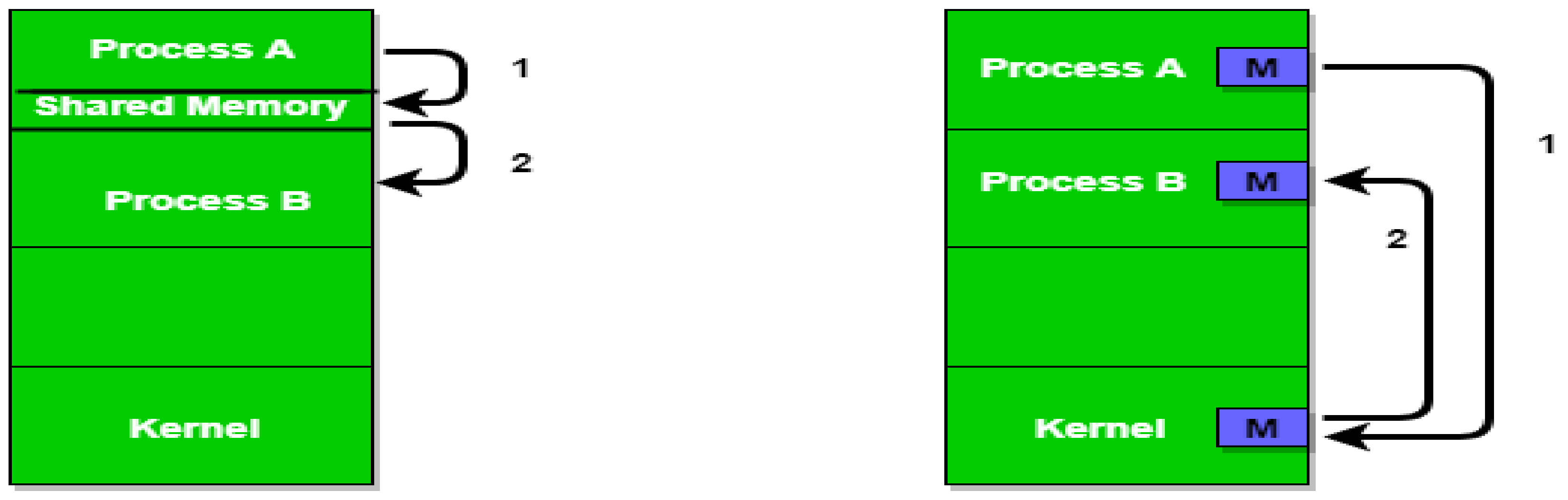


**Figure 1 -** Shared Memory and Message Passing

**i) Shared Memory Method**

- **Ex: Producer-Consumer problem**
  There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed.
- There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem.
- First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them.

## ii) Messaging Passing Method

- In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

  Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives. We need at least two primitives:
  – **send**(message, destination) or **send**(message)
  – **receive**(message, host) or **receive**(message)

**<u>Scheduling Criteria</u>**

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

<u>CPU Utilization</u>

- To make out the best use of the CPU and not to waste any CPU cycle, the CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

<u>Throughput</u>

- It is the total number of processes completed per unit of time or rather says the total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

<u>Turnaround Time</u>

- It is the amount of time taken to execute a particular process, i.e. The interval from the time of submission of the process to the time of completion of the process(Wall clock time).

<u>Waiting Time</u>

- The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

<u>Load Average</u>

- It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

<u>Response Time</u>

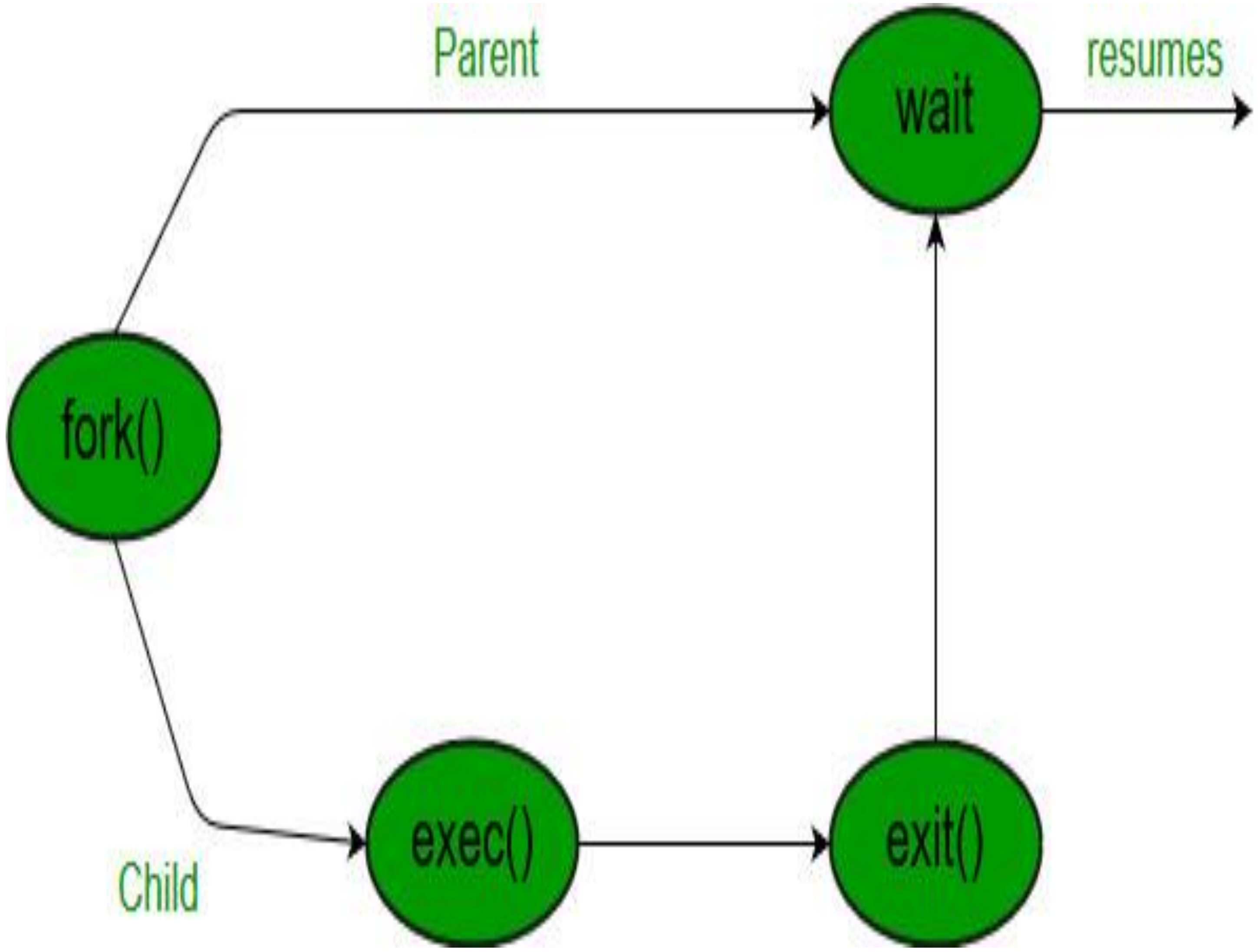
- Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).
- In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

## System Call

- When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.
- When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.
- Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.
- In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context, are described below.


Parent
wait
resumes
fork()
Child
exec()
exit()

## fork()

- Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork().
- ***Negative Value***: creation of a child process was unsuccessful.
  ***Zero***: Returned to the newly created child process.
  ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process

```
// example.c
 #include <stdio.h>
void main()
{
 int val; val = fork();   // line A
 printf("%d", val);    // line B
}
```

When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.

- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.
- This means that according to the above program, the output of parent process will be the **process ID** of the child process and the output of the child process will be 0.

## The “fork()” system call

- A process calling fork()spawns a child process.
- The child is almost an identical clone of the parent:
- Program Text (segment .text)
- Stack (ss)
- PCB (eg. registers)
- Data (segment .data)
- The fork()is called once, but returns twice!After fork()both the parent and the child are executing the same program.
- The “fork()” system call –PID
- pid<0:the creation of a child process was unsuccessful. pid==0: the newly created child.
- pid>0:the process ID of the child process passes to the parent.

```
void main()
{
Int i;
printf("simpfork: pid= %d\n", getpid());
i= fork();
printf("Did a fork. It returned %d.getpid= %d.
   getppid= %d\n“, i, getpid(), getppid());
}
```

Returns:

```
simpfork: pid= 914
Did a fork.Itreturned 915. getpid=914. getppid=381
Did a fork. It returned 0. getpid=915. getppid=914
```

When simpforkis executed, it has a pidof 914. Next it calls fork()creating a duplicate process with a pidof 915. The parent gains control of the CPU, and returns from fork()with a return value of the 915 --this is the child's pid.It prints out this return value, its own pid, and the pidof C shell, which is 381.Note:there is no guarantee which process gains control of the CPU first after a fork(). It could be the parent, and it could be the child.

## Wait System Call

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

**Syntax :**

// take one argument status and returns a process ID of dead children.

pid_t wait(int *stat_loc);

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);           /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);

    return 0;
}
```

Output :

```
Parent pid = 12345678
Child pid = 89546848
```

## Waitpid()

We know if more than one child processes are terminated, then wait() reaps any arbitrarily child process but if we want to reap any specific child process, we use ***waitpid()*** function.

**Syntax in c language:**

pid_t waitpid (child_pid, &status, options);

**Options Parameter**

If 0 means no option parent has to wait for terminates child.

If **WNOHANG** means parent does not wait if child does not terminate just check and return waitpid().(not block parent process)

If child_pid is **-1** then means any ***arbitrarily child***, here waitpid() work same as wait() work.

**Return value of waitpid()**

pid of child, if child has exited

0, if using WNOHANG and child hasn’t exited.

```
// C program to demonstrate waitpid()
#include<stdio.h>  #include<stdlib.h> #include<sys/wait.h> #include<unistd.h>
 Void  waitexample()
{
   int i, stat;    pid_t pid[5];
   for (i=0; i<5; i++)
   {
      if ((pid[i] = fork()) == 0)
      {          sleep(1);          exit(100 + i);        }
   }

   // Using waitpid() and printing exit status of children.
   for (i=0; i<5; i++)
   {
      pid_t cpid = waitpid(pid[i], &stat, 0);
      if (WIFEXITED(stat))    printf("Child %d terminated with status: %d\n", cpid, WEXITSTATUS(stat));
   }
}
 // Driver code
int main()
{
   waitexample();    return 0;
}
```

**Output:**

```
Child 50 terminated with status: 100   Child 51 terminated with status: 101
Child 52 terminated with status: 102 Child 53 terminated with status: 103
Child 54 terminated with status: 104
```

## **Exec()**

- The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.
- It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.
- The exec()call replaces a current process'image with a new one (i.e. loads a new program within current process).
- The new image is either regular executable binary file or a shell script.
- There are many flavors of exec() in UNIX, one being exec1() which is shown below as an example:

```
// example2.c
 #include <stdio.h>
 void main()
 {
execl("/bin/ls", "ls", 0); // line A
 printf("This text won't be printed unless an error occurs in exec().");
}
```

As shown above, the first parameter to the execl() function is the address of the program which needs to be executed, in this case, the address of the **ls** utility in UNIX.

Then it is followed by the name of the program which is **ls** in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).

- When the above example is executed, at line A, the **ls** program is called and executed and the current process is halted. Hence the printf() function is never called since the process has already been halted.
- The only exception to this is that, if the **execl()** function causes an error, then the printf() function is executed.

There’s nota syscallunder the name exec().

By exec()we usually refer to a family of calls:

int execl(char *path, char *arg, ...); intexecv(char *path, char *argv[]); intexecle(char *path, char *arg, ..., char *envp[]);

intexecve(char *path, char *argv[], char *envp[]);

intexeclp(char *file, char *arg, ...);

intexecvp(char *file, char *argv[]);

Where l=argument list, v=argument vector, e=environmental vector, and p=search path

## <u>exit()</u>

- This function is used for normal process termination.
- The status of the process is captured for future reference. There are other similar functions **exit(3)** and **_exit().**, which are used based on the exiting process that one is interested to use or capture.