UNIT-II:
Process Management

Process Concept: Process scheduling, Operations on processes, Inter-process communication.
Process Scheduling: Basic concepts, Scheduling criteria, Scheduling algorithms, Multiple processor scheduling, Thread scheduling.
Multithreaded Programming: Multithreading models, Thread libraries, Threading issues.

Process in Operating System:

- **A Process is a program in execution.**
- The process execution must progress in sequential fashion.
- A process is unit of work in most systems.
- A process is a program in execution which then forms the basis of all computation. The process is not as same as program code but a lot more than it.
- Program is ***passive*** entity stored on disk such as a file containing a list of instructions stored on disk (**executable file**).
- Process is ***active entity***
- Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
- A process is an 'active' entity as opposed to the program which is considered to be a 'passive' entity. Attributes held by the process include hardware state, memory, CPU, etc.

Process in Memory:

Process memory is divided into four sections for efficient working :

- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched. It also includes current activity including **program counter**, processor registers etc.
- The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared. Stack containing temporary data, Function parameters, return addresses, local variables

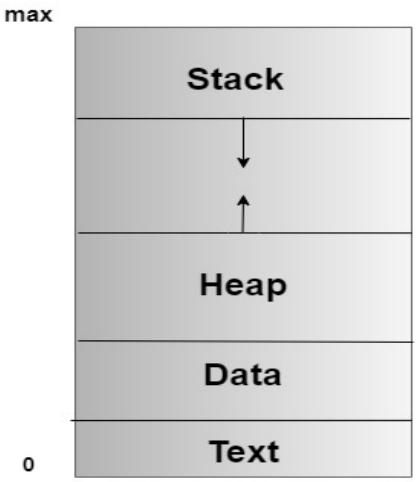


Figure: Process in the Memory

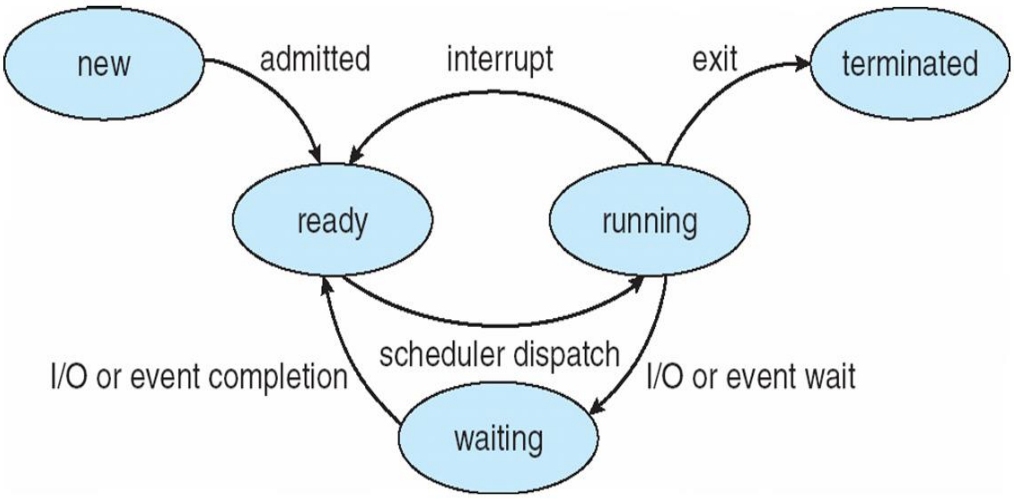
Differences between Process and Program

Process	Program
The process is basically an instance of the computer program that is being executed	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.
A process has a **shorter lifetime**.	A Program has a **longer lifetime**.
A Process requires resources such as memory, CPU, Input-Output devices.	A Program is stored by hard-disk and does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the **running instance** of the code.	On the other hand, the program is the **executable code**.

Process State:

- The process, from its creation to completion, passes through various states. The minimum number of states is five.
- The names of the states are not standardized although the process may be in one of the following states during execution.
- The state of a process is defined by the in part by the current activity of that process.
- Each process may be in any of the following states.

Diagram of Process State:



The states of Process are explained as:

1. New:

The process is being created.

2. **Ready**:

Ready state means the process is created and is waiting to be assigned to a processor. Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.

- The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.

3. **Running**:

When the Instructions of process are being executed, it is in running state. One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.

4. Waiting:

When the process is waiting for some event to occur, it is in Waiting state.

When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

5. **terminated**:

When a process has finished execution, it is Terminated state.

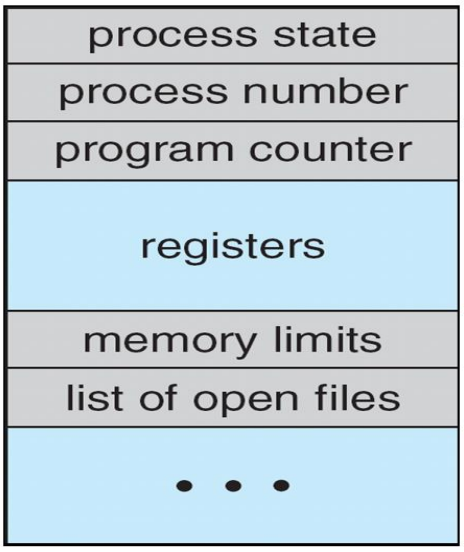
When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

- It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready or waiting.

*Process control block:

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.

Each process is shown in OS by a Process control block(PCB). (also called **task control block**)



The Attributes of the process are used by the Operating System to create the process control block (PCB) for each of them. This is also called context of the process. Attributes are the properties of process. The attributes which are stored in the PCB are described below.

1. **Process State**

The Process, from its creation to the completion, goes through various states which are new, ready, running and waiting.

2. **Process ID or Process number:**

When a process is created, a unique id is assigned to the process which is used for unique identification of the process in the system.

3. **Program counter**

A program counter stores the address of the next instruction to be executed in the process.

4. **General Purpose Registers**

Every process has its own set of registers which are used to hold the data which is generated during the execution of the process. This part stores contents of all process-centric registers

5. **CPU scheduling information:**

This part contains priorities, scheduling queue pointer and any other scheduling parameters.

6. **Memory-management information** :

This part contains memory allocated to the process. This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

7. **List of open files**

During the Execution, Every process uses some files which need to be present in the main memory. OS also maintains a list of open files in the PCB.

8. **List of open devices-** OS also maintain the list of all open devices which are used during the execution of the process.

9. **Priority**

Every process has its own priority. The process with the highest priority among the processes gets the CPU first. This is also stored on the process control block.

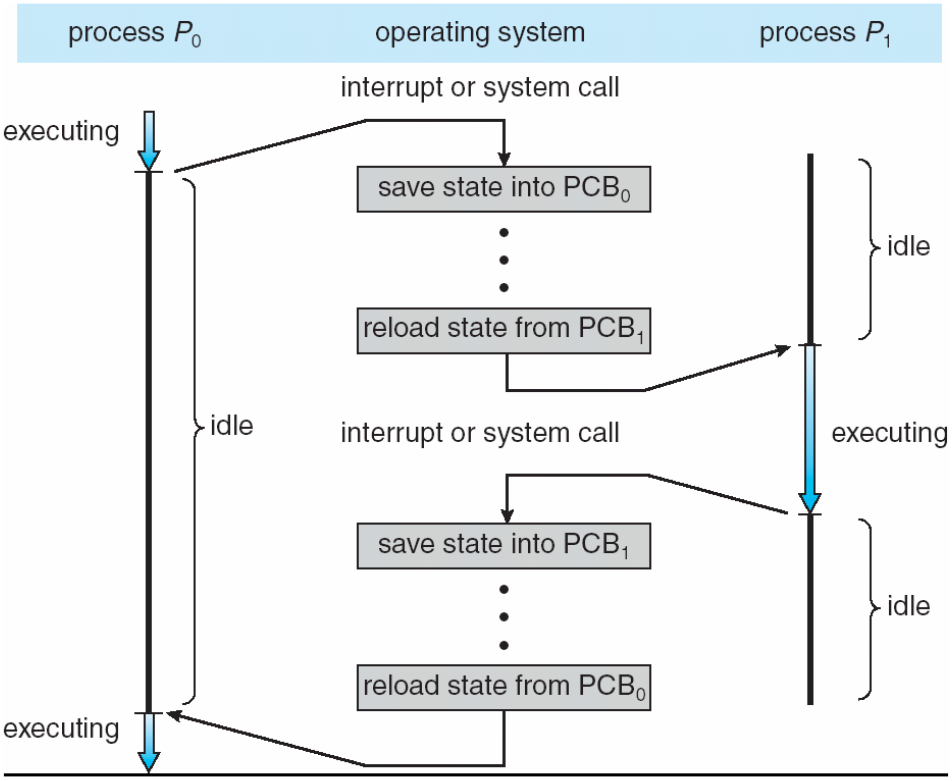
10. **Accounting information** –

This information includes the amount of CPU used and real time used, time limits, account numbers, job or process numbers, and so on.

11. **I/O status information** –

This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

CPU Switches From Process to Process:



When many processes are executing the CPU switches between the processes. When CPU switches from process to process, it stores the state information of first process when an interrupt occurs, to allow the process to be continued correctly afterward.

Process Scheduling:

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes

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.

1. Job queue –

It is the set of all processes in the system. In starting, all the processes get stored in the job queue. It is maintained in the secondary memory. The long term scheduler (Job scheduler) picks some of the jobs and put them in the primary memory.

2. Ready queue

It is the set of all processes residing in main memory, ready and waiting to execute. Processes in the Ready state are placed in the **Ready Queue.**

Ready queue is maintained in primary memory. The short term scheduler picks the job from the ready queue and dispatch to the CPU for the execution.

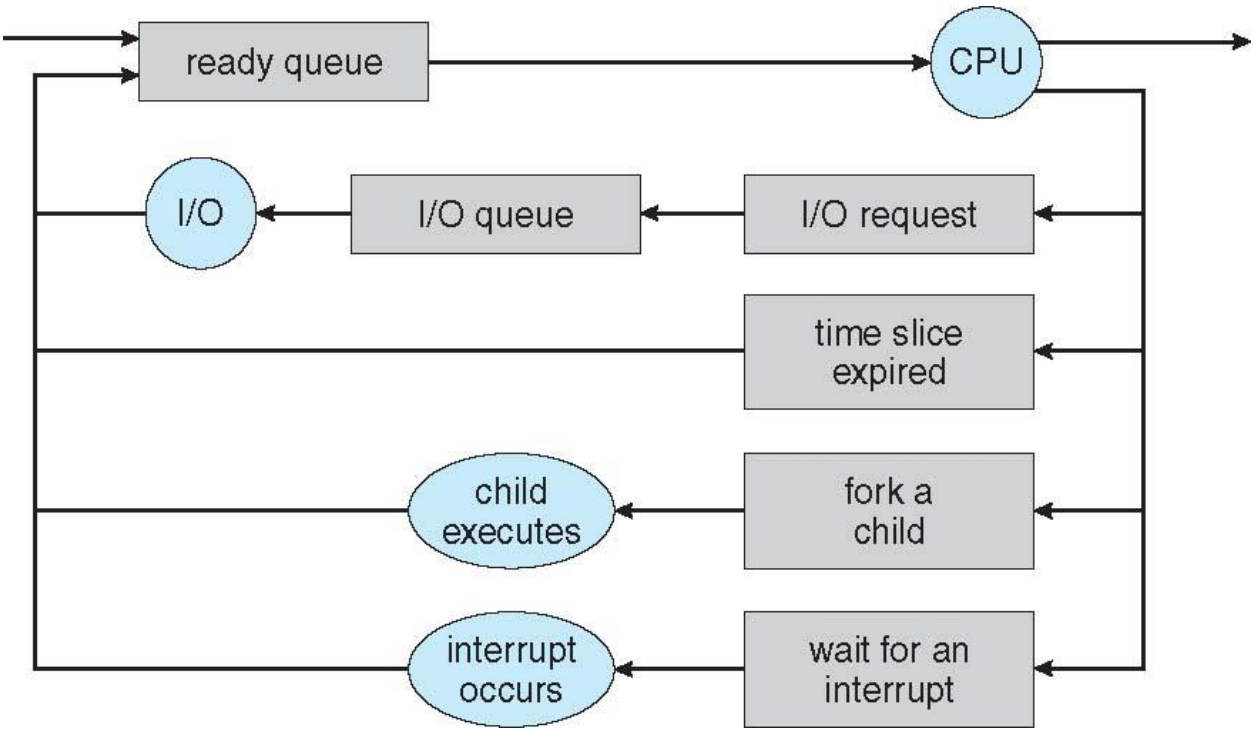
3. Device queues –

It is set of processes waiting for an I/O device. Processes waiting for a device to become available are placed in **Device Queues.** There are unique device queues available for each I/O device.

-When the process needs some IO operation in order to complete its execution, OS changes the state of the process from running to waiting. The context (PCB) associated with the process gets stored on the waiting queue which will be used by the Processor when the process finishes the IO.

Processes migrate among the various queues

Queueing-diagram representation of process scheduling.



The queueing diagram is shown in the above diagram. It can be explained as follows:

A common representation of process scheduling is a queueing diagram.

- 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,
- The arrows indicate the flow of processes in the system
- It represents queues, resources, flows.
- A new process is initially put in the Ready queue. It waits in the ready queue until it is selected for execution(or dispatched).

Once the process is assigned to the CPU and is executing, one of the following several events can occur:

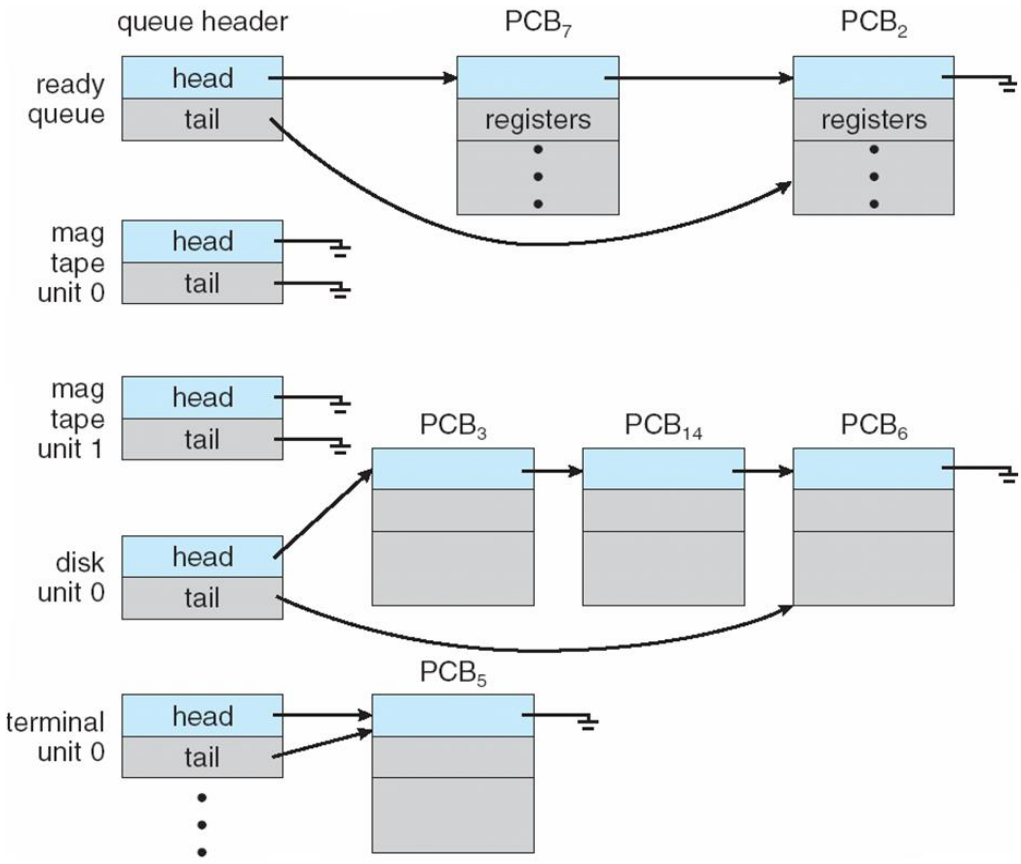
- The process could issue an I/O request, and then be placed in the I/O queue.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

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

Ready Queue and various I/O device Queues

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The ready queues is shown in below diagram.



Scheduler:

- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.

- The Process scheduler selects an available process (from set of several processes available) for execution on the CPU. In Single processor system, one executes, remaining waits until processor is free.

1. **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system
    - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)

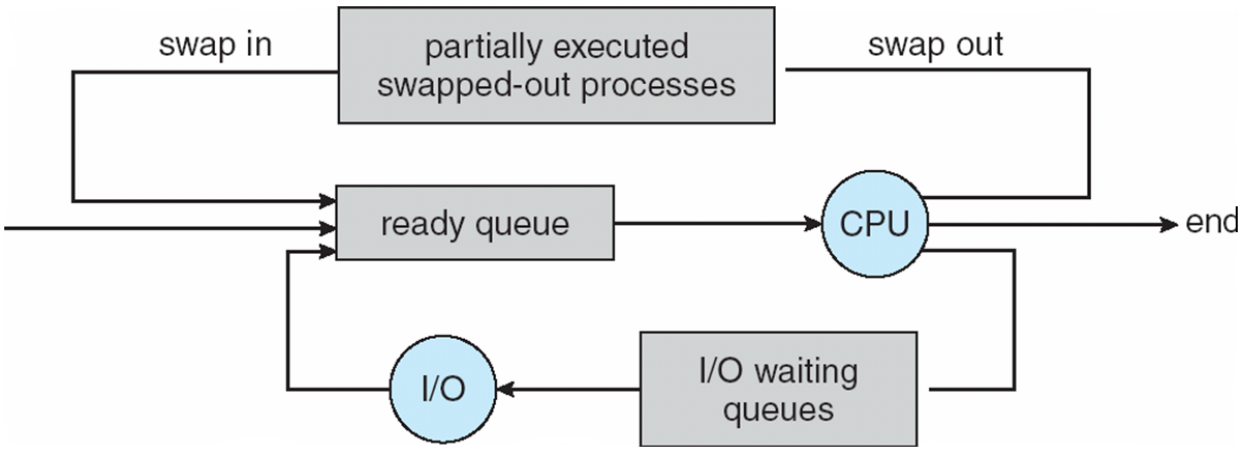
2. **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
    - Long-term scheduler is invoked infrequently (seconds, minutes) - (may be slow)
    - The long-term scheduler controls the **degree of multiprogramming**
    - Long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.

- Processes can be described as either:
    - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good process mix of I/O-bound and CPU-bound processes.

3. **Medium-term scheduler** can be added if degree of multiple programming needs to decrease

-Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

Addition of Medium Term Scheduling:



**Difference between the different types of schedulers:

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:

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- Context Switch – it means Switching the CPU to another process. It requires performing a state save of the current process and a state restore of a different process.
- **Context** of a process represented in the PCB. It includes the values of CPU registers, process state and memory management information.
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Context-switch times are highly dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

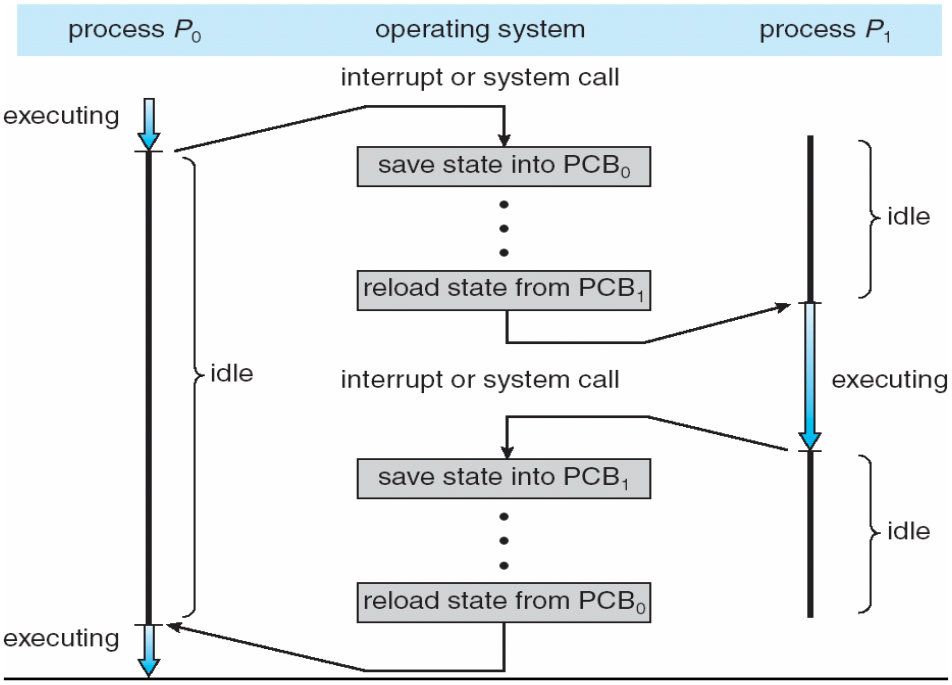
A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is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

The below diagram shows the CPU switch from process to process:



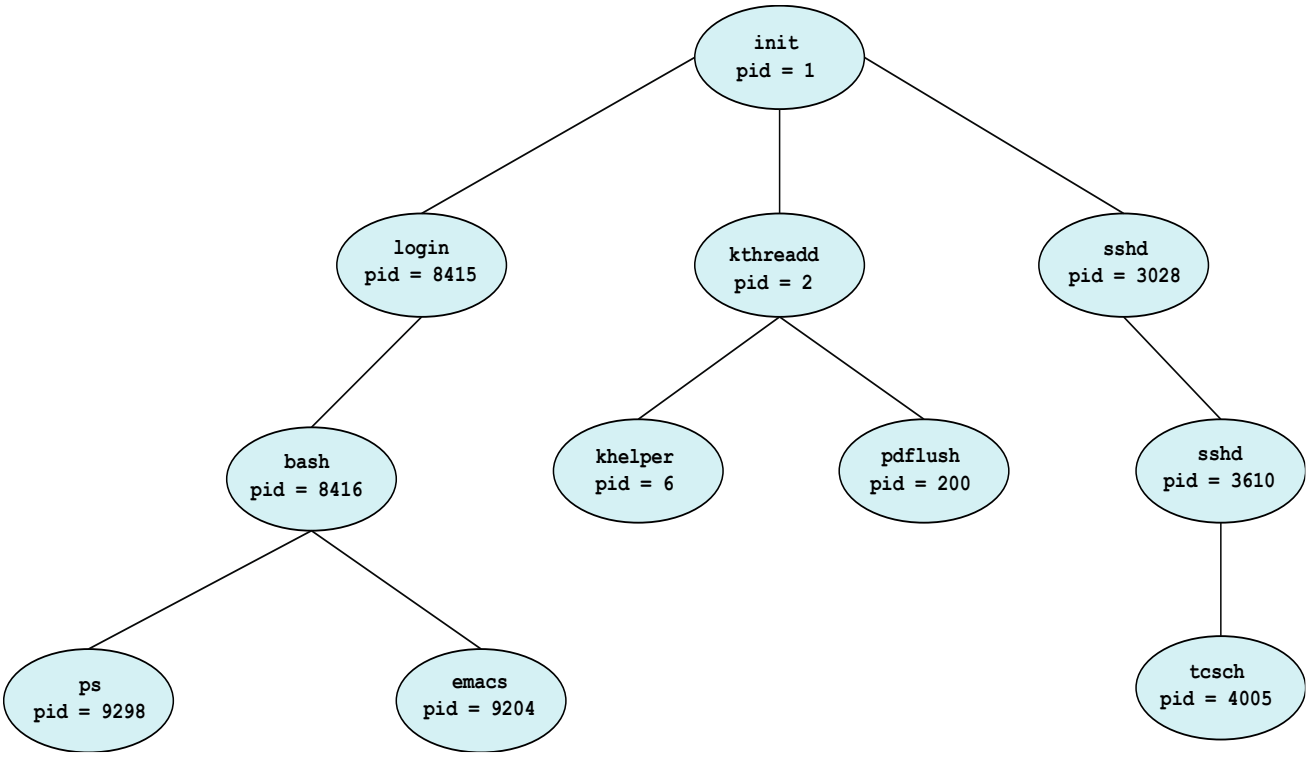
Operations on the Process:

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. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

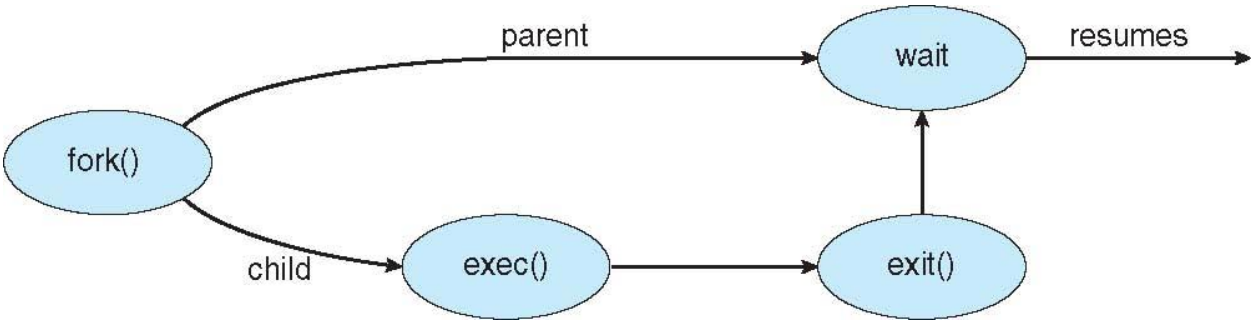
1. Process Creation:

- A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier(pid)
- Generally, process identified and managed via a **process identifier** (**pid**)
- When a process creates a new process, two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children.
  - The parent waits until some or all of its children have terminated.
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- There are also two possibilities in terms of the address space of the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

A tree of processes on a typical Solaris system:



- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program



The C program shown in Figure illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of pid (process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.

```c
#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;
}
```

2. 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).
- When process is terminated, All the resources of the process-including physical and virtual memory, open files, and I/0 buffers-are deallocated by the operating system.
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:

-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 mechanism to 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.

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process

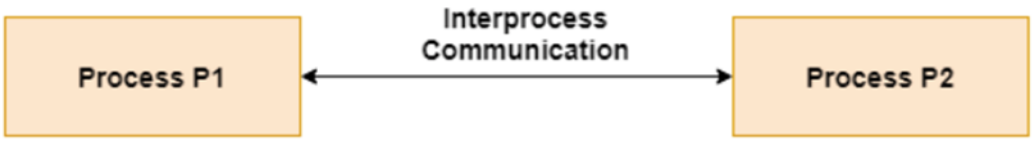
  pid = wait(&status);
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

What is Inter process Communication?

Inter process 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.

A diagram that illustrates interprocess communication is as follows −



- Processes executing concurrently in Operating system may be ***independent*** or ***cooperating.***
- ***Independent*** process are the processes which cannot affect or be affected by the execution of another process
- **Cooperating** process are processes which can affect or be affected by other processes. Any process that shares data with other processes is a Cooperating Process.

- Reasons for cooperating processes:
    1. 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.
    2. 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.
    3. Modularity- We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,
    4. 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.
- Cooperating processes need **inter process communication** (**IPC**) that will allow to them to exchange data and information.
- There are Two models of Interprocess Communication (IPC) :
    1. **Shared memory**
    2. **Message passing**
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

1. Shared Memory:

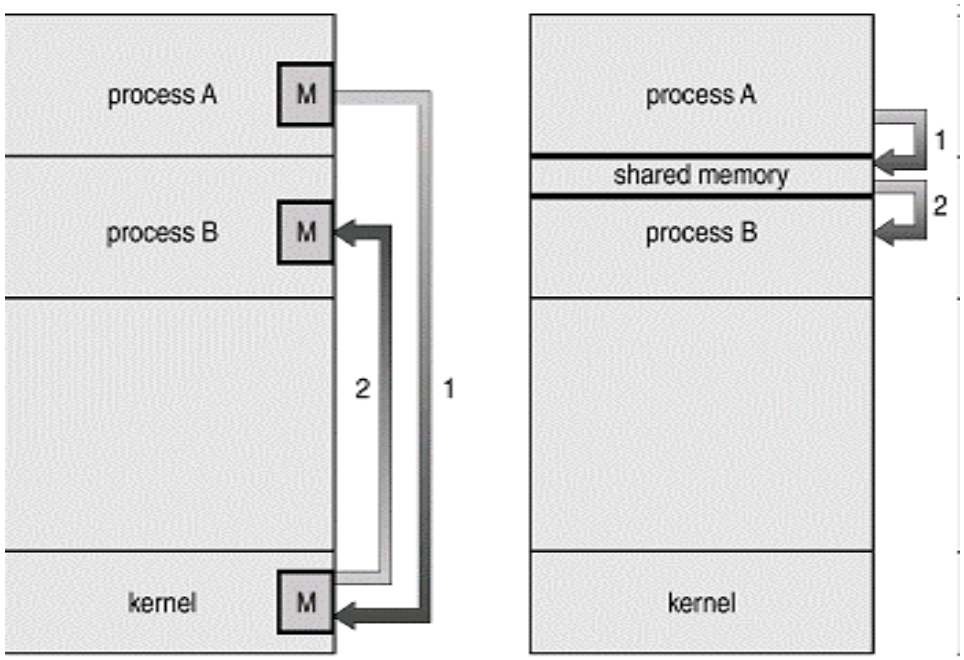
- Shared memory is an area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. 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.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

2. Message Passing:

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables

- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

Communications models. (a) Message passing. (b) Shared memory



(a)Message passing (b) Shared Memory

In Message Passing, a communication link must exist between them. This link can be implemented in a variety of ways.

- Direct or indirect communication
- Synchronous or asynchronous communication

Processes that want to communicate must have a way to refer to each other.

They can use either direct or indirect communication.

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.

Properties of Communication Link

• 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.

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate.

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Cpu Scheduling: Basic concepts:

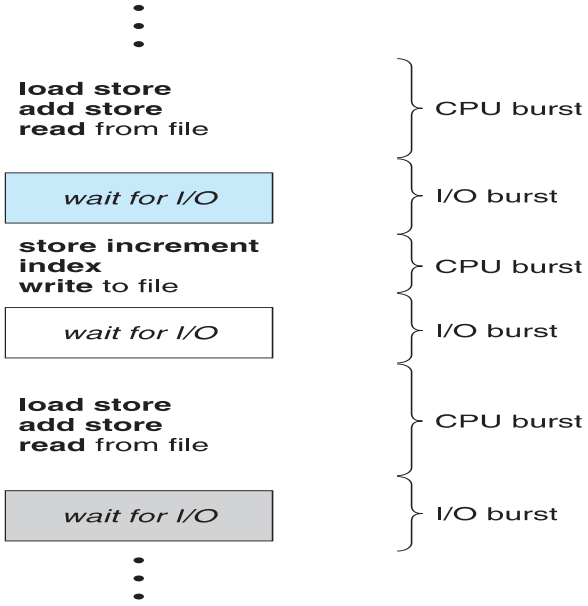
The objective of multiprogramming is to have some process rum1ing at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time.

When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait,

another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU-i/O Burst Cycle :

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/0 wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/0 burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution



CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
    1. Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
    1. Switches from running to waiting state
    2. Switches from running to ready state

3. Switches from waiting to ready
4. Terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

Dispatcher:

- Another component involved in the CPU-scheduling function is the dispatcher.
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves the following:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency

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. Non-Preemptive Scheduling

- Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time) or blocks itself for I/O.
- Example: FCFS, Non -Preemptive SJF, Priority Algorithms

2. Preemptive Scheduling:

- Preemptive algorithms means Currently running process may be interrupted and moved to the ready state by OS and next process is executed. If a process is currently using the processor and a new process with a higher priority enters the ready list, the process on the processor should be removed and high priority process is executed.
- Example: Preemptive SJF(SRTF), Preemptive Priority, Round Robin

Scheduling Criteria:

Many criteria are there for comparing CPU scheduling algorithm. Some criteria required for determining the best algorithm are given below.

- **CPU utilization** – keep the CPU as busy as possible. The range is about 40% for lightly loaded system and about 90% for heavily loaded system.
- **Throughput** – The number of processes that complete their execution per time unit.
- **Turnaround time** – The interval from the time of submission of a process to the time of completion is the Turnaround time.
- **Waiting time** – It is the amount of time a process has been waiting in the ready queue (or) the sum of the periods spent waiting in the ready queue.
- **Response time** – It is the amount of time it takes from when a request was submitted until the first response is produced, not the output (for time-sharing environment) is the response time.

Best Algorithm consider following:

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Formulas to calculate Turn-around time & waiting time is:

Waiting time = Finishing Time – (CPU Burst time + Arrival Time)

Turnaround time = Waiting Time + Burst Time

If Arrival times are given, then we should use the following formulas:

Waiting time = Starting Time –Arrival time

Turnaround time = Finishing Time –Arrival Time

1. FCFS Scheduling Algorithm:

FCFS stands for **First come first serve** (FCFS) scheduling algorithm. It simply schedules the jobs according to their arrival time. The process which comes first in the ready queue will execute in the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

Advantages of FCFS

- Simple
- Easy

- First come, First serve

Disadvantages of FCFS

1. The scheduling method is non preemptive, the process will run to the completion.
2. Due to the non-pre-emptive nature of the algorithm, the problem of starvation may occur.
3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

Example: Take the following processes P1, P2, P3 and Apply FCFS Scheduling algorithm.

Suppose that the processes arrive in the order: P_1 ,P_2 , P_3

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time for P_1 = 24; P_2 = 27; P_3 = 30
- Average Turnaround time: (24+27+30)/3=27

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

The Gantt chart for the schedule is:

- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3

- Average waiting time: (6 + 0 + 3)/3 = 3
- Turnaround time for P_1 = 30; P_2 = 3; P_3 = 6
- Average turnaround time: (30 + 3 + 6)/3 = 13
- Much better than previous case

Example 2:

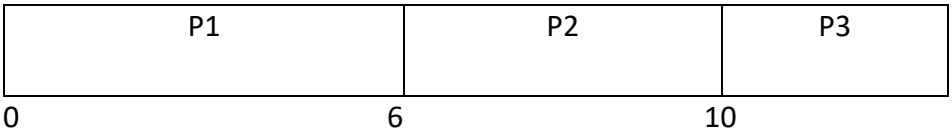
Some times Processes are given arrival times also. Then we can calculate the waiting times and turn around times using the following way given below:

Example: Take the following processes P1, P2, P3 and Apply FCFS Scheduling algorithm.

Suppose that the processes arrive in the order: P_1 ,P_2 , P_3

Process	Arrival time	Burst Time
P_1 P2 P3	0 1 2	6 4 3

The Gantt Chart for the schedule is:



We can calculate the Waiting times and Turn around times by using following formulas:

Waiting time= Starting time – Arrival Time

Turn Around time = Completion Time – Arrival Time

Process	Arrival time	Burst Time	Start time	Completion time	Waiting time	Turn-around time
P_1 P2 P3	0 1 2	6 4 3	0 6 10	6 10 13	0 5 8	6 9 11

- Average Waiting time: (0+ 5+ 8)/3= 4.33
- Average Turnaround time: (6+ 9+ 11)/3=8.66

Shortest Job First (SJF) Scheduling algorithm:

- In SJF scheduling algorithm, schedules the processes according to their burst time.
- In SJF scheduling, the process with smallest burst time will be executed first.
- The process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.
- SJF is optimal – gives minimum average waiting time for a given set of processes
- It is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

Advantages of SJF:

1. Maximum throughput
2. Minimum average waiting and turnaround time

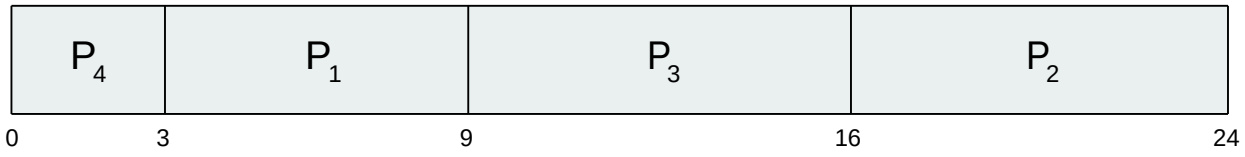
Disadvantages of SJF:

1. May suffer with the problem of starvation
2. It is not implementable because the exact Burst time for a process can't be known in advance.

Example for SJF:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Waiting time for P_1 = 3; P_2 = 16; P_3 = 9 ; P_4 = 0
- Average waiting time: (3+16+9+0)/4 = 7
- Turnaround time for P_1=9; P_2=24;
  P_3 =16;P_4 =3
- Average Turnaround time: (9+24+16+3)/4=13

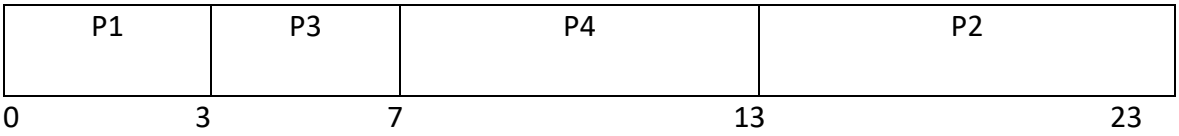
Example 2:

Some times Processes are given arrival times also. Then we can calculate the waiting times and turn around times using the following way given below: Take below example:

The following Process

Process	Arrival time	Burst Time
P_1 P2 P3 P4	0 1 2 3	3 10 4 6

- SJF scheduling Gantt chart



- If arrival times are given, then we can calculate the Waiting times and Turn around times by using following formulas:

Waiting time= Starting time – Arrival Time

Turn Around time = Completion Time – Arrival Time

Process	Arrival time	Burst Time	Start time	Completion time	Waiting time	Turn-around time
P_1 P2 P3 P4	0 1 2 3	3 10 4 6	0 13 3 7	3 23 7 13	0 12 1 4	3 22 5 10

- Average waiting time: (0+12+ 1 +4)/4 = 4.25
- Average Turnaround time: (3 + 22 + 5+ 10)/4= 10

Preemptive SJF Algorithm (or) Shortest-remaining-time-first Algorithm(SRTF):

Now we add the concepts of varying arrival times and preemption to the analysis to SJF. Then it is called Shortest-remaining-time-first.

Example of Shortest-remaining-time-first:

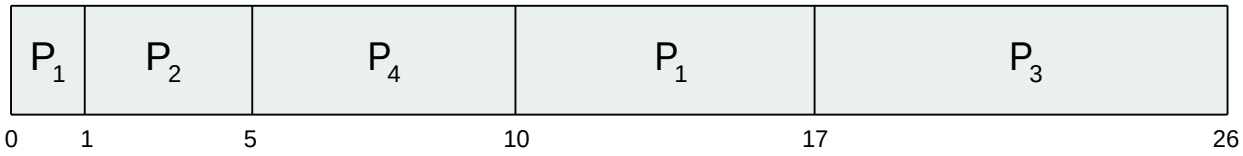
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. Now calculate Remaining time as shown below:

Process	Arrival Time	Burst Time	Remaining time
P_1	0	8	7
P2	1	4	4
P3	2	9	9
P4	3	5	5

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

Preemptive SJF Gantt Chart



Average waiting time = [(10-1-0)+(1-0-1)+(17-0-2)+(5-0-3)]/4 = 26/4 = 6.5 msec

Average Turnaround time=[(9+8)+(0+4)+(15+9)+(2+5)]/4=13

Priority Scheduling:

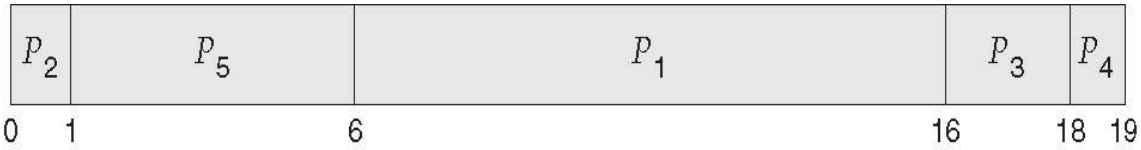
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- The priority algorithms are two types
  - Preemptive
  - Non preemptive

- 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.
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process
- There are two types of priority scheduling algorithm exists. One is Preemptive priority scheduling while the other is Non Preemptive Priority scheduling.
- A preemptive priority algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority algorithm will simply put the new process at the head of the ready queue. That means , non preemptive scheduling does not interrupt a process running in the CPU in the middle of execution. Instead it waits till the process completes its burst time, and then can allocate CPU to another process.

Example:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Waiting time for processes

  P_1 =6 ;
  P_2 = 0;
  P_3 = 16 ;
  P_4 =18 ;
  P_5=1
- Average waiting time: (6+0+16+18+1)/5 = 41/5 =8.1
- Turnaround time for processes= Waiting Time + Burst time

  P_1=16;
  P_2=1;

P_3 =18;
P_4 =19;
P_5=6

- Average Turnaround time:
  (16+1+18+19+6)/5=60/5 =12
- If arrival times are given , We can calculate the Waiting times and Turn around times by using following formulas:

Waiting time= Starting time – Arrival Time
Turn Around time = Completion Time – Arrival Time

(b) Round Robin Scheduling algorithm:

- Round Robin is the preemptive process scheduling algorithm.
- RR is similar to FCFS, but preemption is added to switch between the processes.
- Each process is provided a fix time to execute, it is called a **time quantum** or **time slice.** usually 10-100 milliseconds.
- Once a process is executed for a given time slice, it is preempted and other process executes for a given time slice. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Processes are stored in FIFO queue.
- Context switching is used to save states of preempted processes.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Advantages:

1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

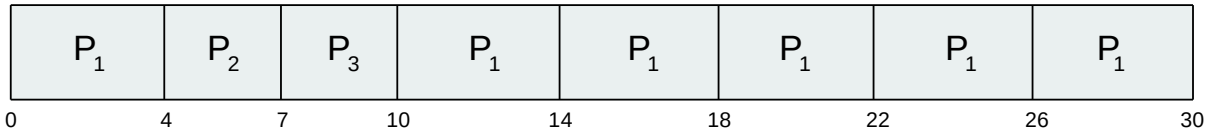
Disadvantages:

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.

Example: Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



- Waiting time for processes

  P_1 = 10-4=6;

  P_2 = 4;

  P_3 = 7

- Average waiting time: (6+4+7)/3 =17/3
- Turnaround time for processes
  - P_1=30;

    P_2=7;

    P_3 =10
- Average Turnaround time: (30+7+10)/3=47/3

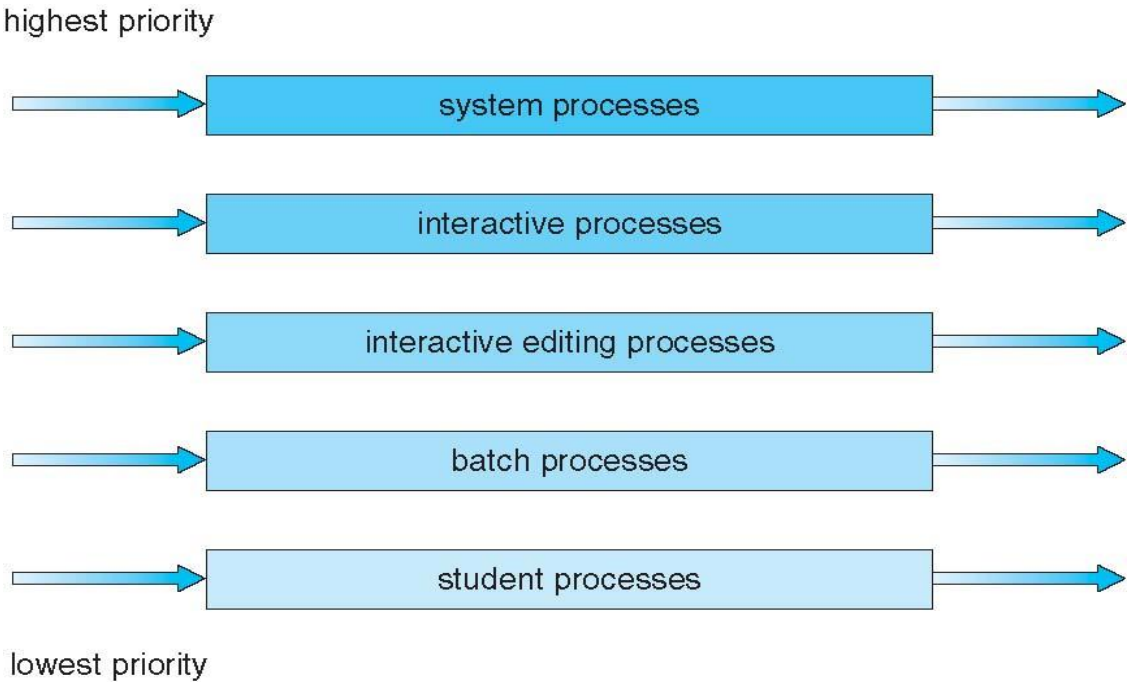
Problem:

. Explain the FCFS, preemptive and non-preemptive versions of Shortest Job First and Round Robin (time-slice2) scheduling algorithms with Gantt Chart for the four processes given. Compare their average turn around and wait time.

Process	Arrival Time	Burst time
P1	0	10
P2	1	6
P3	2	12
P4	3	15

Multilevel Queue Scheduling:

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.For example, a common division is made between
  - Foreground (interactive) processes and
  - Background (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.
- The multilevel queue scheduling algorithm partitions the **ready queue** into several separate queues.
- The process are permanently assigned to one queue, based on some property of the process such as memory size, process priority, or process type.
- For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
- For example, the foreground queue may have absolute priority over the background queue.
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS
- In this, commonly implemented as fixed – priority preemptive scheduling.



Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

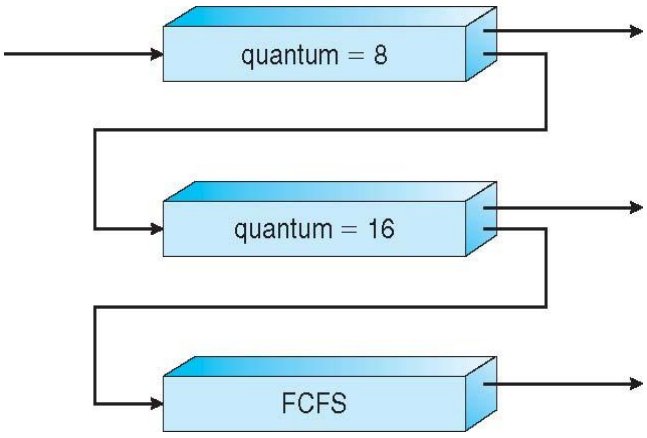
Multilevel Feedback Queue:

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter a system. If there are separate queues for foreground and background processes, processes do not move from one queue to other queue
- This setup has advantage of low scheduling overhead, but it is inflexible.
- Multilevel feedback queue scheduling algorithm, in contrast allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to lower priority queue.
- Multilevel-feedback-queue scheduler 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
- This scheme leaves I/O bound and interactive processes in the higher priority queues.
- In addition , a process that waits too long in a low-priority queue may be moved to a high priority queue.
- This form of aging prevents starvation.

Example of Multilevel Feedback Queue:

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Muti-level Feedback Queue

Threads:

- A thread is a basic unit of CPU utilization.
- It comprises of a thread ID, a program counter, a register set, and a stack.

- It shares with the other threads of same process its code section, data section and other operating system resources such as open files and signals.
- If a process has multiple threads of control, it can perform more than one task at a time.
- Within an address space, we can have more units of execution: threads
- All the threads of a process share the same address space and the same resource.

Need of Thread:

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

Process Vs Threads:

- Process
  - A process is a program in execution
  - all resources allocated: IPC channels, files etc..
  - a virtual address space that holds the process image
- Threads
  - A thread is light weight process.
  - a dispatchable unit of work
  - an execution state: running, ready, etc..
  - an execution context: PC, SP, other registers
  - a per-thread stack

Differences between Processes and Threads:

Process	Thread
A process can be defined as a program in execution.	A thread can be defined as the flow of execution via the process code.
In the process, switching requires interaction with the operating system.	In thread switching, there is no requirement to interact with the operating system.
It is heavyweight.	It is lightweight.

In a process, if a process is blocked due to some reasons, then the other processes cannot be executed until the process which is blocked will not be unblocked.	In a thread, if one thread is blocked, then the other thread is able to do the same task.
The Process consumes more resources.	Thread consumes fewer resources.
Context switching requires more time in process.	Context switching requires less time in thread.
The Process needs more time for termination.	The Thread takes less time for termination.
The Process takes more time for creation	The Thread takes less time for execution.
In terms of communication, the process is less efficient.	In terms of communication, the thread is more efficient.
In Process switching, the interface in the operating system is used.	In thread switching, no need to call an operating system.
It is Isolated.	It shares memory.

Types of threads:

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

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

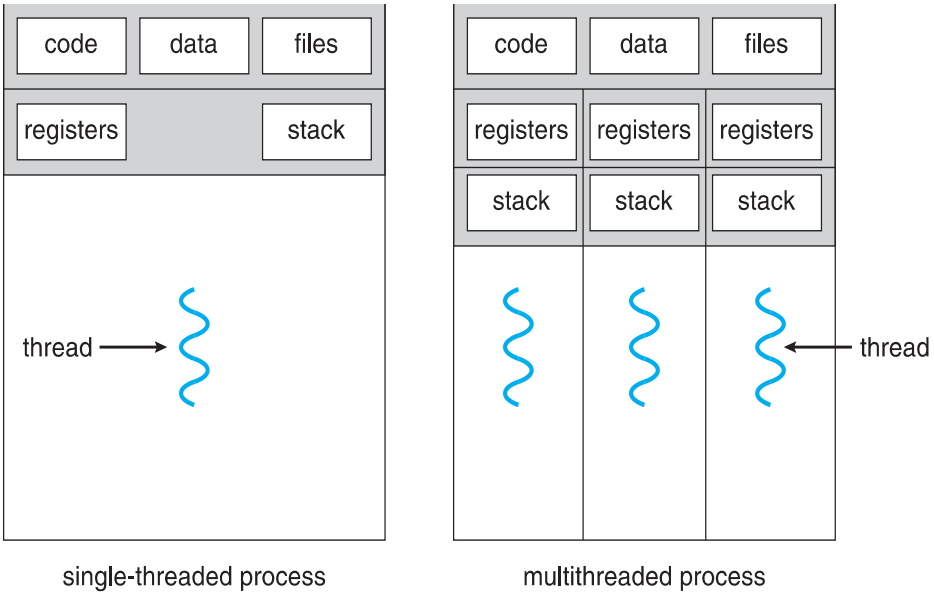
- User-level threads
  - User threads are implemented by a thread library, which contains the code for thread creation, termination, scheduling, and switching
  - User threads are above the kernel and without kernel support. These are the threads that application programmers use in their programs.
  - Kernel sees one process and it is unaware of its thread activity
  - can be preemptive or not (co-routines)
  - Three primary thread libraries:

    - POSIX **Pthreads**
    - Windows threads
    - Java threads
- Kernel-level threads
    - The kernel threads management done by the kernel
    - Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel-level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
    - Examples – virtually all general purpose operating systems, including:
    - Windows , Solaris, Linux, Mac OS X

User Level threads	Kernel Level Threads
These threads are implemented by users.	These threads are implemented by Operating systems
These threads are not recognized by operating systems,	These threads are recognized by operating systems,
In User Level threads, the Context switch requires no hardware support.	In Kernel Level threads, hardware support is needed.
These threads are mainly designed as dependent threads.	These threads are mainly designed as independent threads.
In User Level threads, if one user-level thread performs a blocking operation then the entire process will be blocked.	On the other hand, if one kernel thread performs a blocking operation then another thread can continue the execution.
Example of User Level threads: Java thread, POSIX threads.	Example of Kernel level threads: Window Solaris.
Implementation of User Level thread is done by a thread library and is easy.	While the Implementation of the kernel-level thread is done by the operating system and is complex.
This thread is generic in nature and can run on any operating system.	This is specific to the operating system.

Single and Multi-Threaded process:

- A traditional (or heavy weight:) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. The below Figure illustrates the difference between a traditional Single threaded process and a Multi- threaded process.



single-threaded process multithreaded process

Motivation:

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads.
- For example : A Word processor may have several threads for
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Benefits:

The benefits of multithreaded programming can be broken down into four major categories:

- **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
  - Example: Web browser can allow to search in one thread while an image is being loaded in another thread.
- **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.
  - The 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.
- **Economy –**
  - Creating threads is cheaper than process creation, thread switching lower overhead than context switching
  - Allocating memory and resources for process creation is costly but threads is cheaper
- **Utilization of Multiprocessor architecture**
  - The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multi Threading models:

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. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship.

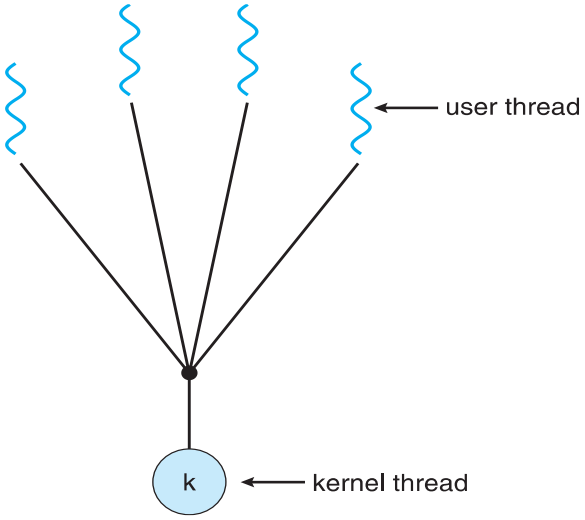
There are different multi-threading models:

- Many-to-One
- One-to-One
- Many-to-Many

1. Many to One Model:

- The many-to-one model maps many user-level threads to one kernel thread.

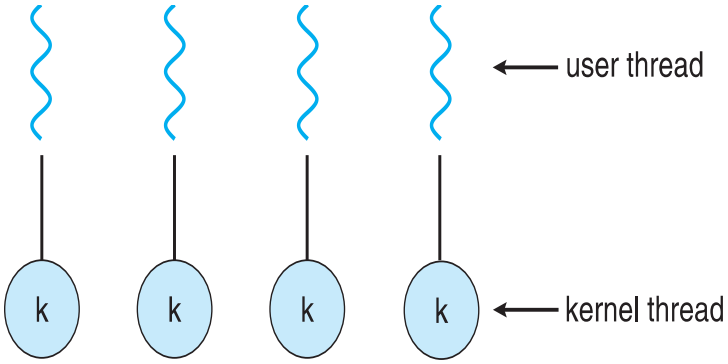
- Thread management is done by the thread library in user space, so it is efficient
- The entire process will block if one thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- A very few systems continue to use the model because of its inability to take advantage of multiple processing cores
- Example: Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.
- GNU Portable Threads also an example.



2. One to One Model:

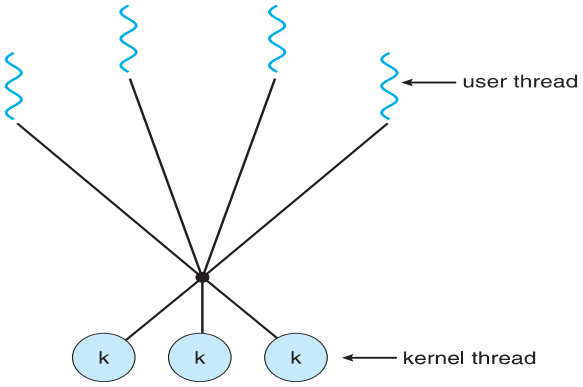
The one-to-one model maps each user thread to a kernel thread.

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- When one thread is blocking, it allows another thread to run.
- **Drawback -** creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one-to-one m
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



3. Many to Many model:

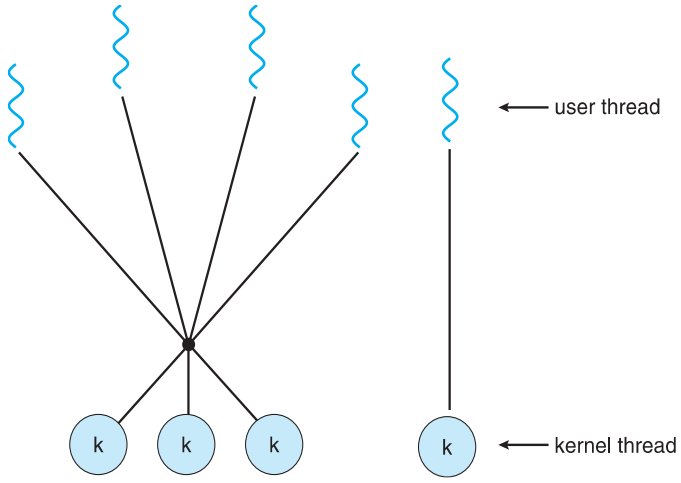
- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine
- Many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.
- **Drawback:**
  Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Two-level Model:

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the two-level model.

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - Solaris 8 and earlier



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 user space.
- 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:
    - (1) POSIX Pthreads,
    - (2) Win32, and
    - (3) Java.
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library. The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Win32 API; UNIX and Linux systems often use Pthreads.

Threading Issues in OS:

Following threading issues are:

- The fork() and exec() system call
- Signal handling
- Thread cancelation
- Thread local storage
- Scheduler activation

The fork() and exec() system call:

In case if a thread fork is the complete process copied or is the new process single threaded? The answer is here it depends on the system and in case of if new process execs immediately then there is no need to copy all the other thread and if it does not create new process then the whole process should be copied.

Signal Handling:

Whenever a multithreaded process receives a signal then to what thread should that signal be conveyed? There are following four main option for signal distribution:

1. Signal deliver to the thread to which the signal applies.
2. Signal deliver to each and every thread in the process.
3. Signal deliver to some of the threads in the process.
4. Assign a particular thread to receive all the signals in a process.

Thread Cancellation:

Terminating a thread before it has finished
Threads that are no-longer required can be cancelled by another thread in one of two techniques:

1. Asynchronous cancellation
2. Deferred cancellation

Asynchronous Cancellation

It means cancellation of thread immediately. Allocation of resources and inter thread data transfer may be challenging for asynchronies cancellation.

Deferred Cancellation

In this, the target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
In this method a flag is sets that indicating the thread should cancel itself when it is feasible. It's upon the cancelled thread to check this flag intermittently and exit nicely when it sees the set flag.

Thread Local Storage

The benefit of using threads in the first place is that Most data is shared among the threads but, sometimes threads also need thread explicit data. Major libraries of threads are pThreads, Win32 and java which provide support for thread specific which is called as TLS thread local storage.

Scheduler Activation

Numerous implementation of threads provides a virtual processor as an interface b/w user and kernel thread specifically for two tier model. The virtual processor is called as low weight process (LWP). Kernel thread and LWP has one-to-one correspondence. The available numbers of kernel threads can be changed dynamically. The O.S is used to schedule on to the real system.

Thread scheduling:

The scheduling of thread involves two boundary scheduling:

1. Scheduling of Kernel-Level Threads by the system scheduler.
2. Scheduling of User-Level Threads or ULT to Kernel-Level Threads or KLT by using Lightweight process or LWP.

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as process-contention scope (PCS) since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

Lightweight process (LWP):

The Lightweight process is threads that act as an interface for the User-Level Threads to access the physical CPU resources.

The number of the lightweight processes depends on the type of application, for an I\O bound application the number of LWP depends on the user level threads, and for CPU bound application each thread is connected to a separate kernel-level thread.

In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority, therefore, it requires two controls to be specified for the User level threads:

1. **Contention scope** – Control scope defines the extent to which contention takes place. Contention refers to the competition among the ULTs to access the KLTs.

Contention scope can be further classified into Process Contention Scope (PCS) and System Contention Scope (SCS).

- **Process Contention Scope:** Process Contention Scope is when the contention takes place in the same process.
- **System contention scope** (SCS): System Contention Scope refers to the contention that takes place among all the threads in the system.

2. **Allocation domain** – The allocation domain is a set of multiple (or single) resources for which a thread is competing.

Advantages of PCS over SCS:

The advantages of PCS over SCS are as follows:

1. It is cheaper.
2. It helps reduce system calls and achieve better performance.
3. If the SCS thread is a part of more than one allocation domain, the system will have to handle multiple interfaces.
4. PCS thread can share one or multiple available LWPs, while every SCS thread needs a separate LWP. Therefore, for every system call, a separate KLT will be created.

Multiple Processor Scheduling:

The availability of multiple processors makes scheduling more complicated, as there is more than one CPU that has to be kept busy at all times.
Load sharing is used in this case. It can be defined as balancing the load between multiple processors.

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- Processor affinity – process has affinity for processor on which it is currently running
- soft affinity
- hard affinity

Approaches to Multiple processor scheduling:

There are two approaches: Symmetric Multiprocessing and Asymmetric Multiprocessing.

- **Symmetric Multiprocessing:** In Symmetric Multiprocessing, all processors are self-scheduling.
- Asymmetric Multiprocessing: In Asymmetric Multiprocessing, scheduling decisions and I\O processes are handled by a single processor known as the Master Server.

Processor Affinity:

In processor affinity, the processes have a priority for the processor which they are running.

- **Soft affinity:** When the system tries to keep the processes on the same processor.
- Hard affinity: When the process specifies that it should not be moved between the processors.

Load Balancing:

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system so that one processor doesn't sit idle while the other is being overloaded.

- **Push Migration:** A task regularly checks if there is an imbalance of load among the processors and then shifts\distributes the load accordingly.
- **Pull Migration:** It occurs when an idle processor pulls a task from an overloaded\busy processor.

Multiprogramming – We have many processes ready to run. There are two types of multiprogramming:

1. **Pre-emption –** Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.
2. **Non pre-emption –** Processes are not removed until they complete the execution.

Degree of multiprogramming –

The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100, this means 100 processes can reside in the ready state at maximum.