UNIT II PROCESS MANAGEMENT

Processes - Process Concept - Process Scheduling - Operations on Processes - Inter-process Communication; CPU Scheduling - Scheduling criteria - Scheduling algorithms: Threads - Multithread Models – Threading issues; Process Synchronization - The Critical-Section problem - Synchronization hardware – Semaphores – Mutex - Classical problems of synchronization - Monitors; Deadlock - Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

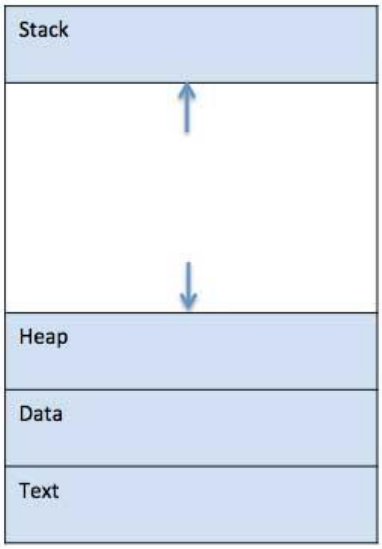
PROCESSES

PROCESS CONCEPT

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.

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.



Layout of a process

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

Program

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.

A computer program is a collection of instructions that performs a specific task when executed by a computer.

Algorithm

A part of a computer program that performs a well-defined task is known as an algorithm.

Software

A collection of computer programs, libraries and related data are referred to as a software.

Process Life Cycle / State

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.

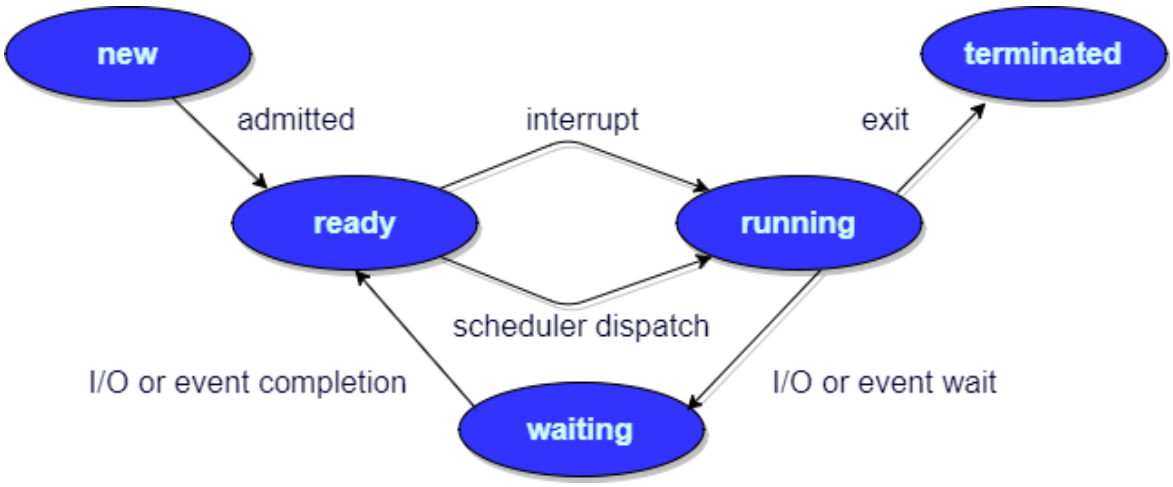
NEW - The process is being created.

READY - The process is waiting to be assigned to a processor.

RUNNING - Instructions are being executed.

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

TERMINATED - The process has finished execution.



Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process.

A PCB keeps all the information needed to keep track of a process.

Process State - The current state of the process i.e., whether it is ready, running, waiting, or whatever.

Process ID - Unique identification for each of the process in the operating system.

Program Counter - Program Counter is a pointer to the address of the next instruction to be executed for this process.

CPU registers - Various CPU registers where process need to be stored for execution for running state.

Memory management information - This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
Accounting information - This includes the amount of CPU used for process execution, time limits, execution ID etc.
IO status information - This includes a list of I/O devices allocated to the process.

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

PROCESS SCHEDULING
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.
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.

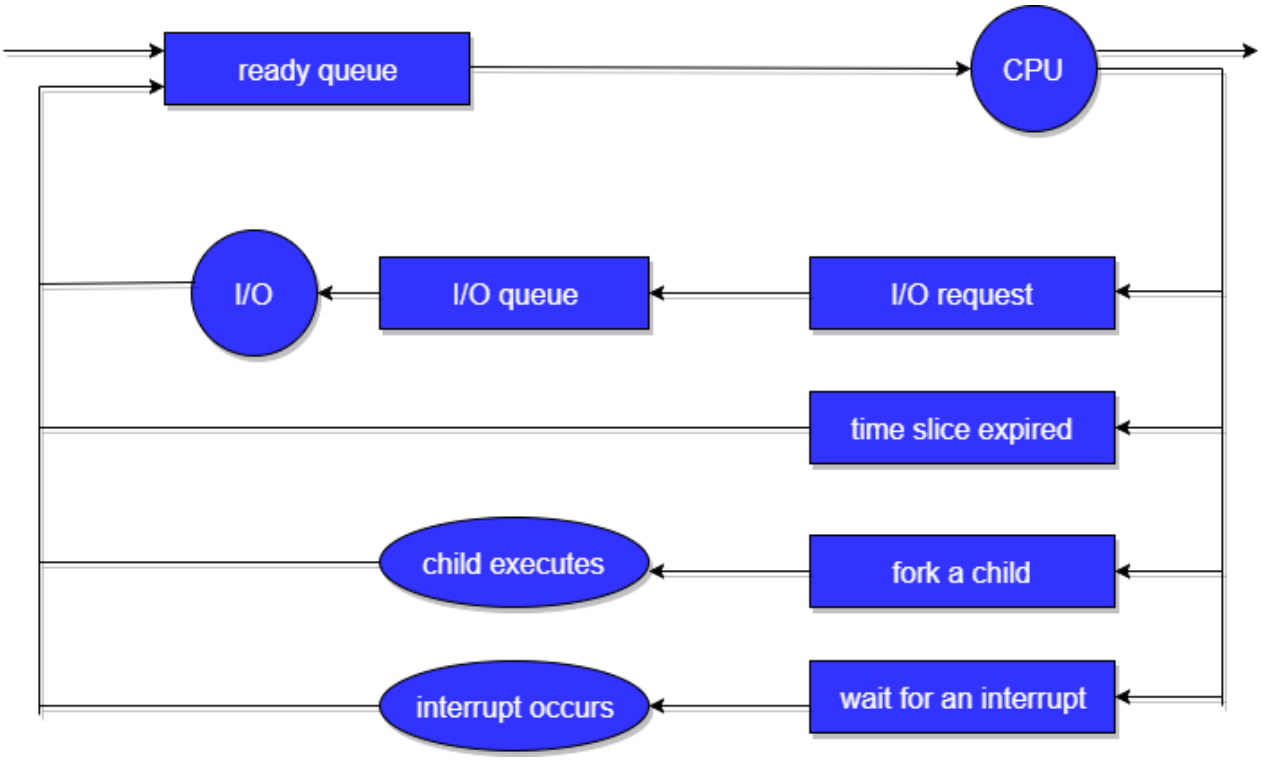
Scheduling Queues.
It refers to queues of process or devices.

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.

Queue is represented by **rectangular box**.
Circle represents the **resources** that serve the queues.
Arrows indicate the **process flow** in the system.

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 sub process 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.

Two-State Process Model

Two-state process model refers to running and non-running states

Running - When a new process is created, it enters into the system as in the running state.

Not Running - Processes that are not running are kept in queue, waiting for their turn to execute.

Schedulers

Schedulers are special system software which handles process scheduling in various ways.

Its main task is to select the jobs to be submitted into the system and to decide which process to run.

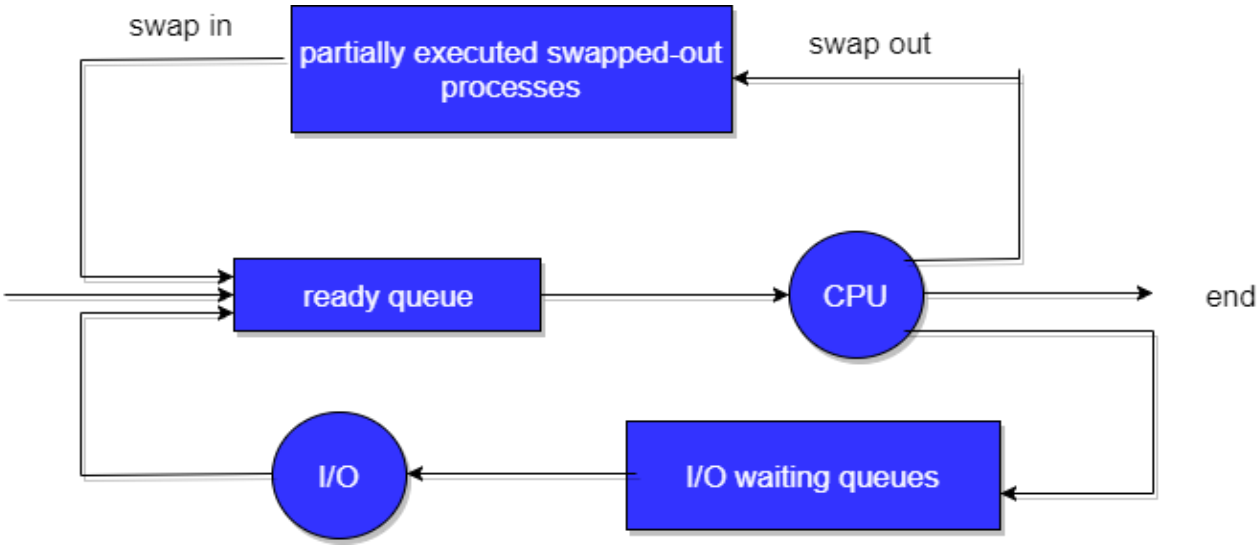
Schedulers are of three types:

- 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.

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.**



Comparison among Scheduler

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.

OPERATIONS ON PROCESS

The two major operation provided by OS are Process Creation and Process Termination.

Process Creation

The system calls, fork is used by processes may create other processes.

The process which creates other process, is termed the parent process, while the created sub-process is termed its child.

The child process is a duplicate of the parent process.

Each process is given an integer identifier, termed as process identifier, or PID.

The parent PID (PPID) is also stored for each process.

Process Termination

A process terminates when it finishes executing its last statement.

Its resources are returned to the system.

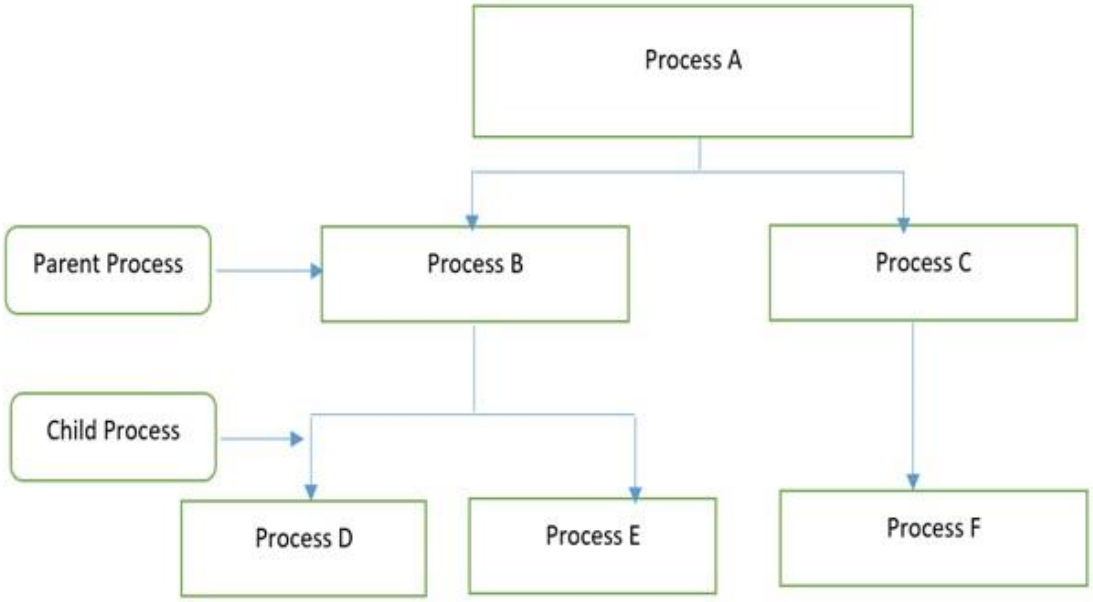
The new process terminates the existing process, usually due to following reasons:

Normal Exist - Most processes terminates because they have done their job. This call is exist in UNIX.

Error Exist - When process discovers a fatal error. For example, a user tries to compile a program that does not exist.

Fatal Error - An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.

Killed by Another Process - A process executes a system call telling the Operating Systems to terminate some other process.



INTER-PROCESS COMMUNICATION

Inter Process Communication (IPC) refers to a mechanism, for processes to communicate and to synchronize their actions.

Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system.

Message system - processes communicate with each other without resorting to shared variables.

IPC facility provides two operations:

Send (message) – message size fixed or variable

Receive (message)

If P and Q wish to communicate, they need to:

Establish a **communication link** between them

Exchange **messages** via send/receive

Implementation of communication link

Physical (e.g., shared memory, hardware bus)

Logical (e.g., logical properties)

Message Passing System

Messages sent by a process can be of either fixed or variable size. Implementation is straight forward.

Several methods for logically implementing a link and the send /receive operations.

- Direct / Indirect Communication
- Symmetric / Asymmetric Communication
- Automatic / Explicit buffering
- Synchronous / Asynchronous Communication

Direct / Indirect 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

Process wants to communicate must name each other explicitly:

send (P, message) - send a message to process P

receive (Q, message) - receive a message from process Q

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.

send (A, message) - send a message to mailbox A

receive (A, message) - receive a message from mailbox A

Symmetric / Asymmetric Communication

Symmetric Communication

Both recipient and sender must name the other for communication.

send (P, message) - Send a message to process P.

receive (Q, message) - Receive a message from process Q.

Asymmetric Communication

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

send (P, message) - Send a message to process P.

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

Automatic / Explicit buffering

Automatic buffering

Provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message.

Explicit buffering

Specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue.

Synchronous / Asynchronous Communication

Synchronous Communication

P and Q have to wait for each other (one blocks until the other is ready).

Asynchronous Communication

Underlying system buffers the messages so P and Q don't have to wait for each other.

CPU SCHEDULING

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

SCHEDULING CRITERIA

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

CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time. (Ideally 100% of the time).

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time.

$$Throughput = \frac{Number\ of\ processes\ completed}{Total\ amount\ of\ time\ required\ to\ complete\ processes}$$

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. the interval from time of submission of the process to the time of completion of the process.

Waiting Time

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.

Response Time

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.

Two Scheduling Philosophies:

Preemptive Scheduling

In preemptive mode, currently running process may be interrupted and moved to the ready State by the operating system.

Non-preemptive Scheduling

In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/Output or by requesting some operating system service.

SCHEDULING ALGORITHMS

To decide which process to execute first and which process to execute last to achieve maximum CPU utilisation, they are:

- First Come First Serve (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR) Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First Come First Serve (FCFS) Scheduling

Jobs are executed on first come, first serve basis.

It is a non-preemptive, pre-emptive scheduling algorithm.

Easy to understand and implement.

The process which arrives first, gets executed first, or we can say that the process which requests the CPU first gets the CPU allocated first.

Its implementation is based on FIFO (First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.

This is used in Batch Systems.

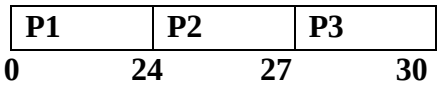
Example Problem:

1. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	24
P2	3
P3	3

What is the average waiting time and average turnaround time for these processes?

Gantt chart



Average Waiting Time

Waiting Time

Waiting Time = Completion Time - Burst Time - Arrival Time

Process	Waiting Time
P1	24 - 24 - 0 = 0
P2	27 - 3 - 0 = 24
P3	30 - 3 - 0 = 27

Average Waiting Time = 0 + 24 + 27 / 3 = 17 ms

Average Turn around Time

Turn around Time

Turn around Time = *Completion Time - Arrival Time*

Process	Turn around Time
P1	24 - 0 = 24
P2	27 - 0 = 27
P3	30 - 0 = 30

Average Turn around Time = 24 + 27 + 30 / 3 = 27 ms

$$Throughput = \frac{Number\ of\ processes\ completed}{Total\ amount\ of\ time\ required\ to\ complete\ processes}$$

= 3 / 30 = 0.1 processes per millisecond

Additional Problems:

2. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	14
P2	2
P3	4

What is the average waiting time and average turnaround time for these processes?

3. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	3
P2	6
P3	4
P4	2

What is the average waiting time and average turnaround time for these processes?

4. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

What is the average waiting time and average turnaround time for these processes?

5. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	4
P2	7
P3	3
P4	3
P5	3

What is the average waiting time and average turnaround time for these processes?

6. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	5
P2	10
P3	8
P4	3

What is the average waiting time and average turnaround time for these processes?

Shortest-Job-First (SJF) / Shortest Job Next (SJN) Scheduling

Shortest Job First scheduling works on the process with the shortest burst time or duration first.
If two processes are available at the same time, FCFS is used to break the tie.
Best approach to minimize waiting time.
This is used in Batch Systems.

It is of two types:

- Non Pre-emptive
- Pre-emptive

Non Pre-emptive

Schedules the processes according to their burst time.

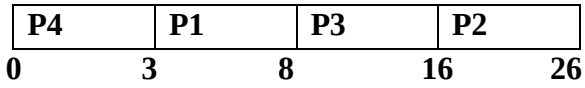
Example Problem:

1. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	5
P2	10
P3	8
P4	3

What is the average waiting time and average turnaround time for these processes?

Gantt chart



Average Waiting Time

Waiting Time

Waiting Time = Completion Time – Burst Time – Arrival Time

Process	Waiting Time
P1	8 – 5 – 0 = 3
P2	26 – 10 – 0 = 16
P3	16 – 8 – 0 = 8
P4	3 – 3 – 0 = 0

Average Waiting Time = 3 + 16 + 8 + 0 / 4 = 6.7 ms

Average Turn around Time

Turn around Time

Turn around Time = *Completion Time - Arrival Time*

Process	Turn around Time
P1	8 – 0 = 8
P2	26 – 0 = 26
P3	16 – 0 = 16
P4	3 – 0 = 3

Average Turn around Time = 8 + 26 + 16 + 3 / 4 = 13.25 ms

Throughput = Number of processes completed
Total amount of time required to complete processes

= 4 / 26 = 0.15 processes per millisecond

Additional Problems:

2. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	21
P2	3
P3	6
P4	2

What is the average waiting time and average turnaround time for these processes?

3. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

What is the average waiting time and average turnaround time for these processes?

4. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	5
P2	2
P3	6
P4	4

What is the average waiting time and average turnaround time for these processes?

5. Consider the following set of processes that arrive at time 0, with the length of CPU burst given in milliseconds.

Process	Burst Time
P1	3
P2	6
P3	4
P4	2

What is the average waiting time and average turnaround time for these processes?

6. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Arrival Time
P1	7	0
P2	4	1
P3	1	4
P4	4	5

What is the average waiting time and average turnaround time for these processes?

Pre-emptive

Pre-emption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling***. i.e. It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

Example Problem:

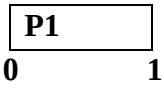
1. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Arrival Time
P1	8	0
P2	4	1
P3	9	2
P4	5	3

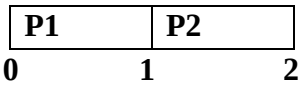
What is the average waiting time and average turnaround time for these processes?

Gantt chart

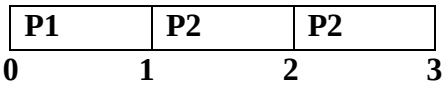
P1 starts at time 0 ms



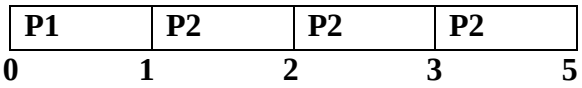
At 1 ms, P1 as 7 ms BT and compared with P2 as 4 ms BT, so we choose P2



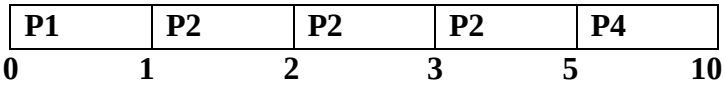
At 2 ms, P1 as 7 ms BT whereas P2 as 3 ms BT and P3 as 9 ms BT, so we choose P2



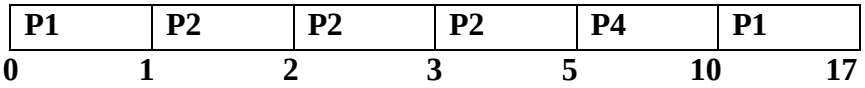
At 3 ms, P1 as 7 ms BT whereas P2 as 2 ms BT, P3 as 9 ms BT and P4 as 5 ms BT, so we choose P2



At 5 ms, P1 as 7 ms BT whereas P2 as 0 ms BT, P3 as 9 ms BT and P4 as 5 ms BT, so we choose P4



Now P2 and P4 are over, P1 as 7 ms BT, P3 as 9 ms BT, so we choose P1



Finally P1, P2 and P4 are over; P3 is remaining as 9 ms to be scheduled

P1	P2	P2	P2	P4	P1	P3

0 1 2 3 5 10 17 26

Average Waiting Time

Waiting Time

Waiting Time = Completion Time – Burst Time – Arrival Time

Process	Waiting Time
P1	17 – 8 – 0 = 9
P2	5 – 4 – 1 = 0
P3	26 – 9 – 2 = 15
P4	10 – 5 – 3 = 2

Average Waiting Time = 9 + 0 + 15 + 2 / 4 = 6.5 ms

Average Turn around Time

Turn around Time

Turn around Time = *Completion Time - Arrival Time*

Process	Turn around Time
P1	17 – 0 = 17
P2	5 – 1 = 4
P3	26 – 2 = 24
P4	10 – 3 = 7

Average Turn around Time = 17 + 4 + 24 + 7 / 4 = 13 ms

Throughput = Number of processes completed
Total amount of time required to complete processes

= 4 / 26 = 0.15 processes per millisecond

Additional Problems:

2. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Arrival Time
P1	6	0
P2	2	1
P3	8	2
P4	4	3

What is the average waiting time and average turnaround time for these processes?

3. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Arrival Time
P1	21	0
P2	3	1
P3	6	2
P4	2	3

What is the average waiting time and average turnaround time for these processes?

Priority Scheduling

Priority is assigned for each process.

Process with highest priority is executed first and so on.

Processes with same priority are executed in FCFS manner.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.

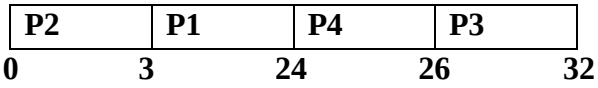
Example Problem:

1. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	21	2
P2	3	1
P3	6	4
P4	2	3

What is the average waiting time and average turnaround time for these processes?

Gantt chart



Average Waiting Time

Waiting Time

Waiting Time = Completion Time – Burst Time – Arrival Time

Process	Waiting Time
P1	24 – 21 – 0 = 3
P2	3 – 3 – 0 = 0
P3	32 – 6 – 0 = 26
P4	26 – 2 – 0 = 24

Average Waiting Time = 3 + 0 + 26 + 24 / 4 = 13.25 ms

Average Turn around Time

Turn around Time

Turn around Time = *Completion Time - Arrival Time*

Process	Turn around Time
P1	24 – 0 = 24
P2	3 – 0 = 3
P3	32 – 0 = 32
P4	26 – 0 = 26

Average Turn around Time = 24 + 3 + 32 + 26 / 4 = 21.25 ms

$$Throughput = \frac{Number\ of\ processes\ completed}{Total\ amount\ of\ time\ required\ to\ complete\ processes}$$

= 4 / 32 = 0.125 processes per millisecond

Additional Problems:

2. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

What is the average waiting time and average turnaround time for these processes?

3. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	5	3
P2	10	2
P3	8	4
P4	3	1

What is the average waiting time and average turnaround time for these processes?

4. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	10	2
P2	7	1
P3	20	3
P4	30	4

What is the average waiting time and average turnaround time for these processes?

5. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	3	2
P2	6	4
P3	4	1
P4	2	3

What is the average waiting time and average turnaround time for these processes?

6. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	5	3
P2	2	4
P3	6	1
P4	4	2

What is the average waiting time and average turnaround time for these processes?

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 pre-empted and other process executes for a given time period.

Every process gets executed in a cyclic way.

Context switching is used to save states of pre-empted processes.

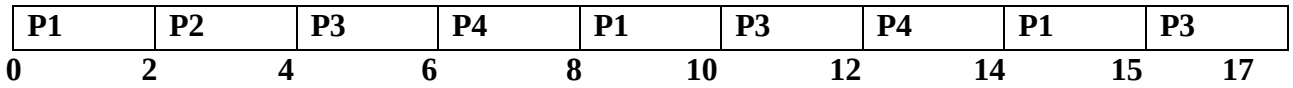
Example Problem:

1. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds. The time quantum given is 2 ms.

Process	Burst Time
P1	5
P2	2
P3	6
P4	4

What is the average waiting time and average turnaround time for these processes?

Gantt chart



Average Waiting Time

Waiting Time

Waiting Time = Completion Time – Burst Time – Arrival Time

Process	Waiting Time
P1	15 – 5 – 0 = 10
P2	4 – 2 – 0 = 2
P3	17 – 6 – 0 = 11
P4	14 – 4 – 0 = 10

Average Waiting Time = 10 + 2 + 11 + 10 / 4 = 8.25 ms

Average Turn around Time

Turn around Time

Turn around Time = *Completion Time - Arrival Time*

Process	Turn around Time
P1	15 – 0 = 15
P2	4 – 0 = 4
P3	17 – 0 = 17
P4	14 – 0 = 14

Average Turn around Time = 15 + 4 + 17 + 14 / 4 = 12.5 ms

$$Throughput = \frac{Number\ of\ processes\ completed}{Total\ amount\ of\ time\ required\ to\ complete\ processes}$$

= 4 / 17 = 0.235 processes per millisecond

Additional Problems:

2. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds. The time quantum given is 2 ms.

Process	Burst Time
P1	3
P2	6
P3	4
P4	2

What is the average waiting time and average turnaround time for these processes?

3. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds. The time quantum given is 4 ms.

Process	Burst Time
P1	24
P2	3
P3	3

What is the average waiting time and average turnaround time for these processes?

4. Consider the following set of processes in the same order of arrival, with the length of CPU burst given in milliseconds. The time quantum given is 5 ms.

Process	Burst Time
P1	25
P2	5
P3	5

What is the average waiting time and average turnaround time for these processes?

Multilevel Queue Scheduling

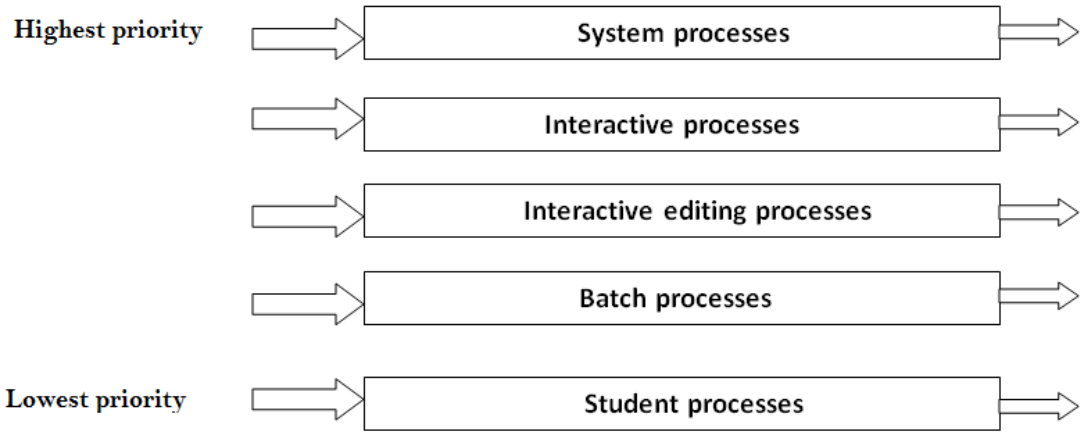
A multilevel queue scheduling algorithm partitions the ready queue in several separate queues.
In a multilevel queue scheduling processes are permanently assigned to one queues.
The processes are permanently assigned to one another, based on some property of the process, such as

- Memory size
- Process priority
- Process type

Algorithm choose the process from the occupied queue that has the highest priority, and run that process either

- Preemptive or
- Non-preemptively

Each queue has its own scheduling algorithm or policy.
The foreground queue can be scheduled by using a round-robin algorithm (80% of the CPU time) while the background queue is scheduled by a first come first serve algorithm (20% of the CPU time).



Let us take an example of a multilevel queue scheduling algorithm with five queues:

1. System process
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

All the five processes have their own queue.
System process: there is various kind of process in the operating system. The operating system itself has own process to run called system process
Interactive process: the interactive process is online game or listing music (there should be the same kind of interaction)
Interactive editing process: the interactive editing process is Data or Image editing (there should be the same kind of interaction)

Batch system: we submit the job to the processor and take the result later.
Student process: normal student processes (Application) comes inside student process.

The system process will get the highest priority and the student process gets the lowest priority.
There are many types of the process we can't put them in one queue and get the result.
This problem is resolved by the multilevel queue scheduling.

Disadvantage:
In multilevel queue scheduling, the main disadvantage is the Starvation problem for lowest level process

Starvation:
Lower priority process never executes. or wait for the long amount of time because of lower priority or highest priority process taking a large amount of time.

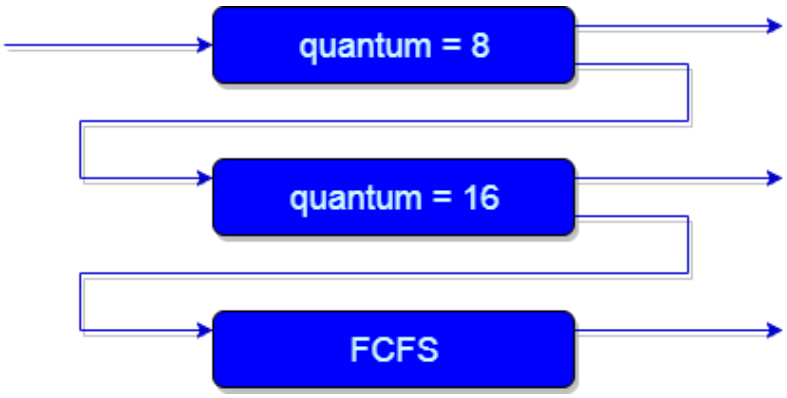
Advantage:
We can apply separate scheduling for various kind of process like
System process - FCFS scheduling algorithm
Interactive process - SJF scheduling
Batch process - Round Robin
Student Process - Priority scheduling

Multilevel Feedback Queue Scheduling
Generally, we see in a ***multilevel queue scheduling algorithm*** processes are permanently stored in one queue in the system and do not move between the queue. There is some separate queue for foreground or background processes but the processes do not move from one queue to another queue and these processes do not change their foreground or background nature, these type of arrangement has the advantage of low scheduling but it is inflexible in nature.

Multilevel feedback queue scheduling it allows a process to move between the queues. This process is separate with different CPU burst time. If a process uses too much CPU time then it will be moved to the lowest priority queue. This idea leaves I/O bound and interactive processes in the higher priority queue. Similarly, the process which waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

The multilevel feedback queue scheduler has the following parameters:

1. The number of queues
2. The scheduling algorithm for each queue
3. The method used to demote/downgrade processes to lower priority queues
4. The method used to promote/upgrade processes to higher priority queues
5. The method used to determine which queue a process will enter



Let's consider there are three queues Q1, Q2 and Q3.

- **Q1** Time quantum 8 milliseconds
- **Q2** Time quantum 16 milliseconds
- **Q3** FCFS

Criteria of Multilevel Feedback Queue:

- The scheduler first executes all processes in Q1.
- Only when Q1 is empty will it execute processes in Q2.
- Similarly, processes in Q3 will be executed only if Q1 and Q2 are empty.
- A process that arrives for Q2 will pre-empt a process in Q3.
- A process that arrives for Q3 will, in turn, pre-empt a process in Q3.

Explanation

Any process entering the ready queue is put in Q1. Then, A process in Q1 is given a time quantum of 8 milliseconds. If it does not finish within time, it is moved to the tail of Q2.
If Q1 is empty, the process at the head of Q2 is given a quantum of 16 milliseconds. Or, if it does not complete, it is pre-empted and is put into Q3.
Processes in Q3 are run on an FCFS basis, only when Q1 and Q2 are empty.

Aging promotes lower priority process to the next higher priority queue after a suitable interval of time.

Advantages

A process that waits too long in a lower priority queue may be moved to a higher priority queue.

Disadvantages

Moving the process around queue produces more CPU overhead.

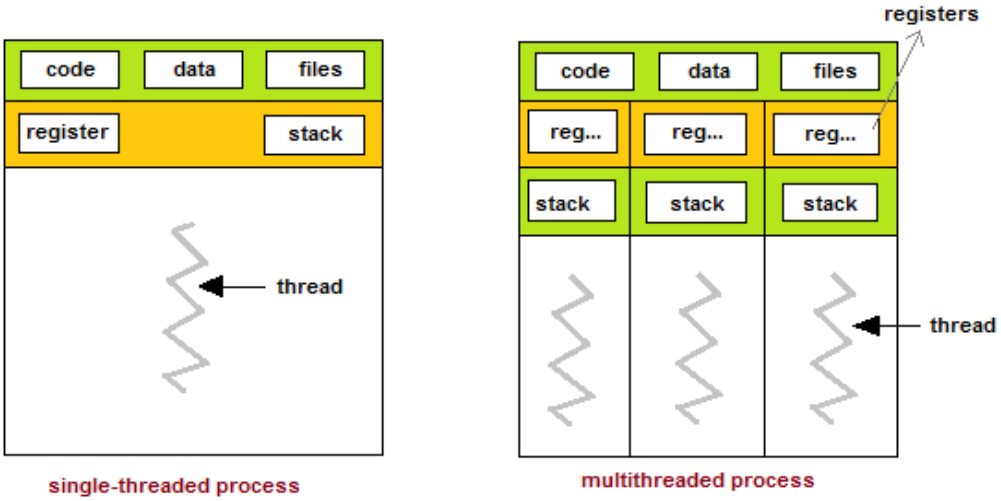
THREADS – OVERVIEW

A Thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID).

Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.

A process can contain multiple threads.

A thread is also known as lightweight process.



Difference between Process and Thread

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.

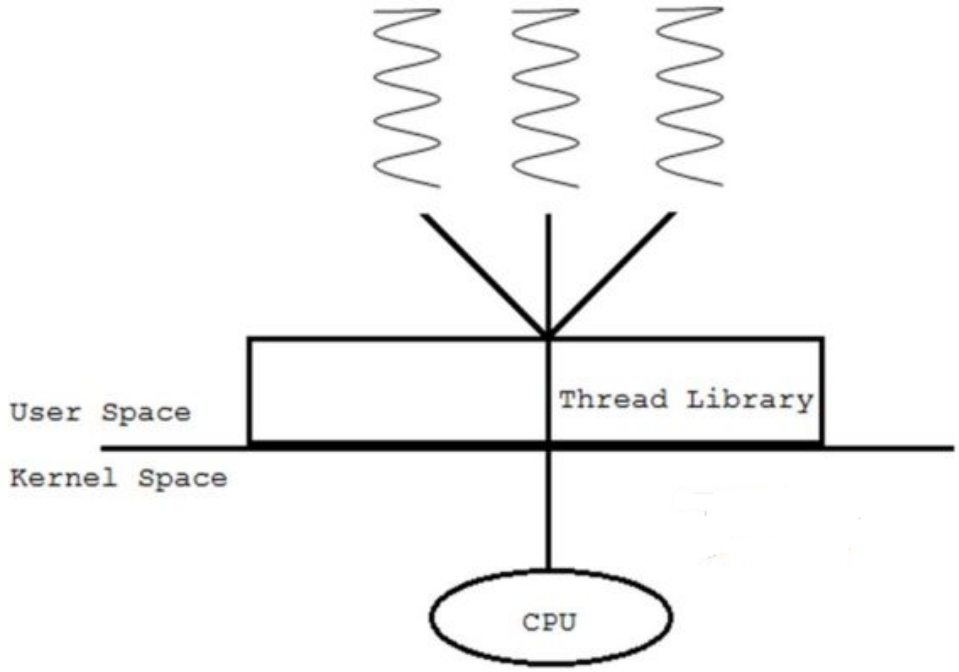
Types of Thread

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 threads are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

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.



Thread Libraries

Thread libraries provide programmers with API for creation and management of threads.

Thread libraries may be implemented either in user space or in kernel space.

The user space involves API functions implemented solely within the user space, with no kernel support.

The kernel space involves system calls, and requires a kernel with thread library support.

Implementations of Thread

POSIX Pitheads may be provided as either a user or kernel library, as an extension to the POSIX standard.

Win32 threads are provided as a kernel-level library on Windows systems.

Java threads Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system.

Linux threads Linux does not distinguish between processes and threads - It uses the more generic term "tasks". The traditional fork ( ) system call completely duplicates a process (task), as described earlier.

An alternative system call, clone ( ) allows for varying degrees of sharing between the parent and child tasks.

Benefits of Multithreading

Responsiveness, one thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

Resource sharing, hence allowing better utilization of resources.

Economy, Creating and managing threads become easier.

Scalability, One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.

Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

MULTITHREADING MODELS

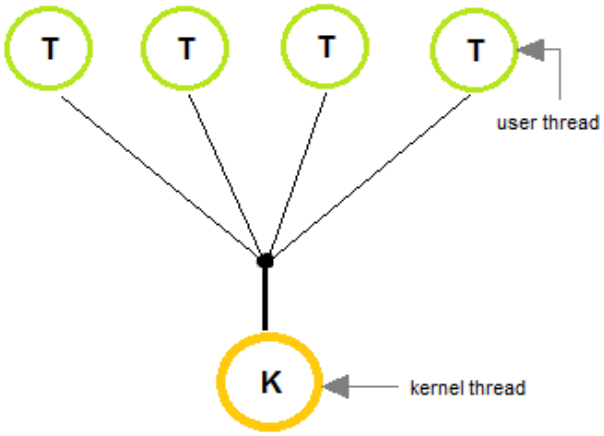
The user threads must be mapped to kernel threads, by one of the following strategies:

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

Many to One Model

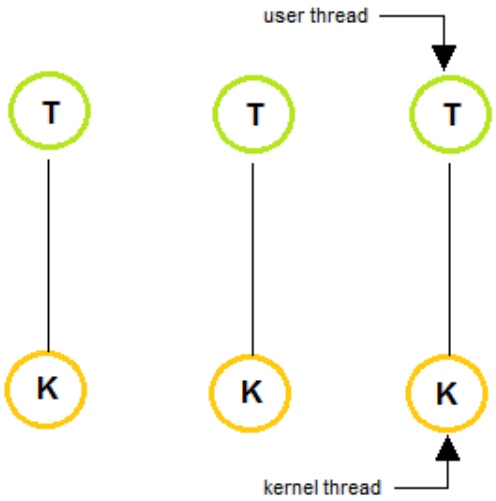
In the many to one model, many user-level threads are all mapped onto a single kernel thread.

Thread management is handled by the thread library in user space, which is efficient in nature.



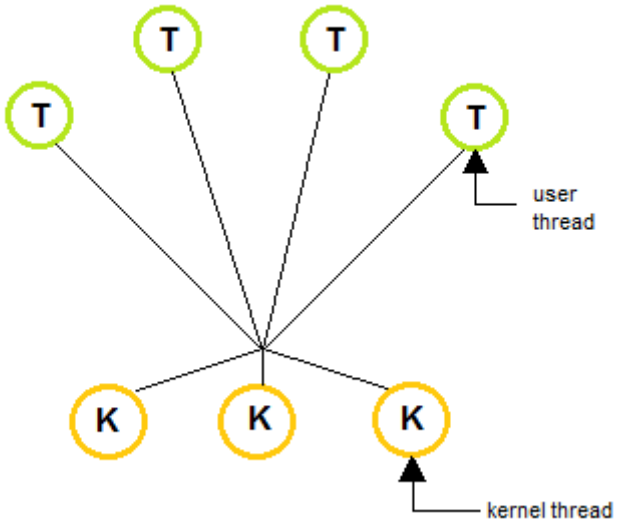
One to One Model

The one to one model creates a separate kernel thread to handle each and every user thread.
Most implementations of this model place a limit on how many threads can be created.
Linux and Windows from 95 to XP implement the one-to-one model for threads.

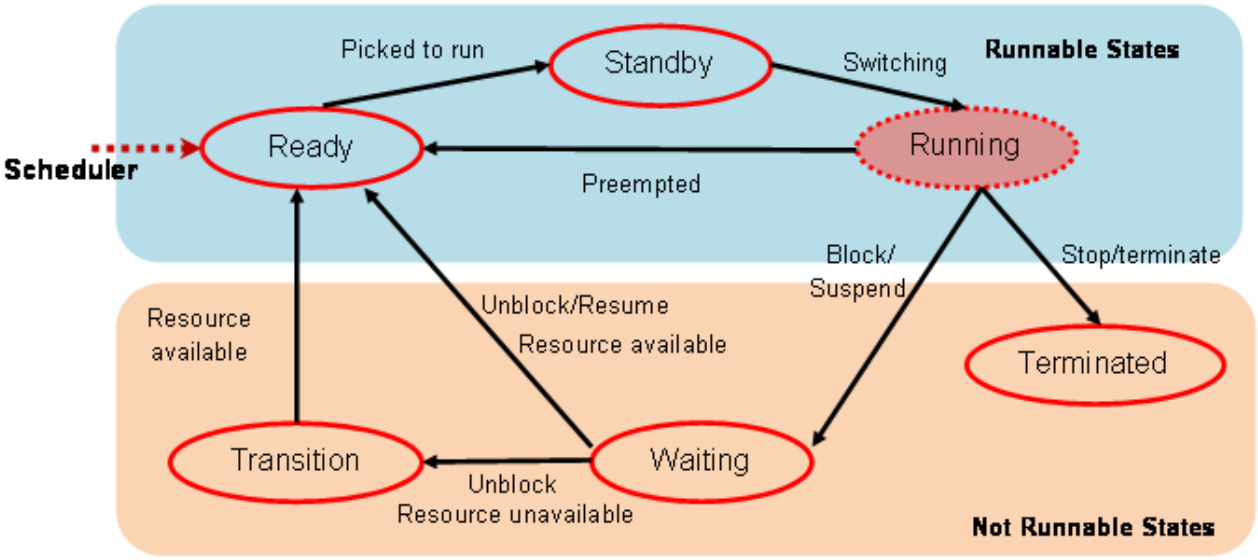


Many to Many Model

The many to many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
Users can create any number of the threads.
Blocking the kernel system calls does not block the entire process.
Processes can be split across multiple processors.



Thread States



Windows Thread States

A thread can be in the following states:

- **Ready** - it is prepared to run on the next available processor.
- **Running** - it is executing.
- **Standby** - it is about to run; only one thread may be in this state at a time.
- **Terminated** - it is finished executing.
- **Waiting** - it is not ready for the processor, when ready, it will be rescheduled.
- **Transition** - the thread is waiting for resources other than the processor,

THREADING ISSUES

The fork( ) and exec( ) System Calls

Q: If one thread forks, is the entire process copied, or is the new process single-threaded?

A: System dependant.

A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.

A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

Signal Handling

Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?

A: There are four major options:

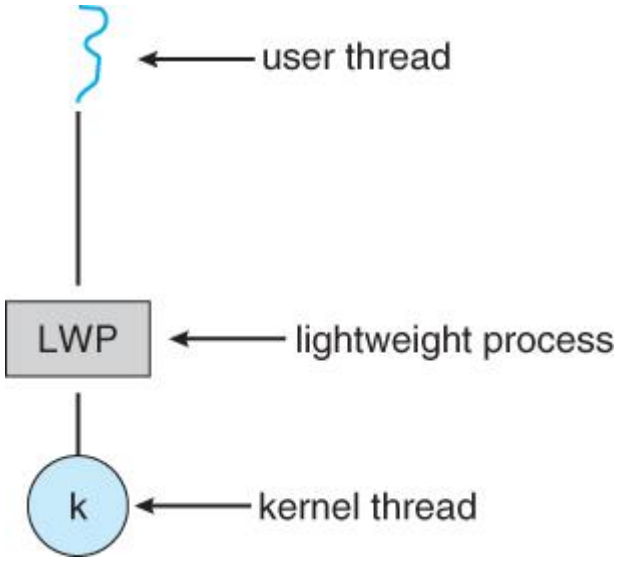
1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals in a process.

The best choice may depend on which specific signal is involved.
UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
UNIX provides two separate system calls***, kill (pid, signal) and pthread_kill (tid, signal),*** for delivering signals to processes or specific threads respectively.
Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

Thread Cancellation
Threads that are no longer needed may be cancelled by another thread in one of two ways:
Asynchronous Cancellation cancels the thread immediately.
Deferred Cancellation sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
(Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

Thread-Local Storage (Thread-Specific Data)
Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
However sometimes threads need thread-specific data also.
Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as thread-local storage or TLS. Note that this is more like static data than local variables, because it does not cease to exist when the function ends.

Scheduler Activations
Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
This virtual processor is known as a "Lightweight Process", LWP.
There is a one-to-one correspondence between LWPs and kernel threads.



The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
The application (user level thread library) maps user threads onto available LWPs.
Kernel threads are scheduled onto the real processor(s) by the OS.
The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an upcall, which is handled in the thread library by an ***upcall handler.***
The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.
If the kernel thread blocks, then the LWP blocks, which blocks the user thread.

PROCESS SYNCHRONIZATION

Process Synchronization was introduced to handle problems that arose while multiple process executions.
Process Synchronization is a technique which is used to coordinate the process that use shared Data.

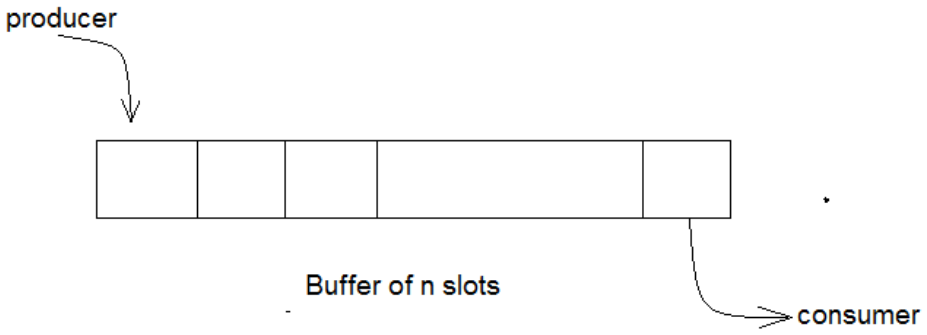
Types of Processes

Independent Process: The process that does not affect or is affected by the other process whiles its execution then the process is called Independent Process. ***Example*** The process that does not share any shared variable, database, files, etc.

Cooperating Process: The process that affect or is affected by the other process while execution, is called a Cooperating Process. ***Example*** The process that share file, variable, database, etc are the Cooperating Process.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

To explain cooperating process, Producer consumer problem is considered.
It's also known as bounded buffer problem.
In this problem we have two processes, producer and consumer, who share a fixed size buffer.
Producer work is to produce data or items and put in buffer.
Consumer work is to remove data from buffer and consume it.

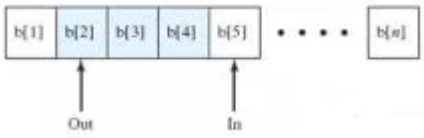


Types of Buffer

- **Unbounded-buffer** places no practical limit on the size of the buffer.
- **Bounded-buffer** assumes that there is a fixed buffer size

Basic synchronization requirement

- Producer should not write into a full buffer
- Consumer should not read from an empty buffer
- All data written by the producer must be read exactly once by the consumer



By using Shared Memory we can solve bounded buffer problem:

The shared buffer is implemented as a circular array with two logical pointers: IN and OUT.

IN points to the next free position in the buffer;

OUT points to the first full position in the buffer.

The buffer is empty when **IN == out;**

The buffer is full when **((IN+1) % BUFFER_SIZE) == OUT.**

Shared data

```
#define BUFFER_SIZE 10
        typedef struct {
        . . .
        } item;

        item buffer[BUFFER_SIZE];
        int in = 0;
        int out = 0;
```

Producer Process

```
item nextProduced;
while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)        /*   (or) while(Counter == BUFFER_SIZE)   */
        ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;                   /*   (or) Counter ++   */
}
```

Consumer Process

```
item nextConsumed;
while (1) {
        while (in == out)                          /* (or) while(Counter == 0)       */
        ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;         /*   (or) Counter --   */
}
```

THE CRITICAL-SECTION PROBLEM

Critical section is a code segment that can be accessed by only one process at a time.

Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

If any other process also wants to execute its critical section, it must wait until the first one finishes.

```
do {
Entry section
    Critical section
Exit section
   Reminder section
} while (1);
```

Entry Section

It is part of the process which decide the entry of a particular process in the Critical Section, out of many other processes.

Critical Section

It is the part in which only one process is allowed to enter and modify the shared variable.This part of the process ensures that only no other process can access the resource of shared data.

Exit Section

This process allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical Section can be removed through this Exit Section.

Remainder Section

The other parts of the Code other than Entry Section, Critical Section and Exit Section are known as Remainder Section.

Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

SYNCHRONIZATION HARDWARE

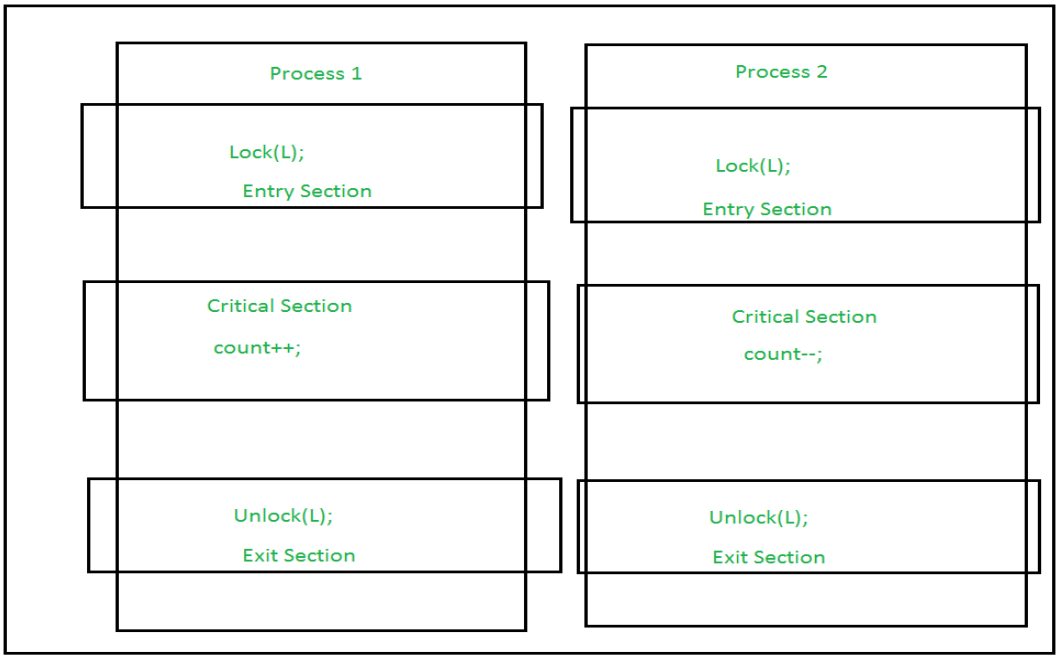
Many systems provide hardware support for critical section code.

1. Lock & Unlock Technique

The Hardware Approach of synchronization can be done through Lock & Unlock technique.

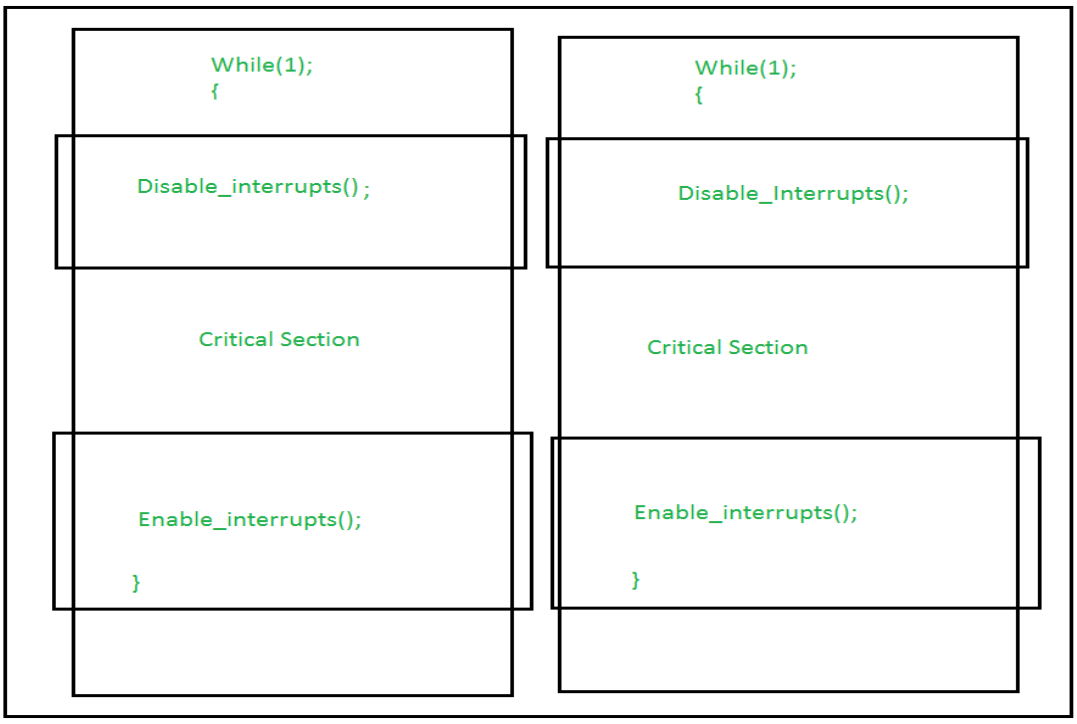
Locking part is done in the Entry Section, so that only one process is allowed to enter into the Critical Section, after it complete its execution, the process is moved to the Exit Section, where Unlock Operation is done so that another process in the Lock Section can repeat this process of Execution.

This process is designed in such a way that all the three conditions of the Critical Sections are satisfied.



2. Using Interrupts

These are easy to implement. When Interrupt are disabled then no other process is allowed to perform Context Switch operation that would allow only one process to enter into the Critical State.



3. Test_and_Set Operation

Test_and_Set is a hardware solution to the synchronization problem.

It allows boolean value (True/False) as a hardware Synchronization, which is atomic in nature i.e no other interrupt is allowed to access. This is mainly used in Mutual Exclusion Application.

In Test_and_Set, we have a shared lock variable which can take either of the two values, 0 or 1.

0 Unlock

1 Lock

Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting till it become free and if it is not locked, it takes the lock and executes the critical section.

In Test_and_Set, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

```
do {
        while(Test_and_Set(lock))
        : do no-op;
                critical section
        lock = false;
                remainder section
  } while(1)
```

MUTEX LOCKS

Mutex is short for **MUTual EXclusion**.

The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent. Therefore most systems offer a software API equivalent called mutex locks or simply mutexes.

The terminology when using mutexes is

- to acquire a lock prior to entering a critical section, and
- to release it when exiting.

```
do {
        acquire lock
                critical section
        release lock
                remainder section
  } while (true);
```

The acquire lock will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.

Acquire and release can be implemented as shown here, based on a Boolean variable **"available".**

The definition of acquire () is as follows:

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

The definition of release() is as follows:

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

Calls to either acquire() or release() must be performed atomically.

SEMAPHORES

Semaphore is a simply a integer value or shared variable.

This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment.

It is basically a synchronizing tool and is accessed only through two standard atomic operations, wait and signal designated by P() and V() respectively.

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s.

- P() stands for prolagen, which means try to decrease.
- V() stands for verhogen, which means increase.

Here atomic operations mean executing operation without interruption.

The definitions of wait and signal are as follows:

Wait : decrement the value of its argument S as soon as it would become non-negative.

```
wait(S) {
while (S <= 0)
// no-op ( No Operation);
S--;
}
```

Signal : increment the value of its argument, S as an individual operation.

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

Semaphore can be divided into two categories.

1. Counting Semaphore
2. Binary Semaphore or Mutex

Binary semaphore

Binary semaphore value can have only two values 0 and 1;

1 to represent that the process/thread is in critical section (code that access the shared resources)
0 indicating that the critical section is free.

A binary semaphore is simpler to implement than counting semaphore.

Wait operation

```
Semaphore b;
Initial value of b=1;
Wait (b)
{
    While (b==0)
    b--;
}
```

Signal operation

```
Semaphore b;
Initial value of b=1;
Signal (b)
{
    b++;
}
```

IMPLEMENTATION

The implementation of mutex locks suffers from busy waiting,

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () operations as follows:

When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation.

The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
int value;
struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list.

When a process must wait on a semaphore, it is added to the list of processes.

A signal () operation removes one process from the list of waiting processes and awakens that process.

wait() operation can be defined as

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
    add this process to S->list;
    block();
    }
}
```

signal() operation can be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
    }
}
```

The block () operation suspends the process that invokes it. The wakeup (P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Counting semaphore:

Counting semaphore takes more than two values. They can have any value you want.

Counting semaphores can take on any integer value, and are usually used to count the number remaining of some limited resource.

The counter is initialized to the number of such resources available in the system.

The counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources.

When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call.

Semaphores can also be used to synchronize certain operations between processes.

For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.

- First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.
- Ehen in process P1 we insert the code:

```
S1;
    signal ( synch ) ;
    and in process P2 we insert the code:
    wait ( synch ) ;
S2;
```

Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

Implementation of computing semaphore

Wait operation

```
Semaphore c;
Initial value c = Number of instance of type R.
Wait(c):
 {
    c--;
    if(c<0)
    {
    Block the process and put in blocked queue
    }
 }
```

Signal operation

```
Semaphore c;
Initial value c = Number of instance of type R.
Signal(S):
 {
    c++;
    if(c<=0)
    {
    Remove process from block queue and keep in ready queue
    }
 }
```

CLASSIC PROBLEMS OF SYNCHRONIZATION

Semaphore can be used in other synchronization problems besides Mutual Exclusion.
Below are some of the classical problems depicting flaws of process synchronization in systems where cooperating processes are present.

Three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

Bounded Buffer (Producer-Consumer) Problem

This is the same producer / consumer problem as before. But now we'll do it with signal and wait. **Remember:** a wait decreases its argument and a signal increases its argument.

N buffers, each can hold one item
„
Binary Semaphore

```
mutex = 1; // can only be 0 or 1
```

Counting Semaphore

```
empty = n; full = 0; // can take on any integer value
```

The structure of the producer process

```
do {
    // produce an item in nextp
    wait (empty);
    wait (mutex);

    // add the item to the buffer

    signal (mutex);
    signal (full);
} while (TRUE);
```

The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

    // remove an item from buffer to nextc

    signal (mutex);
    signal (empty);

    // consume the item in nextc
} while (TRUE);
```

Readers-Writers Problem

A data set is shared among a number of concurrent processes

- **Readers -** only read the data set; they do not perform any updates
- **Writers -** can both read and write

Problem - allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

Several variations of how readers and writers are treated – all involve priorities

Semaphore mutex = 1
Semaphore wrt = 1
Integer readcount = 0

The structure of a writer process

```
do {
    wait (wrt) ;
    // writing is performed
    signal (wrt) ;
} while (TRUE);
```

The structure of a reader process

```
do {
    wait (mutex) ;
    readcount ++ ;

    if (readcount == 1)
    wait (wrt) ;
    signal (mutex)

    // reading is performed


    wait (mutex) ;
    readcount - - ;

    if (readcount == 0)
    signal (wrt) ;
    signal (mutex) ;

} while (TRUE);
```

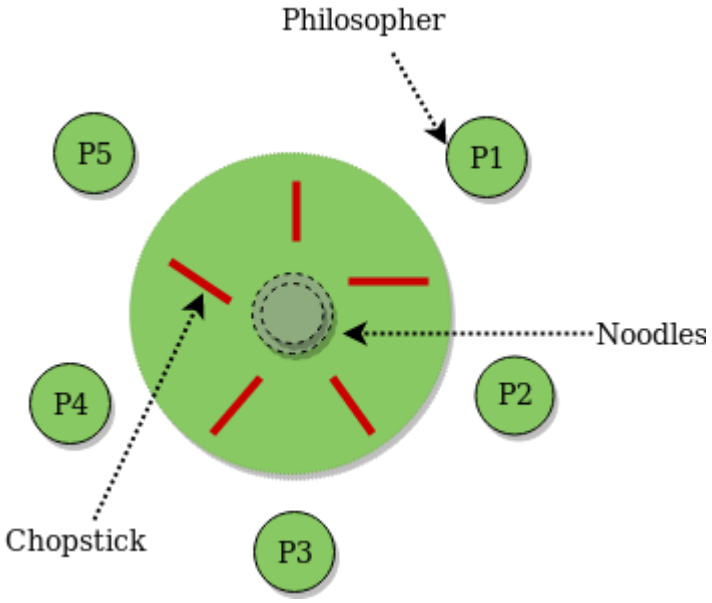
Dining Philosopher problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers.

There is one chopstick between each philosopher.

A philosopher may eat if he can pickup the two chopsticks adjacent to him.

One chopstick may be picked up by any one of its adjacent followers but not both.



The structure of Philosopher

```
process P[i]
   while true do
   {
    THINK;
    PICKUP (CHOPSTICK[i], CHOPSTICK [i+1 mod 5]);
    EAT;
    PUTDOWN (CHOPSTICK[i], CHOPSTICK [i+1 mod 5])
   }
```

There are three states of philosopher: THINKING, HUNGRY and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers.

Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher.

But, semaphores can result in deadlock due to programming errors.

CRITICAL REGIONS

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur.

We review the semaphore solution to the critical-section problem.

1. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

2. Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed, resulting in the following execution:

Signal (mutex);
...
Critical section
...
Wait (mutex);

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

3. Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

Wait (mutex);
...
Critical section
...
Wait (mutex);

In this case, a deadlock will occur.

4. Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

To avoid these errors, High-level synchronization constructed called Critical Region.

Critical Region

A critical region is a section of code that is always executed under mutual exclusion
A shared variable v of type T is declared as:

v : shared T;

variable v accessed only inside statement

region v when B do S;

where B is a Boolean expression.
while statement S is being executed, no other process can access variable v.

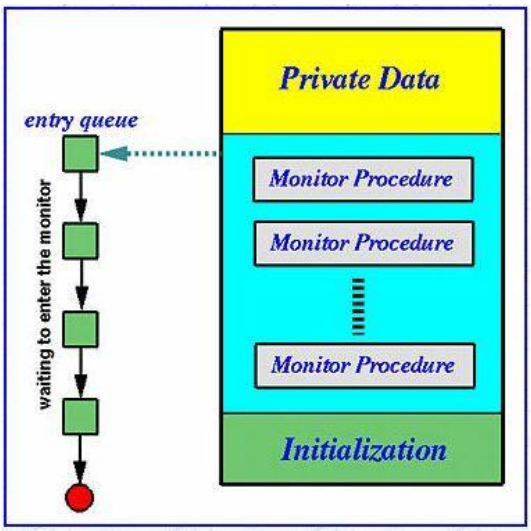
When a process tries to execute the region statement, the Boolean expression B is evaluated.
If B is true, statement S is executed.
If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.
If two statement,

region v when B do S1;
region v when B do S2;

are executed concurrently, in distinct sequential execution. i.e., S1 followed by S2 or S2 followed by S1.
Thus critical region construction makes simple that could occur due to the usage of semaphore.

MONITORS

Monitor is one of the ways to achieve Process synchronization.
Monitor is supported by programming languages to achieve mutual exclusion between processes.
Monitor is an abstract data type.
It is the collection of condition variables and procedures combined together in a special kind of module or a package.
The processes running outside the monitor can't access the internal variable of monitor but can call procedures of the monitor.
Only one process at a time can execute code inside monitors.
When any process wants to access the shared variables in the monitor, it needs to access it through the procedures.



A monitor has four components as shown below:

1. initialization,
2. private data,
3. monitor procedures,
4. monitor entry queue.

The initialization component contains the code that is used exactly once when the monitor is created.
The private data section contains all private data, including private procedures that can only be used within the monitor. Thus, these private items are not visible from outside of the monitor.
The monitor procedures are procedures that can be called from outside of the monitor.
The monitor entry queue contains all threads that called monitor procedures but have not been granted permissions.

The syntax of monitor is as follow:

```
monitor monitor_name
{
    //shared variable declarations

    procedure P1 ( . . . ) {
        // stmt;
        }

    procedure P2 ( . . . ) {
        // stmt;
        }

    procedure Pn ( . . . ) {
        // stmt;
        }

    initialization code ( . . . )
        {
        }
}
```

Condition Variables

Two different operations are performed on the condition variables of the monitor.

- Wait.
- signal.

condition x, y; //Declaring variable

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

DEADLOCK

A process requests resources. If the resources are not available at that time, the process enters a wait state. Waiting processes may never change state again because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

SYSTEM MODEL

A process must request a resource before using it, and must release resource after using it.

1. **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource
3. **Release:** The process releases the resource.

DEADLOCK CHARACTERIZATION

Four Necessary conditions for a deadlock

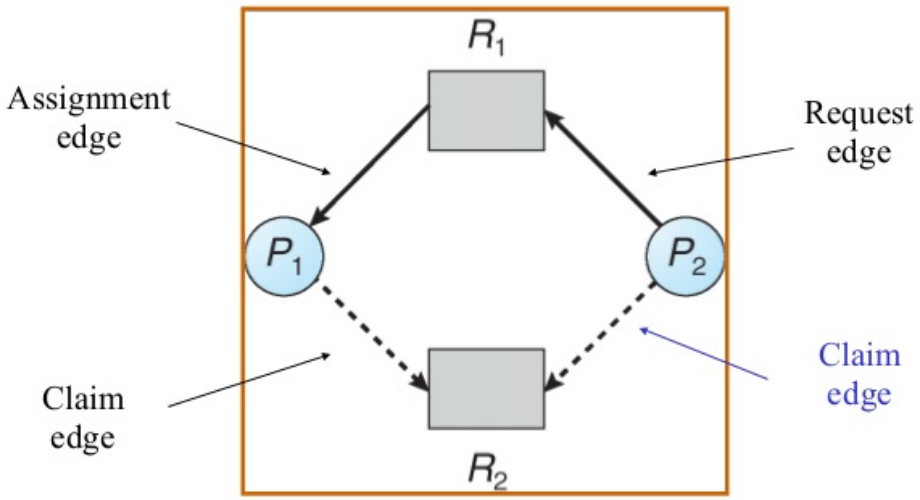
1. **Mutual exclusion:** At least one resource must be held in a non sharable mode. That is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted.
4. **Circular wait:** A set waiting process = {P0, P1, P2, …. Pn} must exit such that,
    - P0 is waiting for a resource that is held by P1,
    - P1 is waiting for a resource that is held by P2...Pn-1.

Resource-Allocation Graph

It is a Directed Graph with a set of vertices V and set of edges E.

V is partitioned into two types:

1. Process type P = {P1, P2, ... Pn}
2. Resource type R = {R1, R2, ... Rm}



Pi-->Rj - request => **request edge**

Rj-->Pi - allocated => **assignment edge**

Pi-->Rj – request(dashed) => **claim edge**

Pi is denoted as a circle and Rj as a square.

Example: Resource-Allocation Graph

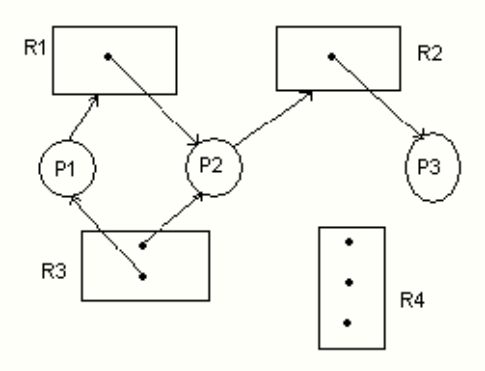
Rj may have more than one instance represented as a dot with in the square.

Sets P, R and E.

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E= {P1->R1, P2->R3, R1->P2, R2->P1, R3->P3}



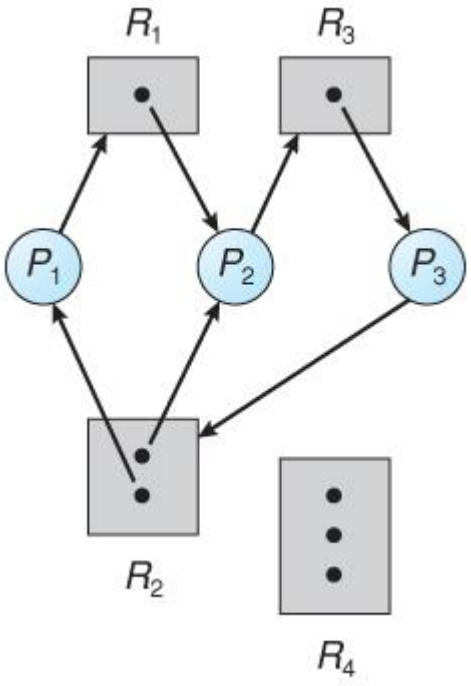
Resource instances

- One instance of resource type R1,
- Two instance of resource type R2,
- One instance of resource type R3,
- Three instances of resource type R4.

Process states

- Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and R2 and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.

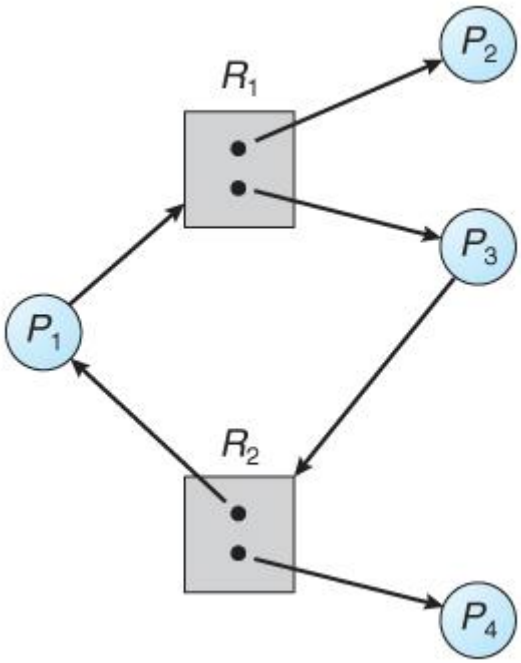
Example: Resource-Allocation Graph with deadlock



Two Cycles exist in the system

P1->R1->P2->R3->P3->R2->P1
P2->R3->P3->R2->P2

Example: Resource-Allocation Graph with no deadlock



METHODS FOR HANDLING DEADLOCKS

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

DEADLOCK PREVENTION

This ensures that the system never enters the deadlock state.

Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold.

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. **Denying Mutual exclusion**

   Mutual exclusion condition must hold for non-sharable resources. Printer cannot be shared simultaneously shared by prevent processes.

   Sharable resource - example Read-only files. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

2. **Denying Hold and wait**

   Whenever a process requests a resource, it does not hold any other resource.

   One technique that can be used requires each process to request and be allocated all its resources before it begins execution.

   Another technique is before it can request any additional resources, it must release all the resources that it is currently allocated.

These techniques have two main disadvantages:

- First, resource utilization may be low, since many of the resources may be allocated but unused for a long time. We must request all resources at the beginning for both protocols.
- Starvation is possible.

3. Denying No preemption

If a Process is holding some resources and requests another resource that cannot be immediately allocated to it. (that is the process must wait), then all resources currently being held are preempted. i.e. These resources are implicitly released.

The process will be restarted only when it can regain its old resources.

4. Denying Circular wait

Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.

Let R = {R1,R2,...Rm} be the set of resource types.

Assign to each resource type a unique integer number. If the set of resource types R includes tape drives, disk drives and printers.

F (tape drive) = 1,
F (disk drive) = 5,
F (Printer) = 12.

Each process can request resources only in an increasing order of enumeration.

DEADLOCK AVOIDANCE

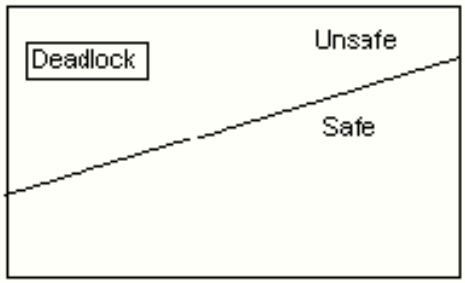
Deadlock avoidance request that the OS be given in advance additional information concerning which resources a process will request and use during its life time. With this information it can be decided for each request whether or not the process should wait.

To decide whether the current request can be satisfied or must be delayed, a system must consider the resources currently available, the resources currently allocated to each process and future requests and releases of each process.

Safe State

A state is safe if the system can allocate resources to each process in some order and still avoid a dead lock.

- A deadlock is an unsafe state.
- Not all unsafe states are dead locks.
- An unsafe state may lead to a dead lock.



Two algorithms are used for deadlock avoidance namely;

- **Resource Allocation Graph Algorithm -** single instance of a resource type.
- **Banker's Algorithm -** several instances of a resource type.

Resource allocation graph algorithm

Claim edge - Claim edge Pi---> Rj indicates that process Pi may request resource Rj at some time, represented by a dashed directed edge.

When process Pi request resource Rj, the claim edge Pi -> Rj is converted to a request edge.

Similarly, when a resource Rj is released by Pi the assignment edge Rj -> Pi is reconverted to a claim edge Pi -> Rj

Banker's algorithm

Available:

Indicates the number of available resources of each type.

Max:

Max[i, j]=k then process Pi may request at most k instances of resource type Rj

Allocation:

Allocation[i. j]=k, then process Pi is currently allocated K instances of resource type Rj

Need :

if Need[i, j]=k then process Pi may need K more instances of resource type Rj

Need [i, j] = Max[i, j] - Allocation[i, j]

Safety algorithm

1. Initialize work: = available and Finish [i]:=false for i=1,2,3 .. n
2. Find an i such that both
   a. Finish[i]=false
   b. Needi<= Work
   c. if no such i exists, goto step 4
3. Work: =work+ allocation i;
   a. Finish[i]:=true
   b. goto step 2
4. If finish[i] =true for all i, then the system is in a safe state

Resource Request Algorithm

1. Let Requesti be the request from process Pi for resources.
2. If Requesti<= Needi goto step2, otherwise raise an error condition, since the process has exceeded its maximum claim.
3. If Requesti <= Available, goto step3, otherwise Pi must wait, since the resources are not available.
   Available: = Available-Requesti;
   Allocationi: = Allocationi + Requesti
   Needi: = Needi - Requesti;

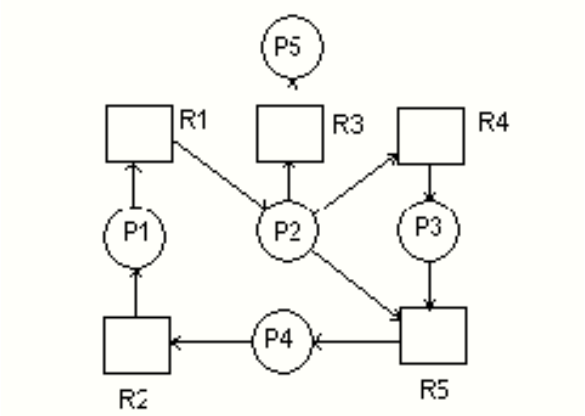
Now apply the safety algorithm to check whether this new state is safe or not.

If it is safe then the request from process Pi can be granted.

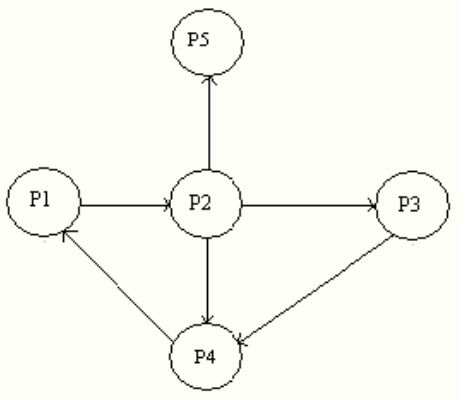
DEADLOCK DETECTION

Single instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that use a variant of resource-allocation graph called a wait for graph.



Resource Allocation Graph



Wait for S

Several Instance of a resource type

Available:

Number of available resources of each type

Allocation:

Number of resources of each type currently allocated to each process

Request :

Current request of each process

If Request [i,j]=k, then process Pi is requesting K more instances of resource type Rj.

1. Initialize work := available
   Finish[i]=false, otherwise finish [i]:=true
2. Find an index i such that both
   a. Finish[i]=false
   b. Requesti<=work
   if no such i exists go to step4.
3. Work:=work+allocationi
   Finish[i]:=true
   goto step2
4. If finish[i]=false then process Pi is deadlocked

DEADLOCK RECOVERY

1.Process Termination

Abort all deadlocked processes.

Abort one deadlocked process at a time until the deadlock cycle is eliminated. After each process is aborted , a deadlock detection algorithm must be invoked to determine where any process is still dead locked.

2.Resource Preemption

Preemptive some resources from process and give these resources to other processes until the deadlock cycle is broken.

- **Selecting a victim:** which resources and which process are to be preempted.
- **Rollback:** if we preempt a resource from a process it cannot continue with its normal execution. It is missing some needed resource. we must rollback the process to some safe state, and restart it from that state.
- **Starvation :** How can we guarantee that resources will not always be preempted from the same process.

TWO MARKS QUESTIONS WITH ANSWERS

1.Define Process.

Process is defined as

1. Program in execution
2. A synchronous activity.
3. The "animated spirit" of a procedure
4. The "locus of control of a procedure in execution which is manifested by the existence of a "process control block" in the operating system

That entity to which processors are assigned the dispatch able unit

2.What are the different process states available?

Running, if it currently has the CPU

Ready, if it could use a CPU if one were available

Blocked, if it is waiting for some event to happen before it can proceed

3.What is meant by Dispatching?

The Process of assignment of the CPU to the first process on the ready list is called as Dispatching.

4.What is FPCB?

FPCB is a data structure containing certain important information about the process including the following:

- Current state of the process
- Unique identification of the process A pointer to the process's parent
- A pointer to the process's child The process's priority
- Pointers to locate the process's memory and to allocated resources.

5.How is Blocked state different from others?

The Blocked state is different from others because the others are initiated by entities external to the process.

6.What are the different operations that can be performed on a process?

1) Create a process
2) Destroy a process
3) Change a process's priority
4) Wakeup a process
5) Enable a process to communicate with others
6) Suspend a process
7) Resume a process
8) Block a process

7.What is meant by Creating a Process?

Creating a process involves many operations including

1) Name the process
2) Insert it in the system's known processes list
3) Determine the process's initial priority
4) Create the process control block
5) Allocate the process's initial resources

8.What is means by Resuming a Process?

Resuming a process involves restarting it from the point at which it was suspended.

9.What is meant by Suspending a Process?

Suspending is often performed by the system to remove certain processes temporarily to reduce the system load during a peak loading situation.

10.What is meant by Context Switching?

When an interrupt occurs, the operating system saves the status of the interrupted process routes control to the appropriate first level interrupt handler.

11.What is meant by PSW?

Program Status Words (PSW) controls the order of instruction execution and contains various information about the state of a process. There are three types of PSW's namely

- Current PSW
- New PSW
- Old PSW

12.Define Mutual Exclusion.

Each process accessing the shared data excludes all others from doing simultaneously called as Mutual Exclusion.

13.What is meant by Co-operating process?

If a process can affect or be affected by the other processes executing in the system, that process which shares data with other process is called as Co-operating process.

14.What is meant by Interrupt?

An Interrupt is an event that alters the sequence in which a processor executes instructions. It is generated by the hardware of the computer System.

15.What is meant by Degree of Multiprogramming? And when it is said to be Stable?

Degree of Multiprogramming means the number of processes in memory. And it is said to be stable when the average rate of the number of process creation is equal to the average departure rate of processes leaving the system.

16.What is meant by CPU-bound process?

A CPU-bound process generates I/O requests infrequently using more of its time doing computation than an I/O processes. If all processes are CPU-bound, the I/O waiting queue will almost be empty and the devices will go unused and the system will be unbalanced.

17.What is meant by I/O-bound process?

An I/O-bound process spends more of its time doing I/O than it spends doing computations .If all processes are I/O-bound, the ready queue will almost be empty.

18.What is meant by Independent process?

A Process is Independent if it cannot affect or be affected by the other processes executing in the system. Here no process shares its data with other process available.

19.What is meant by Direct Communication?

In Direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the Send & Receive primitives are defined as

- send (p , message) - Send a message to process P
- receive (p , message) - Receive a message to process p

20.What is meant by Indirect Communication?

In Indirect Communication, the messages are sent to and received from mailboxes or ports. A mailbox is an object into which messages can be placed by processes and from which messages can be removed. In this scheme, the Send & Receive primitives are defined as:

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

21.What are benefits of Multiprogramming?

- Responsiveness Resource Sharing Economy
- Utilization of multiprocessor architectures.

22.What are the conditions that must hold for Deadlock Prevention?

Mutual Exclusion Condition Hold and Wait Condition No Pre-emption condition Circular Wait Condition.

23.What are the options for breaking a Deadlock?

- Simply abort one or more process to break the circular wait.
- Preempt some resources from one or more of the deadlocked processes.

24.What are the algorithms available for Deadlock avoidance?

1) Resource-Allocation Graph Algorithm
2) Banker's Algorithm
   a.Safety Algorithm
   b.Resource-Request Algorithm

25.What is a Monitor?

A Monitor is characterized by a set of programmer-defined operators. The representation of a Monitor type consists of declaration of variables whose value define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type.

26.What is meant by Counting Semaphore?

A Counting Semaphore is a semaphore whose integer value that can range between 0 & 1.

27.What is meant by Binary Semaphore?

A Binary Semaphore is a semaphore with an integer value that can range between 0 and 1. It can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture.

28.What is meant by Race Condition?

A condition, when several processes access and manipulate the same data on currently and the outcome of the execution depends on the particular order in which the access takes place is called as Race condition.

29.What does a solution for Critical-Section Problem must satisfy?

- Mutual Exclusion.
- Progress Bounded
- Waiting

30.What is meant by Indefinite Blocking or Starvation?

Indefinite Blocking is a situation where process waits indefinitely within the semaphore. This may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

31.What is meant by CPU Scheduler?

When the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed .This selection process is carried out by the CPU Scheduler.

32.What is meant by CPU Scheduling?

The process of selecting among the processes in memory that are ready to execute and allocates the CPU to one of them is called as CPU Scheduling.

33.What are the types of Scheduling available?

- Preemptive Scheduling
- Non - preemptive Scheduling Priority Scheduling

34.What is meant by Priority Scheduling?

The basic idea here is straight toward. Each process is assigned a priority and the run able process with the highest priority is allowed to run.

35.What is Preemptive Scheduling?

A Scheduling discipline is Pre-emptive if the CPU can be taken away before the process completes.

36.What is Non - Preemptive Scheduling?

A Scheduling discipline is non pre-emptive if once a process has been given the CPU, the CPU cannot be taken away from the process.

37.What are the properties of Scheduling Algorithms?

- CPU Utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

38.What is known as Resource Reservation in Real time Scheduling?

The Scheduler either admits the process, guaranteeing that the process will complete on time or rejects the request as impossible. This is known as Resource Reservation.

39.What is known as Priority inversion?

The high priority process would be waiting for a lower -priority one to finish. This situation is known as Priority Inversion.

40.What is meant by Dispatch latency?

The time taken by the dispatcher to stop one process and start another running is known as Dispatch Latency.

41.What is meant by Dispatcher?

It is a module that gives control of the CPU to the process selected by the short-term scheduler .This function involves

- Switching Context
- Switching to User Mode
- Jumping to the proper location in the user program to restart that program

42.What is meant by First Come, First Served Scheduling?

In this Scheduling, the process that requests the CPU first is allocated the CPU first. This Scheduling algorithm is Non Pre-emptive.

43.What is meant by Shortest Job First Scheduling?

When the CPU is available, it is assigned to the process that has the smallest next CPU burst. This Scheduling algorithm is either Pre-emptive or Non Pre-emptive.

44.What is meant by Priority Scheduling?

A Priority is associated with each process and the CPU is allocated to the process with the highest priority. This is also either Pre-emptive or Non Pre-emptive.

45.What is meant by Memory-Management Unit?

The run-time mapping form virtual to physical addresses is done by a hardware device is a called as Memory Management Unit.

46.What is meant by Input Queue?

The Collection of processes on the disk that is waiting to be brought into memory for execution forms the Input Queue.

47.What is Round-Robin Scheduling?

In Round-Robin Scheduling, processes are dispatched FIFO, but are given a limited amount of CPU time. If a process doesn't complete before it's CPU time expires, the CPU is Pre- emptied and given to the next waiting process. The Pre-emptied is then placed at the back of the ready list.

SIXTEEN MARK QUESTIONS WITH ANSWERS

1. What is meant by a process? Explain states of process with process state transition and PCB in detail.

- Process is a program in execution
- Process states with diagram
  - ✓ New
  - ✓ Wait
  - ✓ Running
  - ✓ Ready
  - ✓ Terminated
- Process Control block with diagram

2. Describe the Inter Process Communication in client server systems.

- Definition
- Message passing System
- Synchronization
  - ✓ Blocking send, Non-blocking receive
  - ✓ Non-blocking send, Blocking receive
  - ✓ Non-blocking send, Non-blocking receive
- Addressing
  - ✓ Direct Communication
  - ✓ Indirect Communication
- Buffering
  - ✓ Zero Capacity
  - ✓ Bounded Capacity
  - ✓ Unbounded Capacity
- Message Format

3. Explain about the various CPU scheduling algorithms.

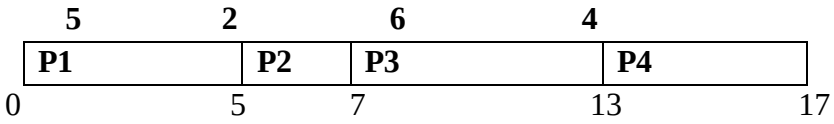
- First-come, first-served scheduling
- Shortest-job-first scheduling
- Priority Scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

4.Display the Gantt chart and find the Waiting Time, Turn Around Time, Average Waiting Time and Average Turn Around Time for the following process by using FCFS, Pre-emptive-SJF, Non-Pre-emptive Priority and Round Robin Scheduling algorithm, quantum=2.

PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY
P1	5	0	4
P2	2	1	1
P3	6	1	2
P4	4	2	3

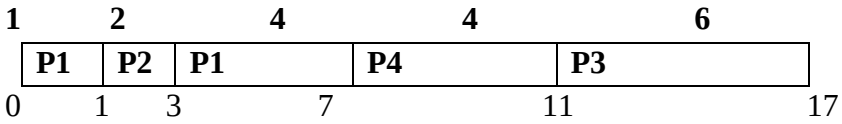
FCFS Scheduling:

Gantt chart



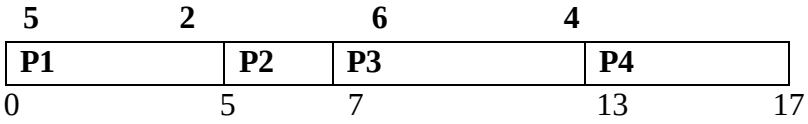
SJF Pre-emptive Scheduling:

Gantt chart



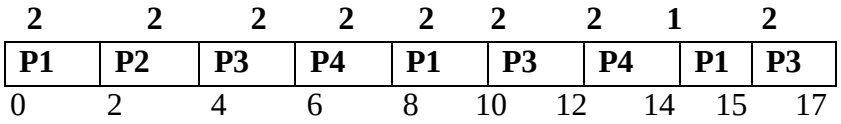
Non-Pre-emptive Priority Scheduling:

Gantt chart



Round Robin Scheduling:

Gantt chart



5. Write notes about multiple-processor scheduling and real-time scheduling.

Multiple-Processor Scheduling

- Approaches to Multiple-Processor Scheduling
- Load Balancing

Real Time Scheduling

- Characteristics
- Approaches

6. What is critical section problem and explain the solutions for this problem.

- Critical section syntax
- Solution to Critical Section Problem
    1. Mutual Exclusion
    2. Progress
    3. Bounded Waiting

7. Explain semaphores are their usage, implementation to avoid busy waiting and achieve synchronization

- Semaphore definition
- Usage for mutual exclusion and process synchronization
- Binary semaphores

8. Explain the classic problems of synchronization with the necessary algorithm

- The bounded-buffer problem with structure
- The readers-writers problem with structure
- The dining-philosophers problem with structure

9. Write about critical regions and monitor with algorithm.

Critical region definition

- Implementation of the conditional-region construct

Monitor definition

- Syntax of monitor
- Schematic view of monitors
- Monitor with condition variables

10. Give a detailed description about deadlocks and its characterization with resource allocation graph

- Deadlock definition
- Deadlock conditions
    - Mutual exclusion
    - Hold and wait
    - No pre-emption
    - Circular wait
    - Resource allocation graph

11. Explain about the methods used to prevent deadlocks.

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

12. Write in detail about deadlock avoidance with algorithms

- Safe state and safe sequence
- Diagram for safe, unsafe & deadlock states
- Resource-allocation graph algorithm

13. Explain the Banker's algorithm for deadlock avoidance with an example

- Deadlock avoidance definition
- Data structures used
- Safety algorithm
- Resource request algorithm

14. Give an account about deadlock detection with the algorithm

- Single instance of each resource type
- Wait-for graph
- Several instances of a resource type
- Detection-algorithm usage

15. Illustrate the Deadlock avoidance algorithm for the given snapshot. System consists of five processes (P0, P1, P2, P3, P4) and three resources (R1, R2, R3). Resource type R1 ha s10 instances, resource type R2 has 5 instances and R3 has 7 instances. The following snapshot of the system has been taken; find whether the system is in safe state.

Processes	Allocation			Max		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Solution:

Need

	A	B	C
P_0	6	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Work = 3 3 2

Finish

P_1 : Finish [1] = T; Work = 5 3 2
P_3 : Finish [3] = T; Work = 7 4 3
P_4 : Finish [4] = T; Work = 7 4 5
P_0 : Finish [0] = T; Work = 7 5 5
P_2 : Finish [2] = T; Work = 10 5 7

Therefore system is in safe state. Because processes can be executed in the sequence $<P_1P_3P_4P_0P_2>$.