OPERATING SYSTEMS
UNIT-2
CPU SCHEDULINGAND INTER PROCESSCOMMUNICATION

PROCESS SHEDULING:

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

PROCESS SCHEDULING OBJECTIVES

- Fairness
- Throughput Maximization
- Predictability
- Scheduling Overhead Minimization
- Graceful Degradation Under Heavy Load Conditions
- Pre-emptability

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.

TYPES OF SCHEDULING

There are two categories of scheduling:

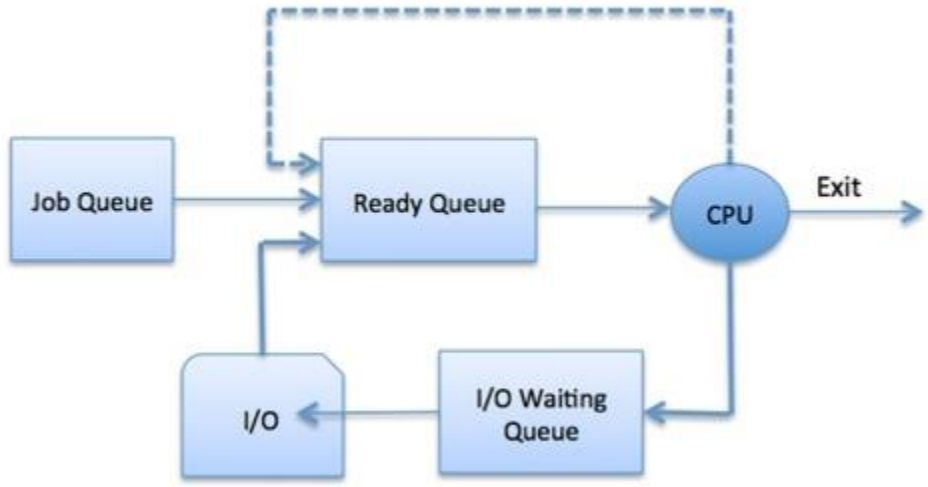
1. **Non-preemptive:** Here the resource can't be taken from a process until the process completes execution. The switching of resources occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

Process Scheduling Queues

The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



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

Schedulers

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

TYPES OF SCHEDULERS:

Schedulers are of three types −

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

Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

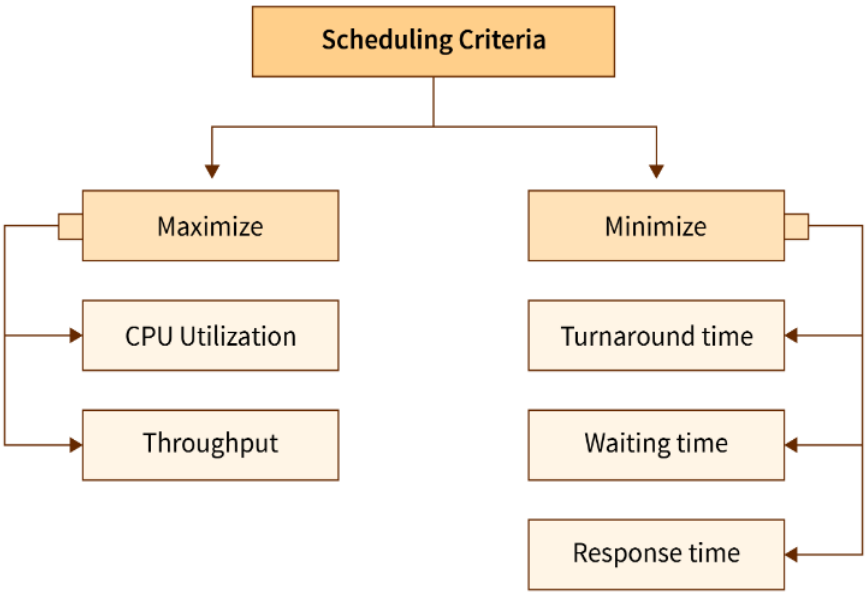
Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The

medium-term scheduler is in-charge of handling the swapped out-processes.

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

SHEDULING CRITERIA:



Maximize:

- **CPU utilization** - It makes sure that the CPU is operating at its peak and is busy.
- **Throughoutput** - It is the number of processes that complete their execution per unit of time.

Minimize:

- **Waiting time**- It is the amount of waiting time in the queue.
- **Response time**- Time retired for generating the first request after submission.
- **Turnaround time**- It is the amount of time required to execute a specific process.

Types of Scheduling Criteria in an Operating System

There are different CPU scheduling algorithms with different properties. The choice of algorithm is dependent on various different factors. There are many criteria suggested for comparing CPU schedule algorithms, some of which are:

- CPU utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

CPU utilization- The object of any CPU scheduling algorithm is to keep the CPU busy if possible and to maximize its usage. In theory, the range of CPU utilization is in the range of 0 to 100 but in real-time, it is actually 50 to 90% which relies on the system's load.

Throughput- It is a measure of the work that is done by the CPU which is directly proportional to the number of processes being executed and completed per unit of time. It keeps on varying which relies on the duration or length of processes.

Turnaround time- An important Scheduling criterion in OS for any process is how long it takes to execute a process. A turnaround time is the elapsed from the time of submission to that of completion. It is the summation of time spent waiting to get into the memory, waiting for a queue to be ready, for the I/O process, and for the execution of the CPU. The formula for.

TurnAroundTime=Compilationtime−Arrivaltime.

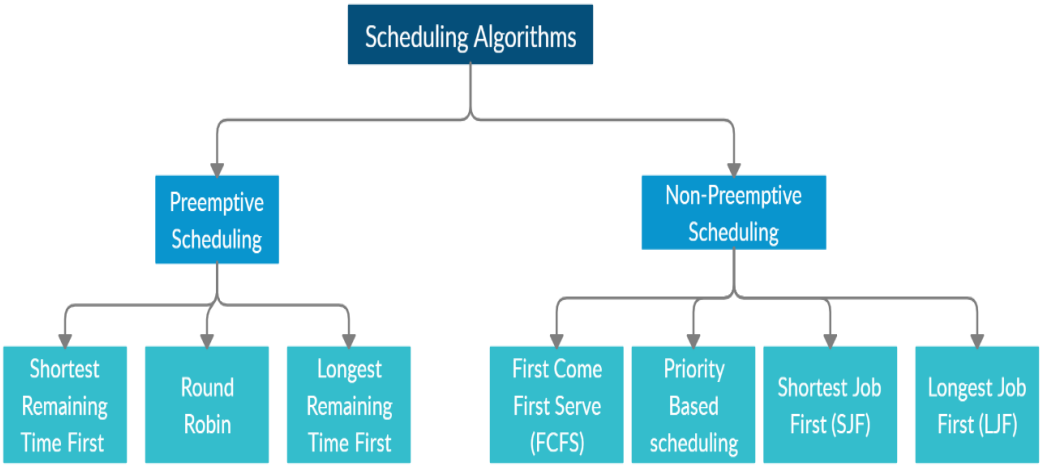
Waiting time- Once the execution starts, the scheduling process does not hinder the time that is required for the completion of the process. The only

II YEAR /IV SEMESTER: U20CBT406-OPERATING SYSTEMS NOTES SMVEC

thing that is affected is the waiting time of the process, i.e the time that is spent by a process waiting in a queue. The formula for calculating KaTeX parse error: Expected 'EOF', got '–' at position 31: …urnaround Time –_Burst Time.

Response time- Turnaround time is not considered as the best criterion for comparing scheduling algorithms in an interactive system. Some outputs of the process might produce early while computing other results simultaneously. Another criterion is the time that is taken from process submission to generate the first response. This is called response time and the formula for calculating it is, KaTeX parse error: Expected 'EOF', got '–' at position 79: …for the first) –_Arrival Time.

SHEDULING ALGORITHMS:



Preemptive Scheduling Algorithms

In these algorithms, processes are assigned with priority. Whenever a high-priority process comes in, the lower-priority process which has occupied the CPU is preempted. That is, it releases the CPU, and the high-priority process takes the CPU for its execution.

Non-Preemptive Scheduling Algorithms

In these algorithms, we cannot preempt the process. That is, once a process is running on CPU, it will release it either by context switching or terminating.

Often, these are the types of algorithms that can be used because of the limitation of the hardware.

Types:

There are various algorithms which are used by the Operating System to schedule the processes on the processor in an efficient way.

The Purpose of a Scheduling algorithm

1. Maximum CPU utilization
2. Fare allocation of CPU
3. Maximum throughput
4. Minimum turnaround time
5. Minimum waiting time
6. Minimum response time

There are the following algorithms which can be used to schedule the jobs.

1. First Come First Serve

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

2. Round Robin

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will get executed in the cyclic way. Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a preemptive type of scheduling.

3. Shortest Job First

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

4. Shortest remaining time first

It is the preemptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution.

5. Priority based scheduling

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is same then they will be scheduled according to their arrival time.

6. Highest Response Ratio Next

In this scheduling Algorithm, the process with highest response ratio will be scheduled next. This reduces the starvation in the system.

FCFS:

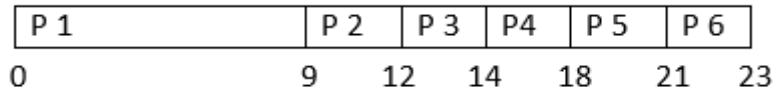
Example

S. No	Process ID	Process Name	Arrival Time	Burst Time
1	P 1	A	0	9
2	P 2	B	1	3
3	P 3	C	1	2
4	P 4	D	1	4
5	P 5	E	2	3
6	P 6	F	3	2

Non Pre Emptive Approach

Now, let us solve this problem with the help of the Scheduling Algorithm named First Come First Serve in a Non Preemptive Approach.

Gantt chart for the above Example 1 is:



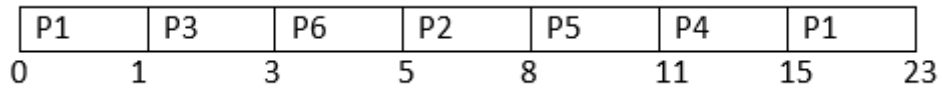
Turn Around Time = Completion Time - Arrival Time
Waiting Time = Turn Around Time - Burst Time

Solution to the Above Question Example 1

S. No	Process ID		Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	P 1	A	0	9	9	9	0
2	P 2	B	1	3	12	11	8
3	P 3	C	1	2	14	13	11
4	P 4	D	1	4	18	17	13
5	P 5	E	2	3	21	19	16
6	P 6	F	3	2	23	20	18

The Average Completion Time is:
Average CT = (9 + 12 + 14 + 18 + 21 + 23) / 6
Average CT = 97 / 6
Average CT = 16.16667
The Average Waiting Time is:
Average WT = (0 + 8 + 11 + 13 + 16 + 18) /6
Average WT = 66 / 6
Average WT = 11
The Average Turn Around Time is:
Average TAT = (9 + 11 + 13 + 17 + 19 +20) / 6
Average TAT = 89 / 6
Average TAT = 14.83334
This is how the FCFS is solved in Non Pre Emptive Approach.
Now, let us understand how they can be solved in Pre Emptive Approach

Pre Emptive Approach
Now, let us solve this problem with the help of the Scheduling Algorithm named First Come First Serve in a Pre Emptive Approach.
In Pre Emptive Approach we search for the best process which is available
Gantt chart for the above Example 1 is:



S. No	Process ID		Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	P 1	A	0	9	23	23	1
2	P 2	B	1	3	8	7	2
3	P 3	C	1	2	3	2	3
4	P 4	D	1	4	15	14	4
5	P 5	E	2	3	11	9	5
6	P 6	F	3	2	5	2	6

The Average Completion Time is:
Average CT = (23 + 8 + 3 + 15 + 11 + 5) / 6
Average CT = 65 / 6
Average CT = 10.83333
The Average Waiting Time is:
Average WT = (14 + 4 + 0 + 10 + 7 + 0) /6

Average WT = 35 / 6
Average WT = 5.83333
The Average Turn Around Time is:
Average TAT = (23 + 7 + 2 + 14 + 9 +2) / 6
Average TAT = 57 / 6
Average TAT = 9.5
This is how the FCFS is solved in Pre Emptive Approach.
SJF:
The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be preemptive or non-preemptive.
Example-1: Consider the following table of arrival time and burst time for five processes **P1, P2, P3, P4** and **P5**.

Process	**Burst Time**	**Arrival Time**
P1	6 ms	2 ms
P2	2 ms	5 ms
P3	8 ms	1 ms
P4	3 ms	0 ms
P5	4 ms	4 ms

The Shortest Job First CPU Scheduling Algorithm will work on the basis of steps as mentioned below:
At time = 0,

- Process P4 arrives and starts executing

Time Instance	**Process**	**Arrival Time**	**Waiting Table**	**Execution Time**	**Initial Burst Time**	**Remaining Burst Time**
0-1ms	P4	0ms		1ms	3ms	2ms

At time= 1,

- Process P3 arrives.
- But, as P4 still needs 2 execution units to complete.
- Thus, P3 will wait till P4 gets executed.

Time Instance	**Process**	**Arrival Time**	**Waiting Table**	**Execution Time**	**Initial Burst Time**	**Remaining Burst Time**
1-2ms	P4	0ms		1ms	2ms	1ms
	P3	1ms	P3	0ms	8ms	8ms

At time =2,

- Process P1 arrives and is added to the waiting table
- P4 will continue its execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
2-3ms	~~P4~~	~~0ms~~		~~1ms~~	~~1ms~~	~~0ms~~
	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3, P1	0ms	6ms	6ms

At time = 3,

- Process P4 will finish its execution.
- Then, the burst time of P3 and P1 is compared.
- Process P1 is executed because its burst time is less as compared to P3.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	6ms	5ms

At time = 4,

- Process P5 arrives and is added to the waiting Table.
- P1 will continue execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
4-5ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	5ms	4ms
	P5	4ms	P3, P5	0ms	4ms	4ms

At time = 5,

- Process P2 arrives and is added to the waiting Table.
- P1 will continue execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
5-6ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	4ms	3ms
	P5	4ms	P3, P5	0ms	4ms	4ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
	P2	5ms	P3, P5, P2	0ms	2ms	2ms

At time = 6,

- Process P1 will finish its execution.
- The burst time of P3, P5, and P2 is compared.
- Process P2 is executed because its burst time is the lowest among all.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
6-9ms	P3	1ms	P3	0ms	8ms	8ms
	~~P1~~	~~2ms~~	~~P3~~	~~3ms~~	~~3ms~~	~~0ms~~
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms

At time=9,

- Process P2 is executing and P3 and P5 are in the waiting Table.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
9-11ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	~~P2~~	~~5ms~~	~~P3, P5~~	~~2ms~~	~~2ms~~	~~0ms~~

At time = 11,

- The execution of Process P2 will be done.
- The burst time of P3 and P5 is compared.
- Process P5 is executed because its burst time is lower than P3.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
11-15ms	P3	1ms	P3	0ms	8ms	8ms
	~~P5~~	~~4ms~~	~~P3~~	~~4ms~~	~~4ms~~	~~0ms~~

At time = 15,

- Process P5 will finish its execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
15-23ms	~~P3~~	~~1ms~~		~~8ms~~	~~8ms~~	~~0ms~~

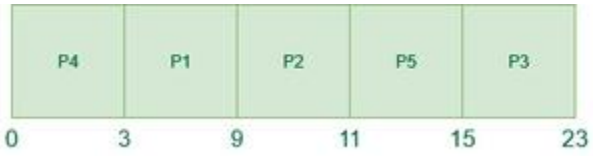
At time = 23,

- Process P3 will finish its execution.
- The overall execution of the processes will be as shown below:

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
0-1ms	P4	0ms		1ms	3ms	2ms
1-2ms	P4	0ms		1ms	2ms	1ms
	P3	1ms	P3	0ms	8ms	8ms
2-3ms	~~P4~~	~~0ms~~		~~1ms~~	~~1ms~~	~~0ms~~
	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3, P1	0ms	6ms	6ms
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	6ms	5ms
4-5ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	5ms	4ms
	P5	4ms	P3, P5	0ms	4ms	4ms
5-6ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	4ms	3ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms
6-9ms	P3	1ms	P3	0ms	8ms	8ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
	~~P1~~	~~2ms~~	~~P3~~	~~3ms~~	~~3ms~~	~~0ms~~
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms
9-11ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	~~P2~~	~~5ms~~	~~P3, P5~~	~~2ms~~	~~2ms~~	~~0ms~~
11-15ms	P3	1ms	P3	0ms	8ms	8ms
	~~P5~~	~~4ms~~	~~P3~~	~~4ms~~	~~4ms~~	~~0ms~~
15-23ms	~~P3~~	~~1ms~~		~~8ms~~	~~8ms~~	~~0ms~~

Gantt chart for above execution:



Gantt chart

Now, let's calculate the average waiting time for above example:
P4 = 0 – 0 = 0
P1 = 3 – 2 = 1
P2 = 9 – 5 = 4
P5 = 11 – 4 = 7
P3 = 15 – 1 = 14
Average Waiting Time = 0 + 1 + 4 + 7 + 14/5 = 26/5 = 5.2
RR:

Round Robin CPU Scheduling is the most important CPU Scheduling Algorithm which is ever used in the history of CPU Scheduling Algorithms. Round Robin CPU Scheduling uses Time Quantum (TQ). The Time Quantum is something which is removed from the Burst Time and lets the chunk of process to be completed.

Examples:

1. S. No Process ID Arrival Time Burst Time
2. _
3. 1 P 1 0 7
4. 2 P 2 1 4
5. 3 P 3 2 15

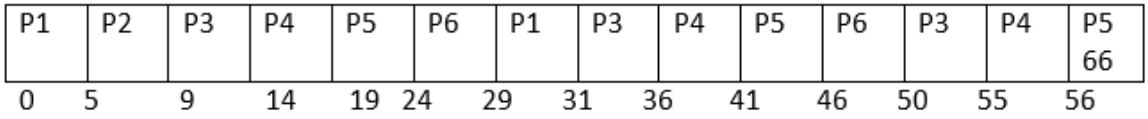
6. 4 P 4 3 11
7. 5 P 5 4 20
8. 6 P 6 4 9

Assume Time Quantum TQ = 5

Ready Queue:

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

Gantt chart:



Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
P1	0	7	31	31	24
P2	1	4	9	8	4
P3	2	15	55	53	38
P4	3	11	56	53	42
P5	4	20	66	62	42
P6	4	9	50	46	37

Average Completion Time

1. Average Completion Time = (31 +9 + 55 +56 +66 + 50) / 6
2. Average Completion Time = 267 / 6
3. Average Completion Time = 44.5

Average Waiting Time

1. Average Waiting Time = (5 + 26 + 5 + 42 + 42 + 37) / 6
2. Average Waiting Time = 157 / 6
3. Average Waiting Time = 26.16667

Average Turn Around Time

1. Average Turn Around Time = (31 + 8 + 53 + 53 + 62 + 46) / 6
2. Average Turn Around Time = 253 / 6
3. Average Turn Around Time = 42.16667

MULTIPROCESSOR SHEDULING :

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

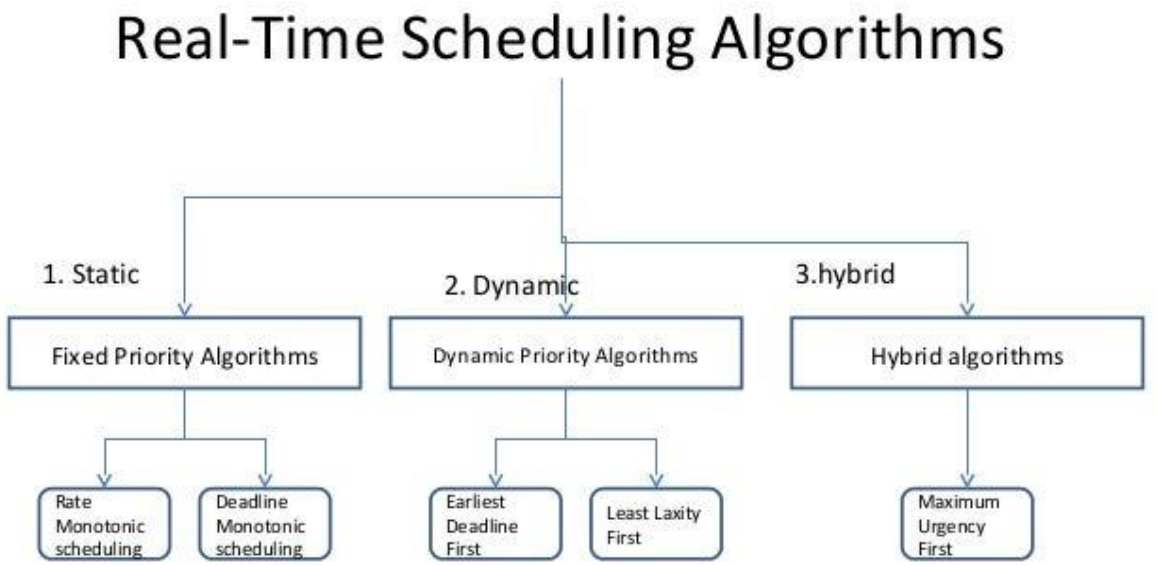
Approaches to Multiple-Processor Scheduling :

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

REAL TIME SHEDULING:

Real time scheduling is of two types: **Soft Real-Time scheduling** which does not guarantee when when a critical real-time process will be scheduled; **Hard Real-Time scheduling** in which the process must be scheduled before the deadline. In this post we will cover two real time scheduling algorithms: rate monotonic scheduling and earliest deadline first.



MULTIPROCESSOR SHEDULING:

Multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

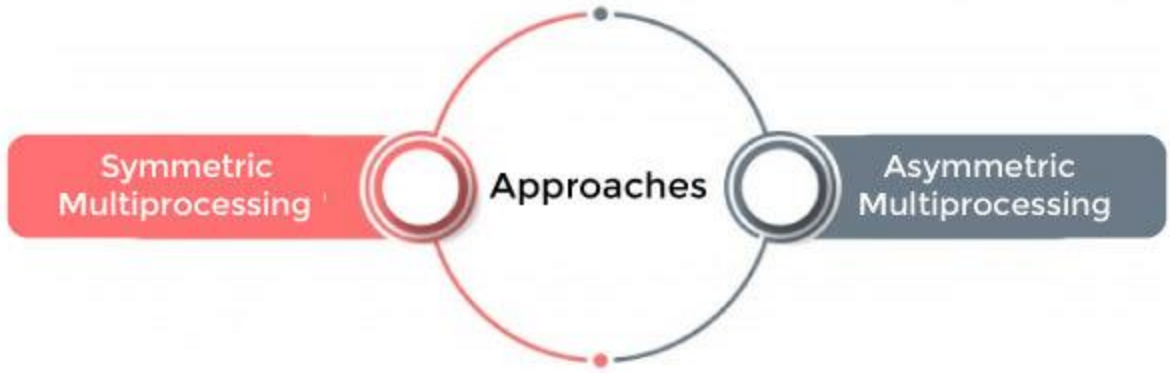
The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be **heterogeneous** (different kinds of CPUs) or **homogenous** (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one

Approaches to Multiple Processor Scheduling

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.



1. **Symmetric Multiprocessing:** It is used where each processor is **self-scheduling**. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.
2. **Asymmetric Multiprocessing:** It is used when all the scheduling decisions and I/O processing are handled by a single processor called the **Master Server**. The other processors execute only the **user code**. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing

REAL TIME SHEDULING:

Real-time systems are systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks.

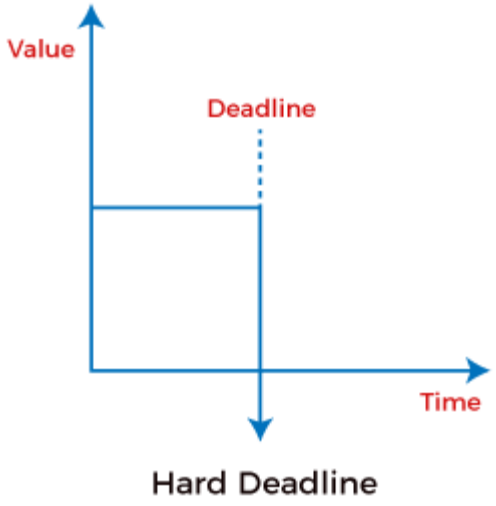
Types of Real-Time System

A real-time operating system is divided into two systems, such as:

1. Hard real-time system
2. Soft real-time system

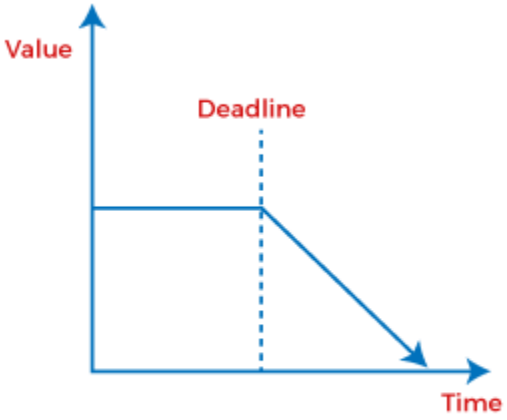
Hard Real-Time System

A hard real-time system considers timelines as a deadline, and it should not be omitted in any circumstances. Hard real-time Systems do not use any permanent memory, so their processes must be complete properly in the first time itself.



Soft Real-Time System

A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirement. In a soft real-time system, the meeting of deadline is not compulsory for every task, but the process should get processed and give the result. Even the soft real-time systems cannot miss the deadline for every task or process according to the priority it should meet the deadline or miss the deadline.



Soft Deadline

INTER-PROCESS COMMUNICATION:

A process can be of two types:

- Independent process.
- Co-operating process.

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

1. Shared Memory
2. Message passing

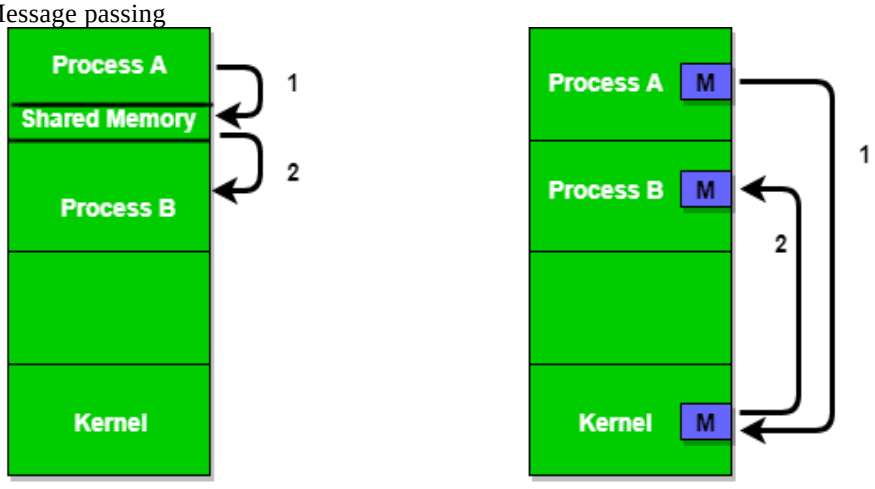


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

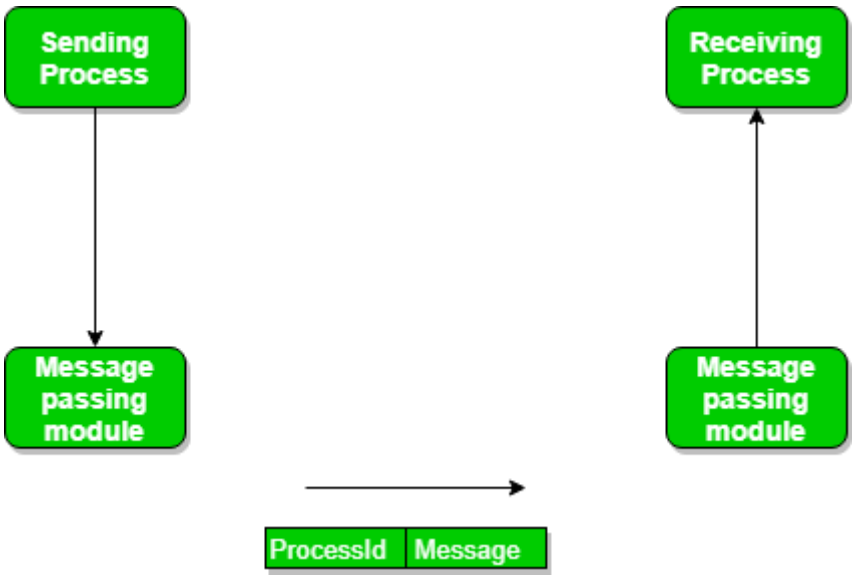
Ex: Producer-Consumer problem

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

ii) Messaging Passing Method

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

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



CONCURRENT PROCESSES:

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Concurrent means, which occurs when something else happens. The tasks are broken into subtypes, which are then assigned to different processors to perform simultaneously, sequentially instead, as they would have to be performed by one processor. Concurrent processing is sometimes synonymous with parallel processing. The term real and virtual concurrency in concurrent processing:

1. **Multiprogramming Environment:** In a multiprogramming environment, there are multiple tasks shared by one processor. While a virtual concept can be achieved by the operating system, if the processor is allocated for each individual task, the virtual concept is visible if each task has a dedicated processor.

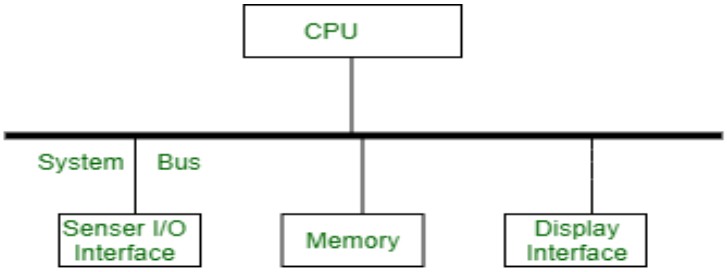


Figure - Multiprogramming (Single CPU) Environment

1. **Multiprocessing Environment :** In multiprocessing environment two or more processors are used with shared memory. Only one virtual address space is used, which is common for all processors. All tasks reside in shared memory. In this environment, concurrency is supported in the form of concurrently executing processors. The tasks executed on different processors are performed with each other through shared memory. The multiprocessing environment is shown in figure.

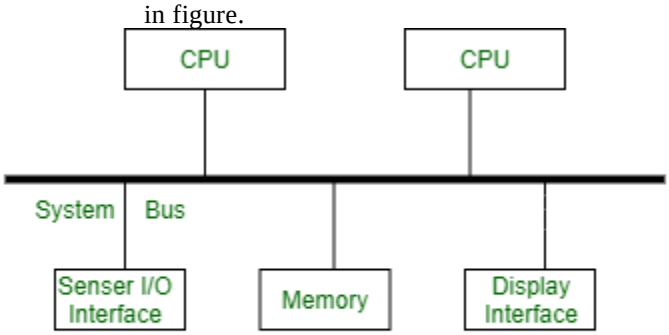


Figure - Multiprocessing Environment

2. **Distributed Processing Environment :** In a distributed processing environment, two or more computers are connected to each other by a communication network or high speed bus. There is no shared memory between the processors and each computer has its own local memory. Hence a distributed application consisting of concurrent tasks, which are distributed over network communication via messages. The distributed processing environment is shown in figure

.

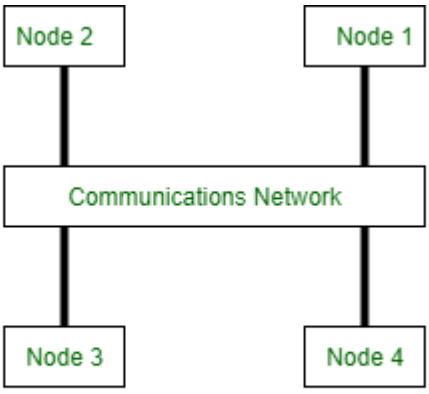


Figure - Distributed processing Environment

PRECEDENCE GRAPHS:

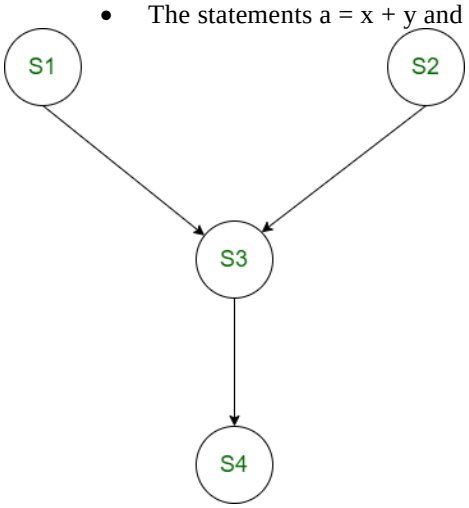
Precedence Graph is a directed acyclic graph which is used to show the execution level of several processes in operating system. It consists of nodes and edges. Nodes represent the processes and the edges represent the flow of execution.

Properties of Precedence Graph : Following are the properties of Precedence Graph:

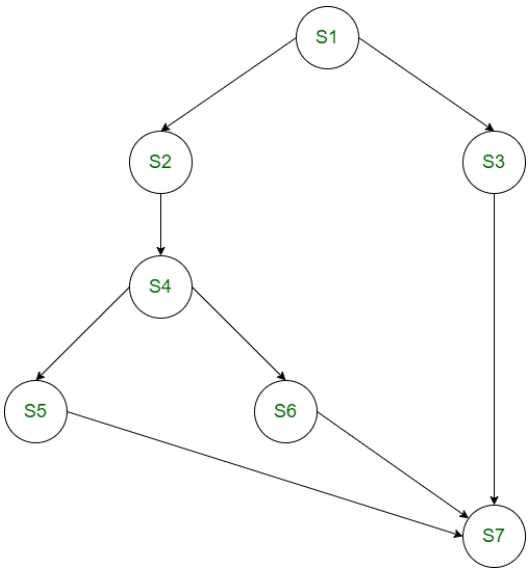
- It is a directed graph.
- It is an acyclic graph.
- Nodes of graph correspond to individual statements of program code.
- Edge between two nodes represents the execution order.
- A directed edge from node A to node B shows that statement A executes first and then Statement B executes.

Consider the following code:
S1 : a = x + y;
S2 : b = z + 1;
S3 : c = a - b;
S4 : w = c + 1;
If above code is executed concurrently, the following precedence relations exist:

- c = a – b cannot be executed before both a and b have been assigned values.
- w = c + 1 cannot be executed before the new values of c has been computed.
- The statements a = x + y and b = z + 1 could be executed concurrently.

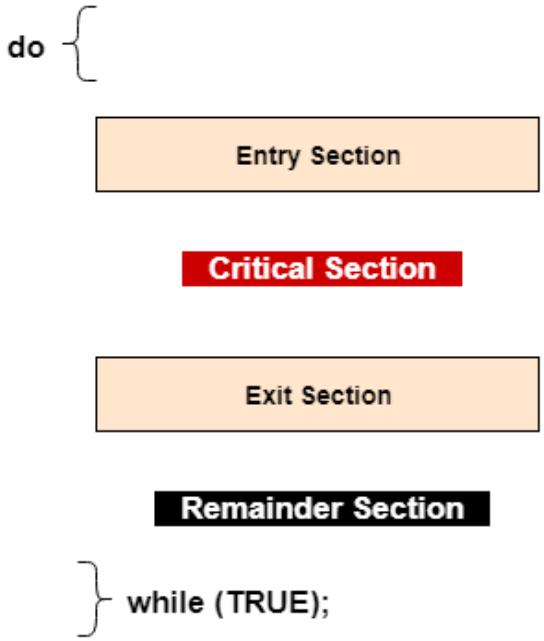


1. S2 and S3 can be executed after S1 completes.
2. S4 can be executed after S2 completes.
3. S5 and S6 can be executed after S4 completes.
4. S7 can be executed after S5, S6 and S3 complete.



CRITICAL SECTION:

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.
A diagram that demonstrates the critical section is as follows −



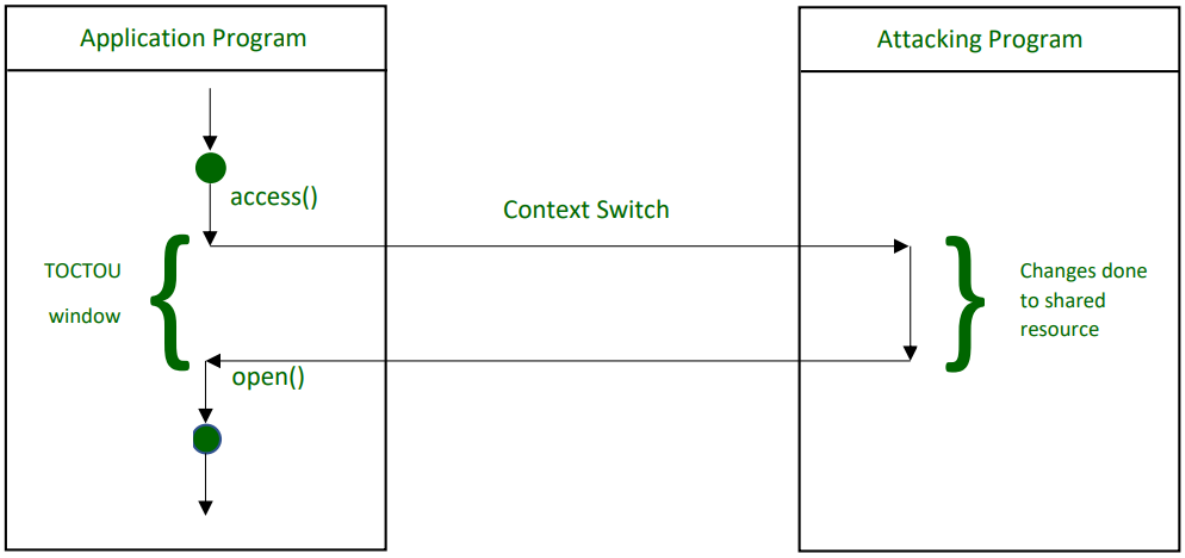
Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**
  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
- **Progress**
  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting**
  Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

RACE CONDITIONS:

Race condition occurs when multiple threads read and write the same variable i.e. they have access to some shared data and they try to change it at the same time. In such a scenario threads are “racing” each other to access/change the data.
This is a major security vulnerability [CWE-362], and by manipulating the timing of actions anomalous results might appear. This vulnerability arises during a TOCTOU (time-of-check, time-of-use) window.



1. **General Misconception –**
   A trivial cure to this vulnerability could be locking the file itself during this check-and-use window, because then no other process can use the file during the time window.
   Seems easy, then why isn't this practical? Why can't we use this approach to solve the race condition problem?
   The answer is simply that such a vulnerability could not actually be prevented with just locking the file.
2. **Problems while locking the file –**
   A file is locked out for other processes only if it is already in open state. This process is called check-and-open process and during this time it is impossible to lock a file. Any locks created can be ignored by the attacking or the malicious process.
   What actually happens is that the call to Open() does not block an attack on a locked file. When the file is available for a check-and-open process, the file actually is open to any access/ change. So it's impossible to lock a file at this point of time. This makes any kind of locks virtually non-existent to the malicious processes.
   Internally it is using the sleep_time which doubles at every attempt. More commonly this is referred to as a spinlock or the busy form of waiting. Also there is always a possibility of the file getting locked indefinitely i.e. danger of getting stuck in a deadlock.
3. **What would happen even if we were somehow able to lock the file?**
   Let's try to lock the file and see what could be the possible drawbacks. The most common locking mechanism that is available is atomic file locking. It is done using a lockfile to create a unique file on the same filesystem. We make use of link() to make a link to the lockfile for any kind of access to the file.
   - If link() returns 0, the lock is successful.

   The most common fix available is to store the PID of the application in the lock file, which is checked against the active PID at that time. Then again a flaw with this fix is that *f* PID may have been reused.
4. **Actual Solutions –**
   A better solution is to rather than creating locks on the file as a whole, lock the parts of the file to different processes.
   Example –
   When a process wants to write into a file, it first asks the kernel to lock that file or a part of it. As long as the process keeps the lock, no other process can ask to lock the sam part of the file. Hence you could see that issue with concurrency is getting resolved like this.

In the same way, a process asks for locking before reading the content of a file, which ensures no changes will be made as long as the lock is kept.
Differentiating these different kind of locks is done by the system itself. The system has the capability to distinguish between the locks required for file reading and those required for file writing. This kind of locking system is achieved by the flock() system call. Flock() call can have different values :

- LOCK_SH (lock for reading)
- LOCK_EX (for writing)
- LOCK_UN (release of the lock)

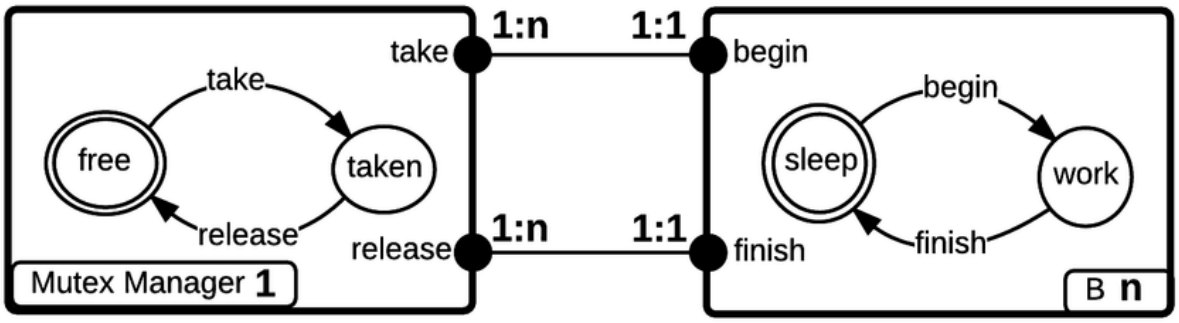
MUTUAL EXCLUSION:

Mutual exclusion also known as Mutex is a unit of code that avert contemporaneous access to shared resources. Mutual exclusion is concurrency control's property that is installed for the objective of averting race conditions. In simple words, it's a condition in which a thread of execution does not ever get involved in a critical section at the same time as a concurrent thread of execution so far using the critical section. This critical section can be a period for which the thread of execution uses the shared resource which can be defined as a data object, that different concurrent threads maybe attempt to alter (where the number of concurrent read operations allowed is two but on the other hand two write or one read and write is not allowed, as it may guide it to **data instability**). Mutual exclusion in OS is designed so that when a write operation is in the process then another thread is not granted to use the very object before the first one has done writing on the critical section after that releases the object because the rest of the processes have to read and write it.

Necessary Conditions for Mutual Exclusion

There are four conditions applied to mutual exclusion, which are mentioned below :

- Mutual exclusion should be ensured in the middle of different processes when accessing shared resources. There must not be two processes within their critical sections at any time.
- Assumptions should not be made as to the respective speed of the unstable processes.
- The process that is outside the critical section must not interfere with another for access to the critical section.
- When multiple processes access its critical section, they must be allowed access in a finite time, i.e. they should never be kept waiting in a loop that has no limits



HARDWARE SOLUTION:

Hardware solutions have been proposed to accelerate neural network computations at various levels of computing domains from mobile to HPC systems.

Embedded devices are hardware solutions with limited resources that require very specific firmware solutions and hardly reusable. This is the case of Microcontroller Units (MCU). However, the current trend is toward abstraction in this area. There are some firmware frameworks that provides an standard layer that can be applied to multiple models or even manufacturers.

SEMAPHORES:

A **Semaphore** can be described as an object that consists of a counter, a waiting list of processes, Signal and Wait functions. The most basic use of semaphore is to initialize it to 1. When a thread want to enter a critical section, it calls down and enter the section. When another thread tries to do the same thing, the operation system will put it to the sleep because the value of semaphore is already zero due to previous call to down. When first thread is finished with the critical section, it calls up, which wakes up the other thread that's waiting to enter.

Logically semaphore S is an integer variable that, apart from initialization can only be accessed through two atomic operations :

- **Wait(S) or P :** If the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.
- **Signal(S) or V :** Increment the value of Semaphore

STRICT ALLOCATION:

Strict Alternation Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.

This approach can only be used for only two processes. In general, let the two processes be Pi and Pj. They share a variable called turn variable. The pseudo code of the program can be given as following.

For Process Pi

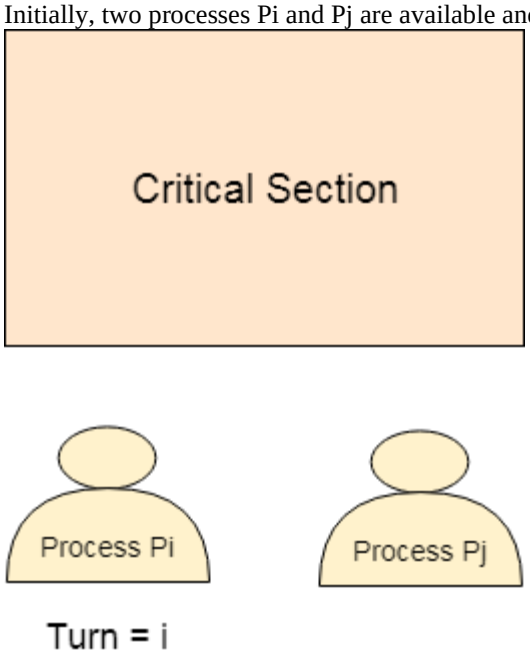
1. Non - CS
2. while (turn ! = i);
3. Critical Section
4. turn = j;
5. Non - CS

For Process Pj

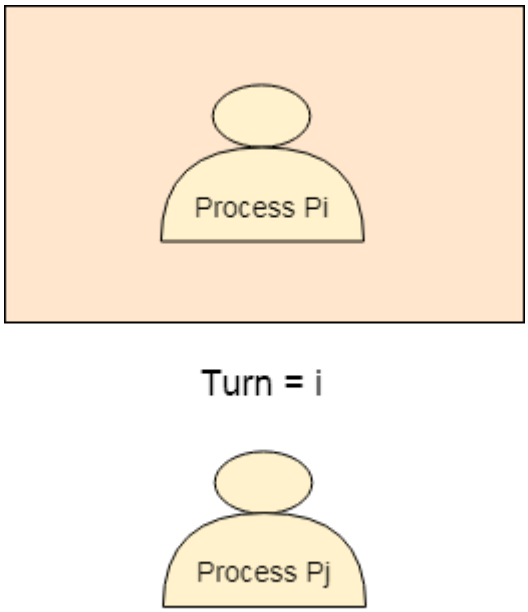
1. Non - CS
2. while (turn ! = j);
3. Critical Section
4. turn = i ;
5. Non - CS

The actual problem of the lock variable approach was the fact that the process was entering in the critical section only when the lock variable is 1. More than one process could see the lock variable as 1 at the same time hence the mutual exclusion was not guaranteed there.
This problem is addressed in the turn variable approach. Now, A process can enter in the critical section only in the case when the value of the turn variable equal to the PID of the process.
There are only two values possible for turn variable, i or j. if its value is not i then it will definitely be j or vice versa.

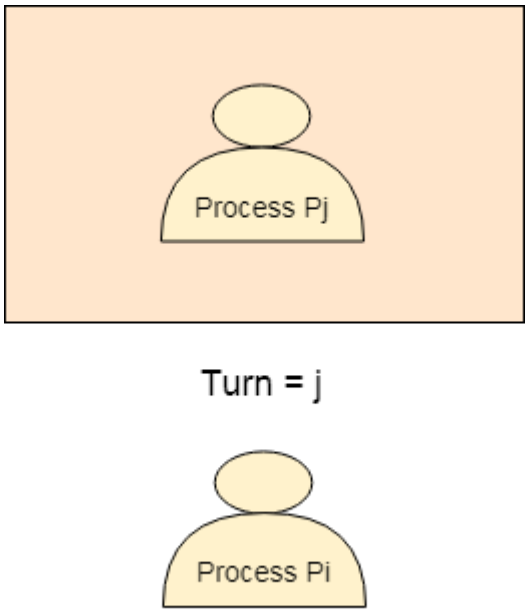
In the entry section, in general, the process Pi will not enter in the critical section until its value is j or the process Pj will not enter in the critical section until its value is i.
Initially, two processes Pi and Pj are available and want to execute into critical section.



The turn variable is equal to i hence Pi will get the chance to enter into the critical section. The value of Pi remains I until Pi finishes critical section.



Pi finishes its critical section and assigns j to turn variable. Pj will get the chance to enter into the critical section. The value of turn remains j until Pj finishes its critical section.



PETERSON's SOLUTION:

Peterson's solution ensures mutual exclusion. It is implemented in user mode and no hardware support is required therefore it can be implemented on any platform. Now Peterson's solution uses two variables: interest and Turn variable.

Now we will first see Peterson solution algorithm and then see how any two processes P and Q get mutual exclusion using Peterson solution.

```
#define N 2
#define TRUE 1
#define FALSE 0
int interested[N]=False
int turn;
void Entry_Section(int process)
{
  int other;
  other=1-process
  interested[process]= TRUE ;
  turn = process;
  while(interested[other]==TRUE && Turn=process);
}
void exit_section(int process)
{
  interested[process]=FALSE;}
```

Explanation

There will be two processes and the process number of the first process=0 and the process number of the second process is equal to 1.

So, if process 1 calls entry_section then other = 1-process =1-1=0.

If process 0 calls then other = 1-process = 1-0 = 1

Now, since the process which called entry_section it means that process want to execute a critical section then that process will set interested[process]=TRUE

So, if process 1 called entry section then interested[1]=TRUE

If process 0 is called entry section then interested[0]=TRUE

After declaring that process is interesting it will set its turn. So, if process 1 is called then turn =1.

Then, while (interested[other]==TRUE && Turn=process); will be executed.

In this line, the process checks whether other processes are interested or not. If that process is interested then interested[other]==TRUE will be true then the process thinks that it may happen that another process is executing the critical section.

For that, it will go into a loop until another process is not interesting. Now if another process becomes interested then interested[other]==TRUE

It will become False and the process will enter into a critical section. So, in this way, only one process may enter into the critical section. Therefore, mutual exclusion is guaranteed in Peterson's solution. While exiting the critical section process will set interest as False.

THE PRODUCER / CONSUMER PROBLEM:

The producer-consumer problem is an example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer that shares a common fixed-size buffer use it as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Problem: Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

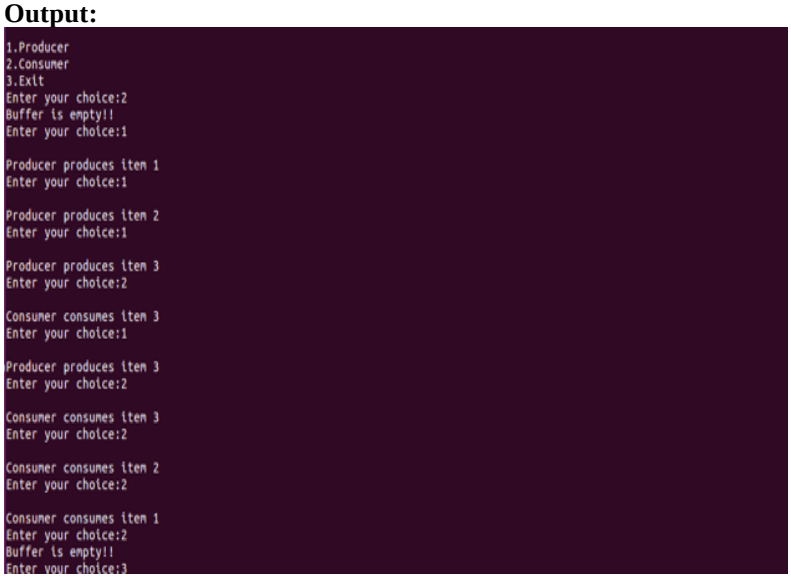
Note: An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Approach: The idea is to use the concept of parallel programming and Critical Section to implement the Producer-Consumer problem in C language using **OpenMP**.

```
// C program for the above approach
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer()
{
```

```
    --mutex;
    ++full;
    --empty;
    x++;
    printf("\nProducer produces" "item %d", x);}
 void consumer()
{
        --mutex;
        --full;
    ++empty;
    printf("\nConsumer consumes ""item %d", x);
    x--;
     ++mutex;}
 int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");
     for (i = 1; i > 0; i++) {
         printf("\nEnter your choice:");
        scanf("%d", &n);
         switch (n) {
        case 1:
            if ((mutex == 1)
              && (empty != 0)) {
              producer();}
            else {
              printf("Buffer is full!"); }
            break;
         case 2:
            if ((mutex == 1)
              && (full != 0)) {
              consumer(); }
            else {
              printf("Buffer is empty!");  }
            break;
         case 3:
           exit(0);
           break; }}}
```

Output:



EVENT COUNTERS:

EventCounter is .NET/.NET Core mechanism to publish and consume counters or statistics. EventCounters are supported in all OS platforms - Windows, Linux, and macOS. It can be thought of as a cross-platform equivalent for the PerformanceCounters that is only supported in Windows systems.

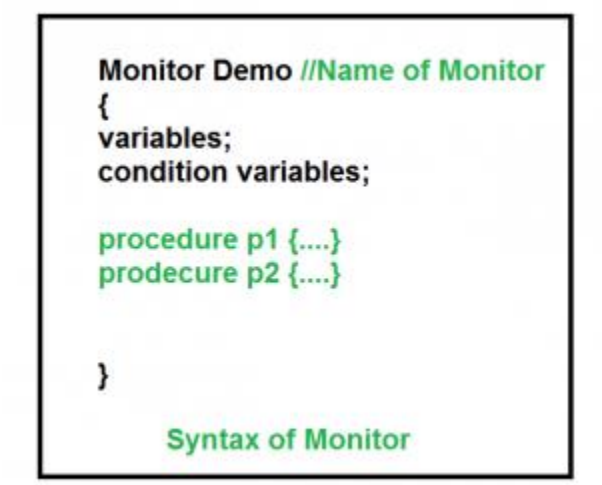
While users can publish any custom EventCounters to meet their needs, .NET Core LTS and higher runtime publishes a set of these counters by default. This document will walk through the steps required to collect and view EventCounters (system defined or user defined) in Azure Application Insights.

MONITORS:

Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.



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

- Wait.
- signal.

Advantages of Monitor: Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore. **Disadvantages of Monitor:** Monitors have to be implemented as part of the programming language . The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

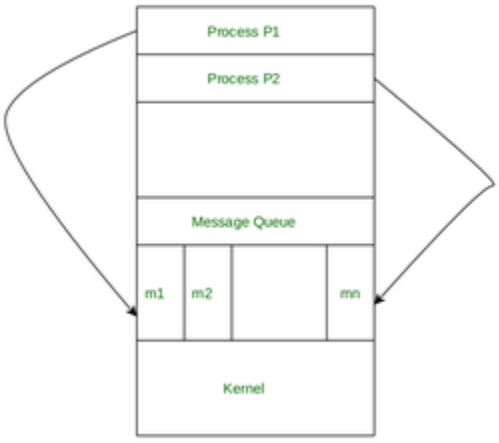
MESSAGE PASSING:

Message passing means how a message can be sent from one end to the other end. Either it may be a client-server model or it may be from one node to another node. The formal model for distributed message passing has two timing models one is synchronous and the other is asynchronous.

The fundamental points of message passing are:

1. In message-passing systems, processors communicate with one another by sending and receiving messages over a communication channel. So how the arrangement should be done?
2. The pattern of the connection provided by the channel is described by some topology systems.
3. The collection of the channels are called a network.

4. So by the definition of distributed systems, we know that they are geographically set of computers. So it is not possible for one computer to directly connect with some other node.
5. So all channels in the Message-Passing Model are private.
6. The sender decides what data has to be sent over the network. An example is, making a phone call.
7. The data is only fully communicated after the destination worker decides to receive the data. Example when another person receives your call and starts to reply to you.
8. There is no time barrier. It is in the hand of a receiver after how many rings he receives your call. He can make you wait forever by not picking up the call.
9. For successful network communication, it needs active participation from both sides.



Message Passing Model

Algorithm:

1. Let us consider a network consisting of n nodes named $p_{0,}\ p_{1,}\ p_{2}$…….p_{n-1} which are bidirectional point to point channels.
2. Each node might not know who is at another end. So in this way, the topology would be arranged.
3. Whenever the communication is established and whenever the message passing is started then only the processes know from where to where the message has to be sent.

CLASSICAL IPC PROBLEMS:

- Reader's & Writer Problem
- Dinning Philosopher Problem
- Barber's shop problem.

READER's WRITER PROBLEM:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows −

Reader Process

The code that defines the reader process is given below −

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.
. READ THE OBJECT
.
```

```
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.
The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When re becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process
The code that defines the writer process is given below:

```
wait(wrt);
.
. WRITE INTO THE OBJECT
.
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.

DINNING PHILOSOPHER PROBLEM:

Dining Philosophers Problem States that there are 5 Philosophers who are engaged in two activities Thinking and Eating. Meals are taken communally in a table with five plates and five forks in a cyclic manner as shown in the figure.

Constraints and Condition for the problem :

1. Every Philosopher needs two forks in order to eat.
2. Every Philosopher may pick up the forks on the left or right but only one fork at once.
3. Philosophers only eat when they had two forks. We have to design such a protocol i.e. pre and post protocol which ensures that a philosopher only eats if he or she had two forks.
4. Each fork is either clean or dirty.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.
The structure of the chopstick is shown below –
semaphore chopstick [5];
Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.
The structure of a random philosopher i is given as follows –

```
do {
  wait( chopstick[i] );
  wait( chopstick[ (i+1) % 5] );
  EATING THE RICE
```

```
  signal( chopstick[i] );
  signal( chopstick[ (i+1) % 5] );
  THINKING
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows −

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

BARBER's SHOP PROBLEM:

The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:

There is a barber shop with one barber and a number of chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

The problem is to write a program that coordinates the actions of the customers and the barber in a way that avoids synchronization problems, such as deadlock or starvation.

One solution to the Sleeping Barber problem is to use semaphores to coordinate access to the waiting chairs and the barber chair. The solution involves the following steps:

Initialize two semaphores: one for the number of waiting chairs and one for the barber chair. The waiting chairs semaphore is initialized to the number of chairs, and the barber chair semaphore is initialized to zero.

Customers should acquire the waiting chairs semaphore before taking a seat in the waiting room. If there are no available chairs, they should leave.

When the barber finishes cutting a customer's hair, he releases the barber chair semaphore and checks if there are any waiting customers. If there are, he acquires the barber chair semaphore and begins cutting the hair of the next customer in the queue.

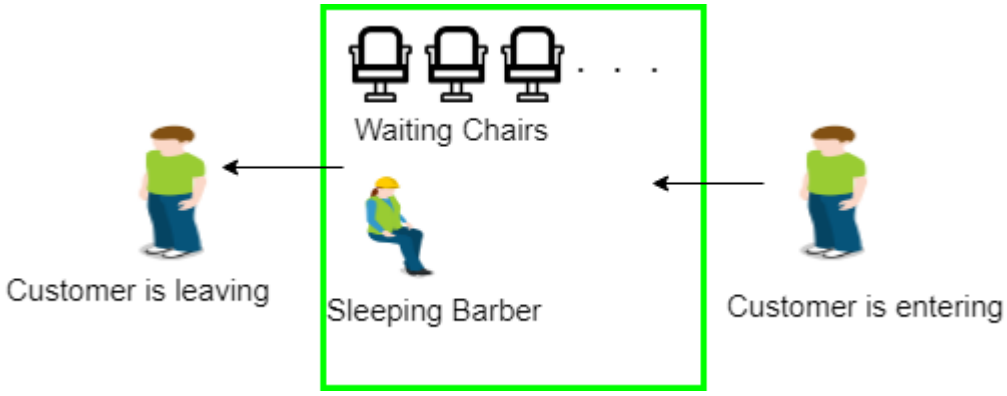
The barber should wait on the barber chair semaphore if there are no customers waiting.

The solution ensures that the barber never cuts the hair of more than one customer at a time, and that customers wait if the barber is busy. It also ensures that the barber goes to sleep if there are no customers waiting.

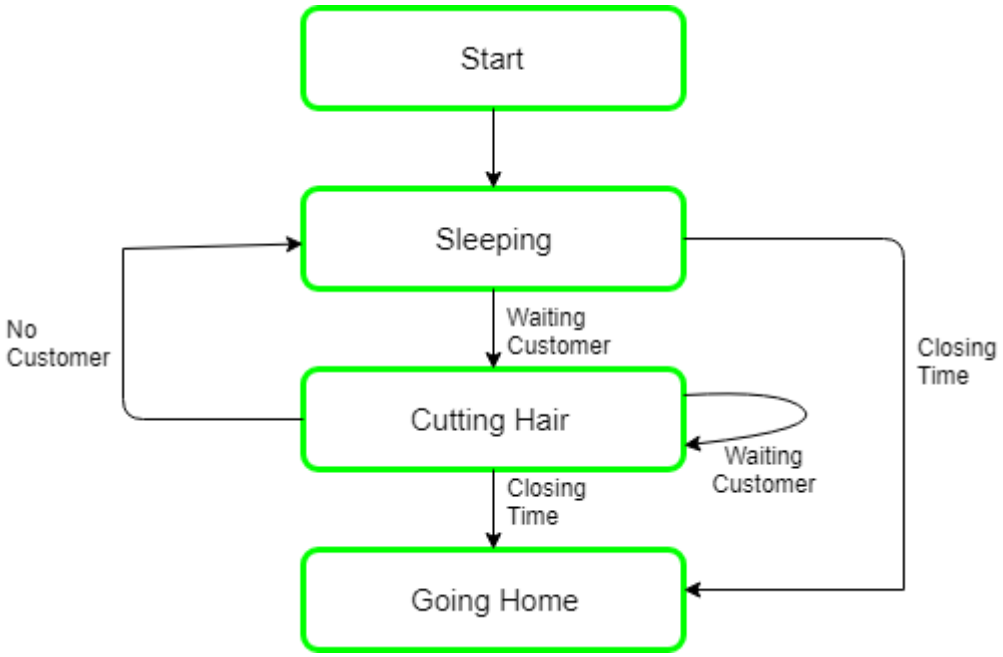
However, there are variations of the problem that can require more complex synchronization mechanisms to avoid synchronization issues. For example, if multiple barbers are employed, a more complex mechanism may be needed to ensure that they do not interfere with each other.

Prerequisite – Inter Process Communication **Problem :** The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



Solution : The solution to this problem includes three semaphores.First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop. When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up. When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex. If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



Algorithm for Sleeping Barber problem:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;
Barber {
while(true) {
down(Customers);
down(Seats);
FreeSeats++;
up(Barber);
up(Seats);
}
}
Customer {
while(true) {
down(Seats); //This line should not be here.
if(FreeSeats > 0) {
FreeSeats--;
up(Customers);
up(Seats);
down(Barber);
} else {
up(Seats);
}
}
}
```