Process Scheduling: Foundation and Scheduling objectives, Types of Schedulers, Scheduling criteria: CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time; Scheduling algorithms: Pre-emptive and Non pre-emptive, FCFS, SJF, RR; Multiprocessor scheduling: Real Time scheduling: RMand EDF.

Inter-process Communication: Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer/Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc.

PROCESS SCHEDULING:

CPU is always busy in **Multiprogramming**. Because CPU switches from one job to another job. But in **simple computers** CPU sit idle until the I/O request granted.

scheduling is a important OS function. All resources are scheduled before use.(cpu, memory, devices…..)

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 Objectives**

Maximize throughput.
Maximize number of users receiving acceptable response times.
Be predictable.
Balance resource use.
Avoid indefinite postponement.
Enforce Priorities.
Give preference to processes holding key resources

SCHEDULING QUEUES: people live in rooms. Process are present in rooms knows as queues. There are 3types

1. **job queue**: when processes enter the system, they are put into a **job queue**, which consists all processes in the system. Processes in the job queue reside on mass storage and await the allocation of main memory.
2. **ready queue**: if a process is present in main memory and is ready to be allocated to cpu forexecution, is kept in **readyqueue.**
3. **device queue**: if a process is present in waiting state (or) waiting for an i/o event to complete is said to bein device queue.(or)

The processes waiting for a particular I/O device is called device queue.

Schedulers : There are 3 schedulers

1. Long term scheduler.
2. Medium term scheduler
3. Short term scheduler.

Scheduler duties:

- Maintains the queue.
- Select the process from queues assign to CPU.

Types of schedulers

1. Long term scheduler:
select the jobs from the job pool and loaded these jobs into main memory (ready queue).
Long term scheduler is also called job scheduler.

2. Short term scheduler:

select the process from ready queue, and allocates it to the cpu.
If a process requires an I/O device, which is not present available then process enters device queue.
short term scheduler maintains ready queue, device queue. Also called as cpu scheduler.

3. Medium term scheduler: if process request an I/O device in the middle of the execution, then the process removed from the main memory and loaded into the waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations pe

Comparison between 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.

Context Switch: Assume, main memory contains more than one process. If cpu is executing a process, if time expires or if a high priority process enters into main memory, then the scheduler saves information about current process in the PCB and switches to execute the another process. The concept of moving CPU by scheduler from one process to other process is known as context switch.

Non-Preemptive Scheduling: **CPU** is assigned to one process, CPU do not release until the competition of that process. The CPU will assigned to some other process only after the previous process has finished.

Preemptive scheduling: here CPU can release the processes even in the middle of the execution.

CPU received a signal from process p2. OS compares the priorities of p1 ,p2. If p1>p2, CPU continues the execution of p1. If p1<p2 CPU preempt p1 and assigned to p2.

Dispatcher: The main job of dispatcher is switching the cpu from one process to another process. Dispatcher connects the cpu to the process selected by the short term scheduler.

Dispatcher latency: The time it takes by the dispatcher to stop one process and start another process is known as dispatcher latency. If the dispatcher latency is increasing, then the degree of multiprogramming decreases.

SCHEDULING CRITERIA:

1. **Throughput**: how many jobs are completed by the cpu with in a timeperiod.
2. **Turn around time** : The time interval between the submission of the process and time of the completion is turn around time.

TAT = Waiting time in ready queue + executing time + waiting time in waiting queue for I/O.

3. **Waiting time**: The time spent by the process to wait for cpu to beallocated.
4. **Response time**: Time duration between the submission and firstresponse.
5. **Cpu Utilization**: CPU is costly device, it must be kept as busy aspossible.

Eg: CPU efficiency is 90% means it is busy for 90 units, 10 units idle.

CPU SCHEDULINGALGORITHMS:

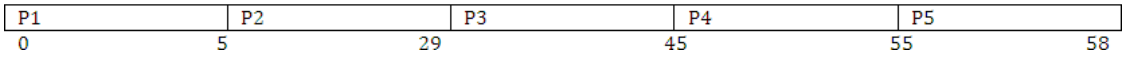
1. First come First served scheduling: (FCFS): The process that request the CPU first is holds the cpu first. If a process request the cpu then it is loaded into the ready queue, connect CPU to that process.

Consider the following set of processes that arrive at time 0, the length of the cpu burst time given in milli seconds.

burst time is the time, required the cpu to execute that job, it is in milli seconds.

Process	Burst time(milliseconds)
P1	5
P2	24
P3	16
P4	10
P5	3

Chart:



Average turn around time:

Turn around time= waiting time + burst time

Turn around time for p1= 0+5=5.
Turn around time for p2=5+24=29 Turn around time for p3=29+16=45 Turn around time for p4=45+10=55 Turn around time for p5= 55+3=58
Average turn around time= (5+29++45+55+58/5) = 187/5 =37.5 millisecounds

Average waiting time:

waiting time= starting time- arrival time

Waiting time for p1=0
Waiting time for p2=5-0=5
Waiting time for p3=29-0=29
Waiting time for p4=45-0=45
Waiting time for p5=55-0=55
Average waiting time= 0+5+29+45+55/5 = 125/5 = 25 ms.

Average Response Time :

Formula : First Response - Arrival Time Response Time for P1 =0
Response Time for P2 => 5-0 = 5
Response Time for P3 => 29-0 = 29
Response Time for P4 => 45-0 = 45
Response Time for P5 => 55-0 = 55

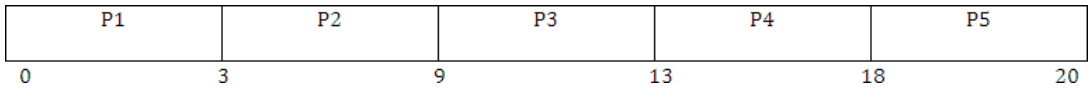
Average Response Time => (0+5+29+45+55)/5 =>25ms

1) First Come FirstServe:

It is Non Primitive Scheduling Algorithm.

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8

Process arrived in the order P1, P2, P3, P4, P5.
P1 arrived at 0 ms.
P2 arrived at 2 ms.
P3 arrived at 4 ms.
P4 arrived at 6 ms.
P5 arrived at 8 ms.



Average Turn Around Time

Formula : Turn around Time =: waiting time + burst time
Turn Around Time for P1 => 0+3= 3
Turn Around Time for P2 => 1+6 = 7
Turn Around Time for P3 => 5+4 = 9
Turn Around Time for P4 => 7+ 5= 12
Turn Around Time for P5 => 2+ 10=12
Average Turn Around Time => (3+7+9+12+12)/5 =>43/5 = 8.50 ms.

Average Response Time :

Formula : Response Time = First Response - Arrival Time
Response Time of P1 = 0
Response Time of P2 => 3-2 = 1
Response Time of P3 => 9-4 = 5
Response Time of P4 => 13-6 = 7
Response Time of P5 => 18-8 =10
Average Response Time => (0+1+5+7+10)/5 => 23/5 = 4.6 ms

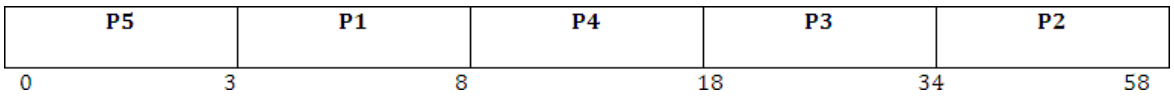
Advantages: Easy to Implement, Simple.

Disadvantage: Average waiting time is very high.

2) Shortest Job First Scheduling (SJF):

Which process having the smallest CPU burst time, CPU is assigned to that process . If two process having the same CPU burst time, FCFS is used.

PROCESS	CPU BURST TIME
P1	5
P2	24
P3	16
P4	10
P5	3



P5 having the least CPU burst time (3ms). CPU assigned to that (P5). After completion of P5 short term scheduler search for nest (P1).....

Average Waiting Time :

Formula = Staring Time - Arrival Time
waiting Time for P1 => 3-0 = 3
waiting Time for P2 => 34-0 = 34
waiting Time for P3 => 18-0 = 18
waiting Time for P4 =>8-0=8
waiting time for P5=0
Average waiting time => (3+34+18+8+0)/5 => 63/5 =12.6 ms

Average Turn Around Time :

Formula = waiting Time + burst Time

Turn Around Time for P1 => 3+5 =8
Turn Around for P2 => 34+24 =58
Turn Around for P3 => 18+16 = 34

Turn Around Time for P4 => 8+10 =18
Turn Around Time for P5 => 0+3 = 3
Average Turn around time => (8+58+34+18+3)/5 => 121/5 = 24.2 ms

Average Response Time :

Formula : First Response - Arrival Time

First Response time for P1 =>3-0 = 3
First Response time for P2 => 34-0 = 34
First Response time for P3 => 18-0 = 18
First Response time for P4 => 8-0 = 8
First Response time for P5 = 0
Average Response Time => (3+34+18+8+0)/5 => 63/5 = 12.6 ms
SJF is Non primitive scheduling algorithm

Advantages : Least average waiting time Least average turn around time Least average response time

Average waiting time (FCFS) = 25 ms
Average waiting time (SJF) = 12.6 ms 50% time saved in SJF.

Disadvantages:

- Knowing the length of the next CPU burst time is difficult.
- Aging (Big Jobs are waiting for long time for CPU)

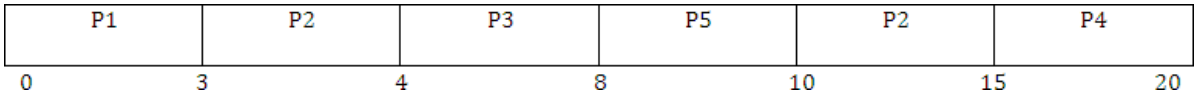
3) Shortest Remaining Time First (SRTF):

This is primitive scheduling algorithm.

Short term scheduler always chooses the process that has term shortest remaining time. When a new process joins the ready queue , short term scheduler compare the remaining time of executing process and new process. If the new process has the least CPU burst time, The scheduler selects that job and connect to CPU. Otherwise continue the old process.

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0

P2	6	2
P3	4	4
P4	5	6
P5	2	8



P1 arrives at time 0, P1 executing First , P2 arrives at time 2. Compare P1 remaining time and P2 (3-2 = 1) and 6. So, continue P1 after P1, executing P2, at time 4, P3 arrives, compare P2 remaining time (6-1=5) and 4 (4<5) .So, executing P3 at time 6, P4 arrives. Compare P3 remaining time and P4 (4-2=2) and 5 (2<5). So, continue P3 , after P3, ready queue consisting P5 is the least out of three. So execute P5, next P2, P4.

FORMULA : Finish time - Arrival

Time Finish Time for P1 => 3-0 = 3

Finish Time for P2 => 15-2 = 13

Finish Time for P3 => 8-4 =4

Finish Time for P4 => 20-6 = 14

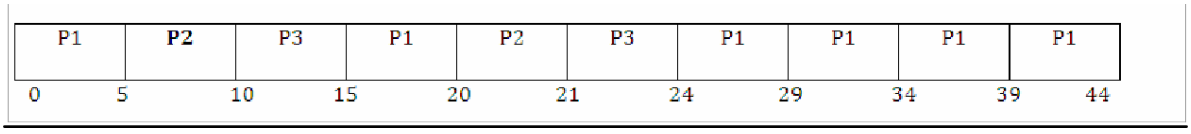
Finish Time for P5 => 10-8 = 2

Average Turn around time => 36/5 = 7.2 ms.

4)ROUND ROBIN SCHEDULING ALGORITHM :

It is designed especially for time sharing systems. Here CPU switches between the processes. When the time quantum expired, the CPU switched to another job. A small unit of time, called a time quantum or time slice. A time quantum is generally from 10 to 100 ms. The time quantum is generally depending on OS. Here ready queue is a circular queue. CPU scheduler picks the first process from ready queue, sets timer to interrupt after one time quantum and dispatches the process.

PROCESS	BURST TIME
P1	30
P2	6
P3	8



AVERAGE WAITING TIME :

Waiting time for P1 => 0+(15-5)+(24-20) => 0+10+4 = 14
Waiting time for P2 => 5+(20-10) => 5+10 = 15
Waiting time for P3 => 10+(21-15) => 10+6 = 16
Average waiting time => (14+15+16)/3 = 15 ms.

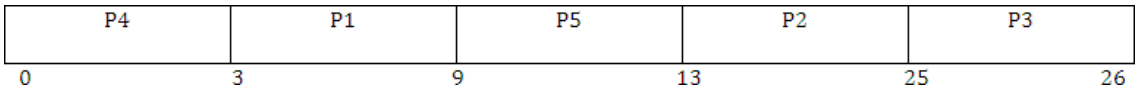
AVERAGE TURN AROUND TIME :

FORMULA : Turn around time = waiting time + burst Time
Turn around time for P1 => 14+30 =44
Turn around time for P2 => 15+6 = 21
Turn around time for P3 => 16+8 = 24
Average turn around time => (44+21+24)/3 = 29.66 ms

5) PRIORITY SCHEDULING :

PROCESS	BURST TIME	PRIORITY
P1	6	2
P2	12	4
P3	1	5
P4	3	1
P5	4	3

P4 has the highest priority. Allocate the CPU to process P4 first next P1, P5, P2, P3.



AVERAGE WAITING TIME :

Waiting time for P1 => 3-0 =3 Waiting time for P2 => 13-0 = 13 Waiting time for P3 => 25-0 = 25 Waiting time for P4 => 0
Waiting time for P5 => 9-0 =9

Average waiting time => (3+13+25+0+9)/5 = 10 ms

AVERAGE TURN AROUND TIME :

Turn around time for P1 =>3+6 = 9
Turn around time for P2 => 13+12= 25
Turn around time for P3 => 25+1 = 26
Turn around time for P4 => 0+3= 3
Turn around time for P5 => 9+4 = 13

Average Turn around time => (9+25+26+3+13)/5 = 15.2 ms

Disadvantage: Starvation

Starvation means only high priority process are executing, but low priority process are waiting for the CPU for the longest period of the time.

Multiple – processor scheduling:

When multiple processes are available, then the scheduling gets more complicated, because there is more than one CPU which must be kept busy and in effective use at all times.
Load sharing resolves around balancing the load between multiple processors. Multi processor systems may be heterogeneous (It contains different kinds of CPU's) (or) Homogeneous(all the same kind of CPU).

1) Approaches to multiple-processor scheduling a)Asymmetric multiprocessing

One processor is the master, controlling all activities and running all kernel code, while the other runs only user code.

b)Symmetric multiprocessing:

Each processor schedules its own job. Each processor may have its own private queue of ready processes.

2) Processor Affinity

Successive memory accesses by the process are often satisfied in cache memory. what happens if the process migrates to another processor. the contents of cache memory must be invalidated for the first processor, cache for the second processor must be repopulated. Most Symmetric multi processor systems try to avoid migration of processes from one processor to another processor, keep a process running on the same processor. This is called processor affinity.

a) Soft affinity:

Soft affinity occurs when the system attempts to keep processes on the same processor but makes no guarantees.

b) Hard affinity:

Process specifies that it is not to be moved between processors.

3) Load balancing:

One processor wont be sitting idle while another is overloaded.
Balancing can be achived through push migration or pull migration.

Push migration:

Push migration involves a separate process that runs periodically(e.g every 200 ms) and moves processes from heavily loaded processors onto less loaded processors.

Pull migration:

Pull migration involves idle processors taking processes from the ready queues of the other processors.

Real time scheduling:

Real time scheduling is generally used in the case of multimedia operating systems. Here multiple processes compete for the CPU. How to schedule processes A,B,C so that each one meets its deadlines. The general tendency is to make them pre-emptable, so that a process in danger of missing its deadline can preempt another process. When this process sends its frame, the preempted process can continue from where it had left off. Here throughput is not so significant. Important is that tasks start and end as per their deadlines.

RATE MONOTONIC (RM) SCHEDULING ALGORITHM

Rate monotonic scheduling Algorithm works on the principle of preemption. Preemption occurs on a given processor when higher priority task blocked lower priority task from execution. This blocking occurs due to priority level of different tasks in a given task set. rate monotonic is a preemptive algorithm which means if a task with shorter period comes during execution it will gain a higher priority and can block or preemptive currently running tasks. In RM priorities are assigned according to time period. Priority of a task is inversely proportional to its timer period. Task with lowest time period has highest priority and the task with highest period will have lowest priority.

For example, we have a task set that consists of three tasks as follows

Tasks	Execution time(Ci)	Time period(Ti)
T1	0.5	3
T2	1	4
T3	2	6

Table 1. Task set

$$\mathbf{U} = 0.5/3 + 1/4 + 2/6 = 0.167 + 0.25 + 0.333 = 0.75$$

As processor utilization is less than 1 or 100% so task set is schedulable and it also satisfies the above equation of rate monotonic scheduling algorithm.

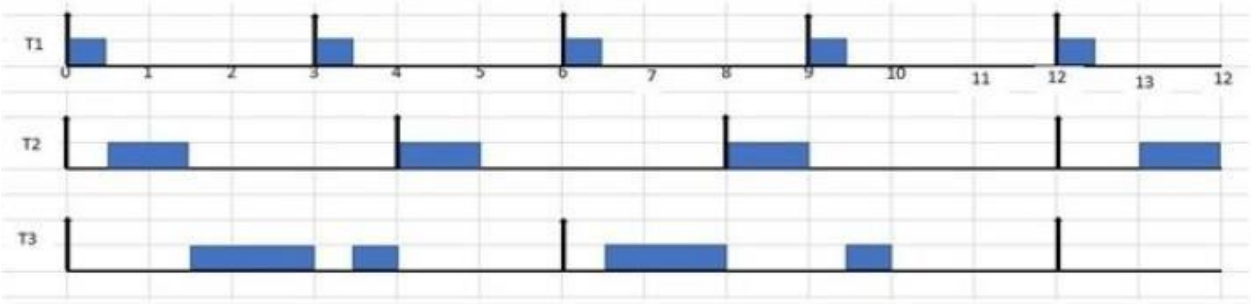


Figure 1. RM scheduling of Task set in table 1.

A task set given in table 1 it RM scheduling is given in figure 1. The explanation of above is as follows

1. According to RM scheduling algorithm task with shorter period has higher priority so T1 has high priority, T2 has intermediate priority and T3 has lowest priority. At t=0 all the tasks are released. Now T1 has highest priority so it executes first till t=0.5.
2. At t=0.5 task T2 has higher priority than T3 so it executes first for one-time units till t=1.5. After its completion only one task is remained in the system that is T3, so it starts its execution and executes till t=3.
3. At t=3 T1 releases, as it has higher priority than T3 so it preempts or blocks T3 and starts it execution till t=3.5. After that the remaining part of T3 executes.
4. At t=4 T2 releases and completes it execution as there is no task running in the system at this time.
5. At t=6 both T1 and T3 are released at the same time but T1 has higher priority due to shorter period so it preempts T3 and executes till t=6.5, after that T3 starts running and executes till t=8.
6. At t=8 T2 with higher priority than T3 releases so it preempts T3 and starts its execution.
7. At t=9 T1 is released again and it preempts T3 and executes first and at t=9.5 T3 executes its remaining part. Similarly, the execution goes on.

Earliest Deadline First (EDF) Scheduler Algorithm

The EDF is a dynamic algorithm, Job priorities are re-evaluated at every decision point, this re-evaluation is based on relative deadline of a job or task, the closer to the deadline, the higher the priority.

The EDF has the following advantages:

1. Very flexible (arrival times and deadlines do not need to be known before implementation).
2. Moderate complexity.
3. Able to handle aperiodic jobs.

The EDF has the following disadvantages:

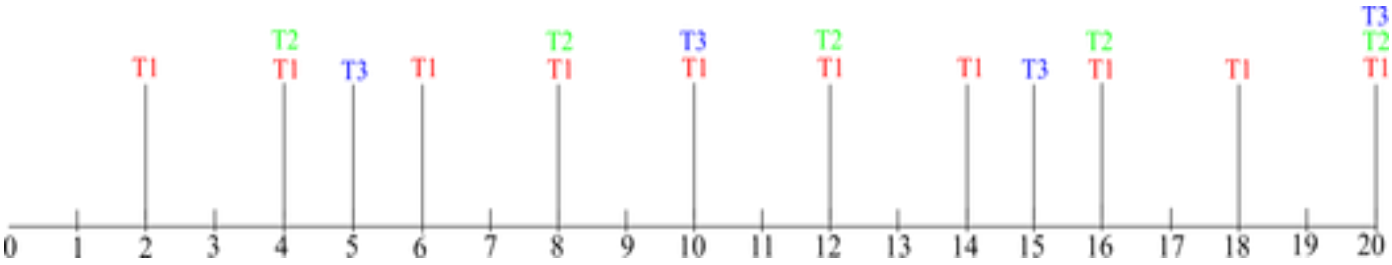
1. Optimally requires pre-emptive jobs.
2. Not optimal on several processors.
3. Difficult to verify.

Example

Consider the following task set in Table 1. P represents the Period, e the Execution time and D stands for the Deadline. Assume that the job priorities are re-evaluated at the release and deadline of a job.

	P	**e**	**D**
T1	2	0.5	2
T2	4	1	4
T3	5	1.5	5

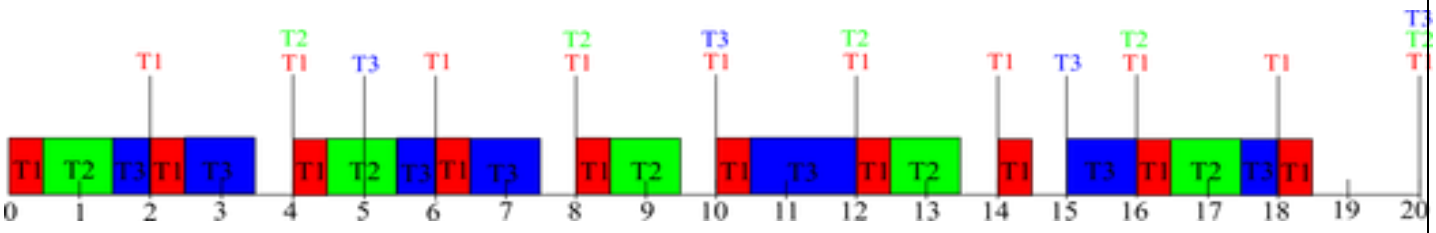
Solution



Mark all deadlines related to all the tasks

- First mark all deadlines related to the tasks as shown in Fig. 1. T1, T2 and T3 are represented with Red, Green and Blue colour respectively. The schedule is from 0 – 20ms as shown.
- At T = 0, T1 has the closest deadline, so schedule T1.
- At T = 0.5, T1 is completed, its next release time is at 2ms. T2 is closer to its deadline so T2 is scheduled next and executes for 1s.
- At T = 1.5, T2 job is completed. T3 is next because it is closer to its deadline while T2 has not been released.
- At T = 2, a new instance of T1 is released, therefore, T3 is interrupted and has 1ms left to complete execution. T1 executes
- At T = 2.5, The only ready job is T3 which is scheduled until completion.
- At T = 4, a new instance of T1 is released which executes for 0.5ms.
- At T = 4.5, T1 is now completed, so T2 is now the task closest to its deadline and is scheduled.
- At T = 5.5, T3 is scheduled but is pre-empted at T = 6 so runs for 0.5ms
- At T = 6, a new instance of T1 is released and therefore scheduled.
- At T = 6.5, T3 is closest to its deadline because T1 and T3 have not been released. So T3 is allowed to complete its execution which is 1ms.
- At T = 8, a new instance of T1 is released and is scheduled.
- At T = 8.5, T2 is the task having the closest deadline and so is scheduled to run for its execution time.
- At T = 10, the next release of T1 is scheduled.

- At T = 10.5, the next job with the closest deadline is T3 because the next T2 job will be released at T = 12. So T3 is scheduled until completion.
- At T = 12, the next release of T1 is scheduled.
- At T = 12.5, T2 is scheduled as it is the job with the closest deadline.
- At T = 14, the next release of T1 is scheduled.
- At T = 15, the next release of T3 is scheduled because it is now the job with the closest deadline because the next release of T1 and T2 is at 16ms. T3 runs for 1ms.
- At T = 16, T3 is pre=empted because a new release of T1 which has the closest deadline is now available.
- T = 16.5, T2 is the job with the closest deadline, so it is scheduled for the duration of its execution time.
- At T = 17.5, since T1 and T2 have completed, T3 resumes execution to complete its task which ran for only 1ms the last time. T3 completes execution at T = 18.
- At T = 18, a new instance of T1 is released and scheduled to run for its entire execution time.
- At T = 18.5, no job is released yet because a new release of T1, T2 and T3 are at 20ms.
- Fig. 2 shows the EDF schedule from T = 0 to T = 20.
- .



Inter Process communication:

Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. Coordination of simultaneous processes to complete a task is known as process synchronization.

The critical section problem

Consider a system , assume that it consisting of n processes. Each process having a segment of code. This segment of code is said to be critical section.
E.G: Railway Reservation System.
Two persons from different stations want to reserve their tickets, the train number, destination is common, the two persons try to get the reservation at the same time. Unfortunately, the available berths are only one; both are trying for that berth.

It is also called the critical section problem. Solution is when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section.**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

Figure General structure of a typical process P_i.

A solution to the critical section problem must satisfy the following 3 requirements:

1.mutual exclusion:

Only one process can execute their critical section at any time.

2. Progress:

When no process is executing a critical section for a data, one of the processes wishing to enter a critical section for data will be granted entry.

3. Bounded wait:

No process should wait for a resource for infinite amount of time.

Critical section:

The portion in any program that accesses a shared resource is called as critical section (or) critical region.

Peterson's solution:

Peterson solution is one of the solutions to critical section problem involving two processes. This solution states that when one process is executing its critical section then the other process executes the rest of the code and vice versa.

Peterson solution requires two shared data items:

1) **turn**: indicates whose turn it is to enter into the critical section. If turn == i ,then process i is allowed into their critical section.

2) **flag:** indicates when a process wants to enter into critical section. when

process i wants to entertheir critical section,it sets flag[i] to true.

do {flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
critical section

flag[i] = FALSE; remainder
section

} while (TRUE);

Synchronization hardware

In a uniprocessor multiprogrammed system, mutual exclusion can be obtained by disabling the interrupts before the process enters its critical section and enabling them after it has exited the critical section**.**

Disable
interrupts
Critical section
Enable interrupts

Once a process is in critical section it cannot be interrupted. This solution cannot be used in multiprocessor environment. since processes run independently on different processors.

In multiprocessor systems, **Testandset** instruction is provided,it completes execution without interruption. Each process when entering their critical section must set **lock**,to prevent other processes from entering their critical sections simultaneously and must release the lock when exiting their critical sections.

do { acquire
lock critical section
release lock
remainder section

} while (TRUE);

A process wants to enter critical section and value of lock is false then **testandset** returns false and the value of lock becomes true. thus for other processes wanting to enter their critical sections **testandset** returns true and the processes do busy waiting until the process exits critical section and sets the value of lock to false.

- **Definition:**

```
boolean TestAndSet(boolean&lock){
boolean temp=lock;
Lock=true;
return temp;
}
```

Algorithm for TestAndSet

```
do{
        while testandset(&lock)
        //do nothing
        //critical section
        lock=false
remainder section
}while(TRUE);
```

Swap instruction can also be used for mutual exclusion

Definition

```
Void swap(boolean &a, boolean &b)
{
boolean temp=a;
a=b;
b=temp;
}
```

Algorithm

```
do
{
key=true;
while(key=true)
```

```
swap(lock,key);
critical section
lock=false;
remainder section
}while(1);
```

lock is global variable initialized to false.each process has a local variable key. A process wants to enter critical section,since the value of lock is false and key is true.

lock=false
key=true

after swap instruction,

lock=true
key=false

now key=false becomes true,process exits repeat-until,and enter into critical section. When process is in critical section (lock=true),so other processes wanting to enter critical section will have

lock=true
key=true

Hence they will do busy waiting in repeat-until loop until the process exits critical section and sets the value of lock to false.

Semaphores

A semaphore is an integer variable.semaphore accesses only through two operations.

1) **wait:** wait operation decrements the count by1.

If the result value is negative,the process executing the wait operation is blocked.

2) **signaloperation:**

Signal operation increments by 1,if the value is not positive then one of the process blocked in wait operation unblocked.

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

```
}

signal (S)
{

S++;
}
```

In binary semaphore count can be 0 or 1. The value of semaphore is initialized to 1.

```
do {
wait (mutex);
// Critical Section
signal (mutex);
// remainder section
} while (TRUE);
```

First process that executes wait operation will be immediately granted sem.count to 0. If some other process wants critical section and executes wait() then it is blocked,since value becomes -1. If the process exits critical section it executes signal().sem.count is incremented by 1.blocked process is removed from queue and added to ready queue.

Problems:

1) **Deadlock**

Deadlock occurs when multiple processes are blocked.each waiting for a resource that can only be freed by one of the other blocked processes.

2) **Starvation**

one or more processes gets blocked forever and never get a chance to take their turn in the critical section.

3) **Priority inversion**

If low priority process is running ,medium priority processes are waiting for low priority process,high priority processes are waiting for medium priority processes.this is called Priority inversion.

The two most common kinds of semaphores are **counting semaphores** and **binary semaphores**. Counting semaphores represent multiple resources, while binary semaphores, as the name implies, represents two possible states (generally 0 or 1; locked or unlocked).

Classic problems of synchronization

1) **Bounded-buffer problem**

Two processes share a common ,fixed –size buffer.

Producer puts information into the buffer, consumer takes it out.

The problem arise when the producer wants to put a new item in the buffer,but it is already full. The solution is for the producer has to wait until the consumer has consumed atleast one buffer. similarly if the consumer wants to remove an item from the buffer and sees that the buffer is empty,it goes to sleep until the producer puts something in the buffer and wakes it up.

synchronisation problems:

i) we must guard against attempting to write data to the buffer when the buffer is full; ie the producer must wait for an 'empty space'.

ii) we must prevent the consumer from attempting to read data when the buffer is empty; ie, the consumer must wait for 'data available'.

To provide for each of these conditions, we require to employ three semaphores which are defined in the following table:

Semaphore	*Purpose*	*Initial Value*
free	mutual exclusion for buffer access	1
space	space available in buffer	N
data	data available in buffer	0

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);
```

2) The readers-writers problem

A database is to be shared among several concurrent processes.some processes may want only to read the database,some may want to update the database.If two readers access the shared data simultaneously no problem.if a write,some other process access the database simultaneously problem arised.Writes have excusive access to

the shared database while writing to the database.This problem is known as readers- writes problem.

First readers-writers problem

No reader be kept waiting unless a writer has already obtained permission to use the shared resource.

Second readers-writes problem:

Once writer is ready,that writer performs its write as soon as possible.
A process wishing to modify the shared data must request the lock in write mode. multiple processes are permitted to concurrently acquire a reader-writer lock in read mode. A reader writer lock in read mode. but only one process may acquirethe lock for writing as exclusive access is required for writers.

Semaphore mutex initialized to 1

- Semaphore wrt initialized to 1
- Integer read count initialized to 0

The structure of a writer process

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

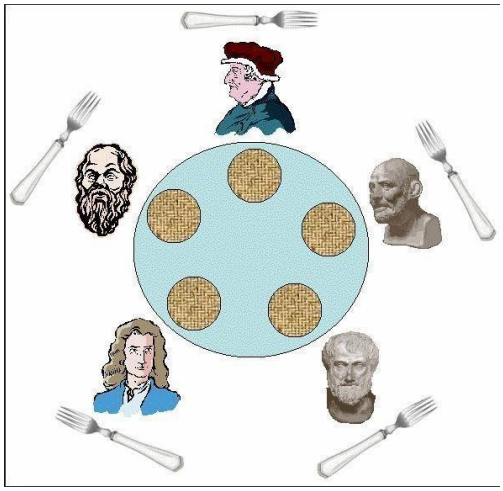
The structure of a reader process

```
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);
```

3) Dining Philosophers problem

Five philosophers are seated on 5 chairs across a table. Each philosopher has a plate full of noodles. Each philosopher needs a pair of forks to eat it. There are only5 forks available all together. There is only one fork between any two plates of noodles.
In order to eat, a philosopher lifts two forks, one to his left and the other to his right. if he is successful in obtaining two forks, he starts eating after some time, he stops eating and keeps both the forks down.



What if all the 5 philosophers decide to eat at the same time ?
All the 5 philosophers would attempt to pick up two forks at the same time. So,none of them succeed.

One simple solution is to represent each fork with a semaphore.a philosopher tries to grab a fork by executing wait() operation on that semaphore.he releases his forks by executing the signal() operation.This solution guarantees that no two neighbours are eating simultaneously.
Suppose all 5 philosophers become hungry simultaneously and each grabs his left fork,he will be delayed forever.

The structure of Philosopher *i*:

```
do{
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5] );
// eat
signal ( chopstick[i] );
signal (chopstick[ (i + 1) % 5] );
// think
} while (TRUE);
```

Several remedies:

1) Allow at most 4 philosophers to be sitting simultaneously at the table.
2) Allow a philosopher to pickup his fork only if both forks are available.
3) An odd philosopher picks up first his left fork and then right fork. an even philosopher picks up his right fork and then his left fork.

MONITORS

The disadvantage of semaphore is that it is unstructured construct. Wait and signal operations can be scattered in a program and hence debugging becomes difficult.

A monitor is an object that contains both the data and procedures needed to perform allocation of a shared resource. To accomplish resource allocation using monitors, a process must call a **monitor entry routine**. Many processes may want to enter the monitor at the same time. but only one process at a time is allowed to enter. Data inside a monitor may be either global to all routines within the monitor (or) local to a specific routine. Monitor data is accessible only within the monitor. There is no way for processes outside the monitor to access monitor data. This is a form of information hiding.

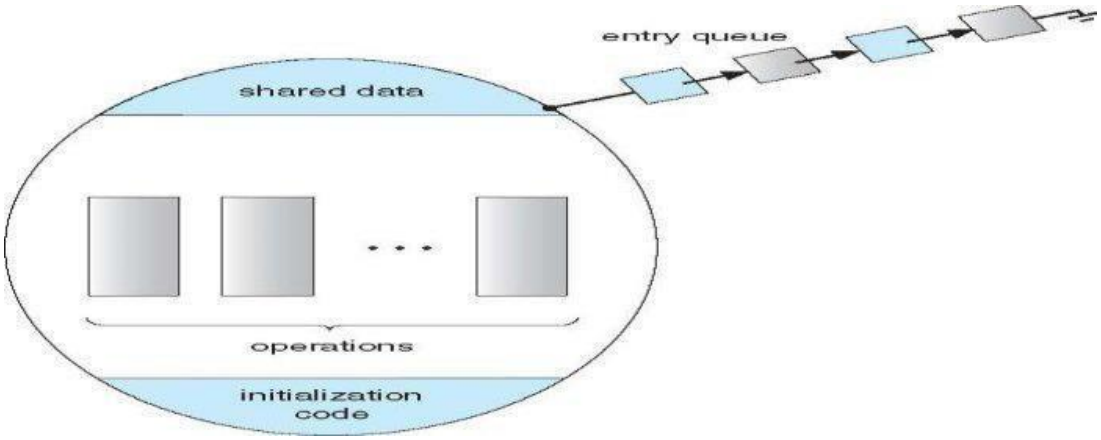
If a process calls a monitor entry routine while no other processes are executing inside the monitor, the process acquires a lock on the monitor and enters it. while a process is in the monitor, other processes may not enter the monitor to acquire the resource. If a process calls a monitor entry routine while the other monitor is locked the monitor makes the calling process wait outside the monitor until the lock on the monitor is released. The process that has the resource will call a monitor entry routine to release the resource. This routine could free the resource and wait for another requesting process to arrive monitor entry routine calls signal to allow one of the waiting processes to enter the monitor and acquire the resource. Monitor gives high priority to waiting processes than to newly arriving ones.

Structure:

```
monitor monitor-name
{
// shared variable declarations
procedure P1 (…) { …. }
procedurePn (…) {……}
Initialization code (…) { … }
}
```

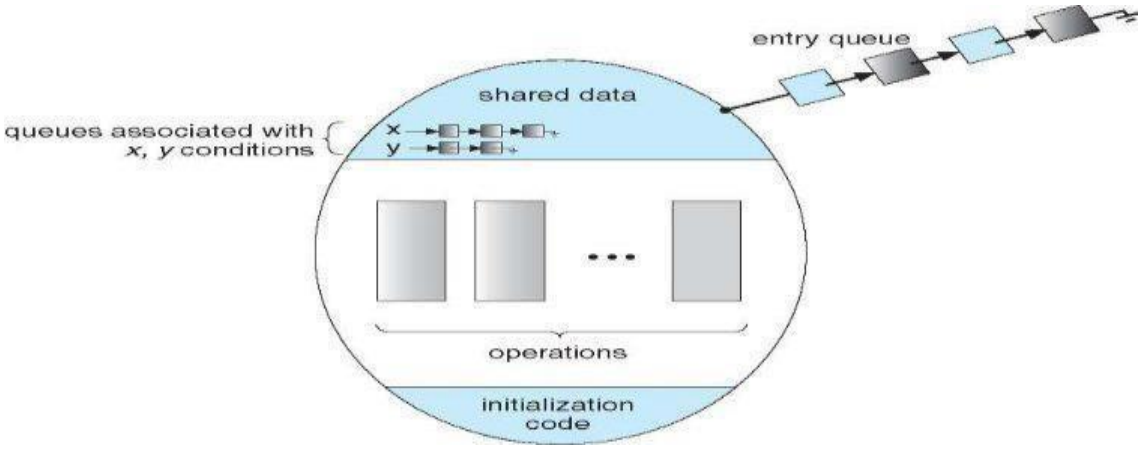
}

Processes can call procedures p1,p2,p3……They cannot access the local variables of the monitor



Schematic view of a Monitor

Monitor with Condition Variables



Monitor provides condition variables along with two operations on them i.e. wait and signal.

wait(condition variable)
signal(condition variable)

Every condition variable has an associated queue.A process calling wait on a particular condition variable is placed into the queue associated with that condition variable.A process calling signal on a particular condition variable causes a process waiting on that condition variable to be removed from the queue associated with it.

Solution to Producer consumer problem using monitors:

```
monitor
producerconsumer
condition
full,empty;

int count;

procedure insert(item)

{

if(count==MAX)
wait(full) ;
insert_item(item);
count=count+1;
if(count==1)
signal(empty);

}

procedure remove()

{

if(count==0)
wait(empty);
remove_item(item);
count=count-1;
if(count==MAX-1)
signal(full);

}

procedure producer()

{

producerconsumer.insert(item);
```

}

procedure consumer()

{

producerconsumer.remove();

}

Solution to dining philosophers problem using monitors

```
        void test(int i) {
           if ((state[(i + 4) % 5] != eating) &&
              (state[i] == hungry) &&
              (state[(i + 1) % 5] != eating)) {
                 state[i] = eating;
                 self[i].signal();
           }
        }

        void init() {
           for (int i = 0; i < 5; i++)
              state[i] = thinking;
        }
     }
```

Figure A monitor solution to the dining-philosopher problem.

```
        test((i + 1) % 5);
     }
```

A philosopher may pickup his forks only if both of them are available.A philosopher can eat only if his two neighbours are not eating.some other philosopher can delay himself when he is hungry.

Diningphilosophers.Take_forks( ) : acquires forks ,which may block the process.

Eat noodles ( )

Diningphilosophers.put_forks( ): releases the forks.

Resuming processes within a monitor

If several processes are suspended on condion x and x.signal( ) is executed by some process. then

how do we determine which of the suspended processes should be resumed next ?

solution is FCFS(process that has been waiting the longest is resumed first).In many circumstances, such simple technique is not adequate. alternate solution is to assign priorities and wake up the process with the highest priority.

Resource allocation using monitor boolean inuse=false;

conditionavailable;

//conditionvariable

```
monitorentry void get resource()
        {
if(inuse)                       //is resource inuse
        {
wait(available);                wait until available
issignaled
        }
inuse=true;                     //indicate resource is now inuse
        }
monitor entry void return resource()
        {
inuse=false;                    //indicate resource is not in use
signal(available);
//signal a waiting
process to proceed
        }
```