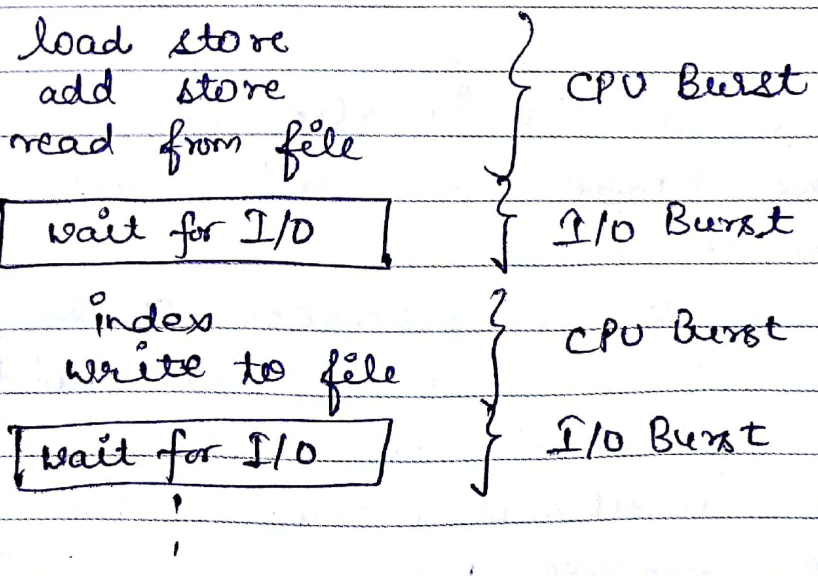
UNIT-II

Process scheduling- Several processes are kept in m/m at one time. When one process has to wait, the O.S takes the CPU away from that process and gives the CPU to another process.

Scheduling is a fundamental O.S function.

CPU - I/O Burst Cycle - Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate b/w these two states.

Process execution begins with a CPU burst, that is followed by an I/O burst, then other CPU burst then other I/O burst and so on.



First any Program → Load to m/m → Allocated to processor or CPU

CPU Scheduler – They are special system s/w which handle process scheduling in various ways.

Three types –

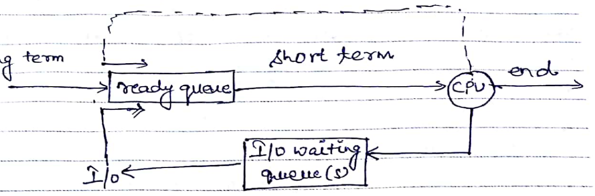
① – Long Term Schedulers
② – Short " "
③ – Medium " "

① – Long – It is also called a job Scheduler. It determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into m/m. for execution.

② – Short – It is also Called CPU scheduler. It is the change of ready state to running state of the process.
It selects a process among the processes that are ready to executes and allocates CPU to one of them. Short term Schedulers, also known as dispatchers, make decision of which process to execute next.

Short → Long. (It is also called
(faster) (slower) dispatcher)

Medium Term → It is a part of swapping. It removes the processes from the m/m.



Scheduling Criteria → Many criteria have been suggested for comparing CPU scheduling algorithms.

→ CPU Utilization – We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In real time it varies from 40 to 90 %.

→ Throughput – If the CPU is busy executing processes, then work is being done. One measure of work is the no. of processes completed per time unit, called throughput.

→ **Turnaround Time** – The point of view, the imp criteria is how long it takes to execute that process.

The interval from the time of submission of a process to the time of completion is the turnaround time.

→ **Waiting Time** – It is the amount of time that a process spends waiting in the ready queue.

→ **Response Time** – Often a process can produce some O/P fairly early, and can continue computing new results while previous results are being O/P to the user.

Thus another measure is the time from the submission of a request until the first response is produced.

Scheduling Algorithm –

Preemptive and non-preemptive algorithms –

In non-preeptive scheduling, a running task is executed till completion. It cannot be interrupted. eg. FCFS.

In preemptive scheduling, a running task is interrupted for some time and resumed later when the priority task has finished its executio This is called preemptive. ex. Round Robin

Static and Dynamic Priority →

Static is the base priority, the one given to the process by the system when the process is created. A higher number gives lower priority when processes fight for CPU time.

Dynamic – It is set by kernel itself. Whenever a process blocks or it has to wait for another process, the dynamic priority is raised. A process that waits a lot therefore gets higher priority than a process that uses a lot of CPU time.

Co-operative and ~~non-cooperative~~ Independent process-

A process is independent if it cannot affect other process or be affected by it. Any process that does not share data with others is independent.

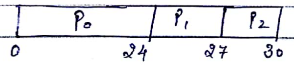
Otherwise the process is co-operating. Co-operation is done to provide infⁿ sharing, speedups, convenience. To allow co-operation there should be some mechanism for communication. (IPC) and synchronize their actions. like- producer-consumer problems.

SCHEDULING ALGORITHMS

① - FCFS - (First Come First Serve)-

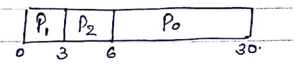
	Processes	Burst time
ex-	P_0	24
	P_1	3
	P_2	3

If they are in FCFS -



waiting time for process $P_0 \Rightarrow 0$, $P_1 = 24$, $P_2 = 27$.
avg. ⇒ 17 milliseconds.

If process arrives in order P_1, P_2, P_0 then -



avg. waiting time ⇒ $(6+0+3)/3 = 3$ millisec.

Convoy Effect in FCFS -

Convoy effect is slowing down of the whole operating system becz of few slow processes.

If multiple processes are waiting for the CPU time for execution in "FCFS" method. And a slow process is utilizing the CPU keeping the fast process on wait. It will lead to the convoy effect. Unnecessary wait will be done by the fast processes.

Let's have an example:-

- Consider 100 I/O bound processes and 1 CPU bound job in system.

- These I/O bound processes will quickly pass through the ready queue and suspend themselves (will wait for I/O).

- Now the CPU-bound process (slow) will get and hold the CPU.

- During this time, all the Other I/O bound processes will finish their I/O and will move into the ready queue, waiting for CPU.

- While the I/O bound processes wait in the ready queue, the I/O devices are idle.

- Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.

- All the I/O bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
- At this point, the CPU sits idle.

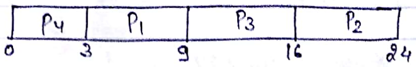
- The CPU-bound process will then move back to the ready queue and be allocated the CPU.

- Again all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.

- There is a convoy effect as all the other processes wait for the one big process to get off the CPU.

② - SJF (Shortest Job First) → When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes has same length next CPU burst, then FCFS is used to break a tie.

Processes	Burst time
P_1	6
P_2	8
P_3	7
P_4	3



waiting time ⇒ $P_1 = 3$, $P_2 = 16$, $P_3 = 9$, $P_4 = 0$
avg. = 7 milliseconds.

$\tau_0 = 10$, $\alpha = 0.5$, 10 $= 0.5 \times 10 + (1-0.5) * \tau_0$
$= 5 + 5 = 10$

$0.5 \times 6 + (0.5)10$ $\tau_0 = 10$, $\alpha = 0.5$ $0.5 \times 4 + 0.5 \times 8$
$3 + 5 = 8$ $2 + 4 = 6$

It is more optimal than FCFS it gives less avg. waiting time. SJF happens in long term scheduling. We can only predict that approximate next CPU burst, will be similar in length to the previous one.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the n^{th} CPU burst, and τ_{n+1} be our predicted value for the next CPU burst. Then

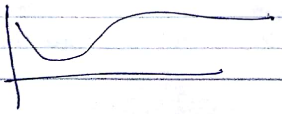
$$\boxed{\tau_{n+1} = \alpha \underline{t_n} + (1-\alpha)\underline{\tau_n}}$$

t_n = Contains our most recent info?

where $0 \leq \alpha \leq 1$. This is called exponential average T_n stores the past history length. α is the relative weight of recent and past history in our prediction.

If $\alpha = 0$ then $T_{n+1} = T_n$.
$\alpha = 1$ then $T_{n+1} = t_n$.

$\underline{\text{ex}}$- $\alpha = 0.5$, $\tau_0 = 10$, $t_1 = 6$. $\Rightarrow$?

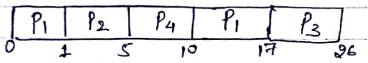


$27/4$

SJF can be preemptive or non-preemptive.
A preemptive SJF algo will preempt the currently executing process, whereas a nonpreemptive SJF will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called "shortest remaining-time-first" scheduling.

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



$P_1 = 10\ (10-0) = 10$
$P_2 = 0\ (1-1) = 0$
$P_3 = 15\ (17-2) = 15$
$P_4 = 2\ (5-3) = 2$

$\text{avg} = \dfrac{10+0+15+2}{4}$
$= 27/4 = 6.5$ milli sec.

A non preemptive SJF = 7.75 millisec.

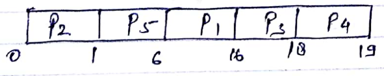
Priority Scheduling :- SJP is a special case of the general priority-scheduling algo.

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

CPU burst ↑ priority ↓

Priority is generally some fixed range of numbers such as 0 to 7 or 0 to 4095.

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



avg. waiting time = 8.2 millisec.

It may be preemptive and non-preemptive

A major problem with priority algo is indefinite blocking (or starvation).

A solution to the problem of starvation of low-priority processes is "Aging".

Aging :- It is a technique of gradually increasing the priority of processes that wait in the system for a long time.

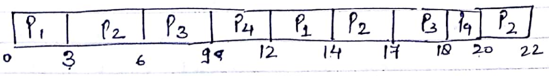
suppose (50) low to 0 (high), we could decreament the priority of a waiting process by 1 at every 5 minutes. After (4 hours 10 min.) it will be able to execute.

Round-Robin Scheduling :- RR scheduling is designed especially for time-sharing environment. A small unit of time called a time-quantum (or time slice), is defined. Range from 1 to 100 milliseconds. The Ready queue will be treaded as FIFO (queue).

$$\left[\begin{array}{l} TAT = C.T - A.T \\ WT = TAT - B.T \end{array}\right]$$

Processes	Burst time
P_1	5
P_2	8
P_3	4
P_4	5

Time q. = 3



Waiting time = S.T − Arrival Time.

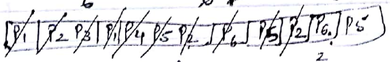
$P_1 = 12 - 3 = 9$
$P_2 = 20 - 6 = 14$
$P_3 = 17 - 3 = 14$
$P_4 = 18 - 3 = 15$

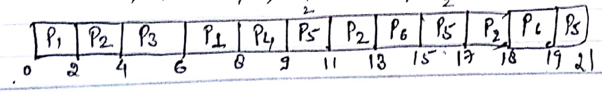
TQ = 2

C.T − A.T

TAT − B.T

P. No.	A.T	B.T	C.T	TAT ~~TQ=2~~	WT
1	0	~~4~~ ~~2~~ 0	8	8	4
2	1	~~5~~ ~~3~~ 1	18	17	12
3	2	~~2~~ 0	6	4	2
4	3	~~1~~ 0	9	6	5
5	4	~~6~~ ~~4~~ ~~2~~ 0	21	17	11
6	6	~~3~~ ~~1~~	19	13	10





$P_5 = 19 - 4 = 15$

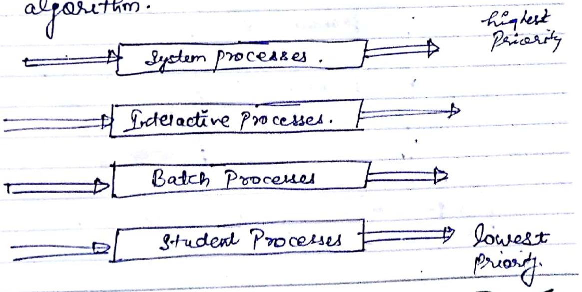
Multilevel Queue Scheduling :- Processes are easily classified into different groups. The two types of processes

→ Foreground Processes (interactive)
→ Background Processes (batch)

F.P processes have more priority than B.P processes.

In this, there are so many partitions of ready queue into several separate queues. The processes are generally assigned to one queue on basis of m/m locations, process size etc.

F.P processes might be scheduled by an RR algo, while the B.P queue ~~to~~ is scheduled by an FCFS algorithm.



Multilevel Feedback Queue Scheduling :-

In MQS, the processes can not move among the queues, They cannot change their nature of Foreground and background.

This system has advantage of low scheduling overhead but disadvantage of being inflexible.

MFQs, allows a process to move b/w queues. The idea is to separate processes with different CPU-burst characterstics.

If a process uses too much CPU time, it will be moved to a lower priority queue, and process which waits for long time will move to upper queue for preventing starvation.

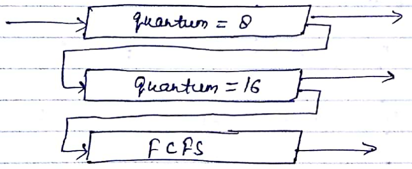


fig :- Multilevel feedback queues .

Inter Process Communication :-

Process may be ~~interactive~~ independat or co-operating processes (Cooking) coop. → study → Indep-

~~Message Passing~~ -

Co-operating process can affect or be affected by other process, including shared data.

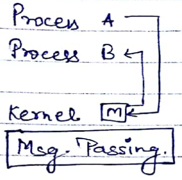
Reason for Copperating -

→ Information Sharing → Salt & pepper
→ Computational Speedup → Two Stoves.
→ Modularity → Including cleaning, frying & combine.
→ Convience → Convienient.

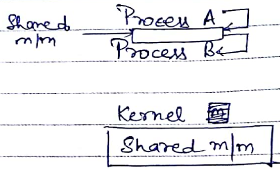
For this ~~[illegible]~~ techniques is (IPC) -
→ Two models of IPC
- Shared m/m
- message passing

Shared m/m → A notice board is used by two employees in BPO's and they leave msg for each other and when second process comes it gets the msg written by first process.

Msg. Passing → Bob → Shally
Sender Reciever.



Msg. Passing.



Shared m/m

Message Passing — (Logical Concepts)

Direct Commⁿ — Processes must name each other explicitly :

→ Send (P, message) - Send a msg to process P.

→ Recieve (Q, message) - recieve a msg. from process Q.

Properties —

- * A link is associated with exactly one pair of communicating processes.
- * B/w each pair there exists exactly one link.
- * The link may be unidirectional or bi-directional.

Indirect Commⁿ — Messages are directed and recieved mailboxes (ports). (Buffer).

- * Each mailbox has a unique id.
- * Processes can communicate only if they share a mailbox.

Properties — * A link may be associated with many processes.

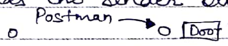
☆ Link may be uni & bi-directional.

Message Synchronization —

→ Msg. passing may be either blocking or non-blocking.

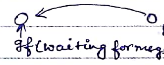
→ Blocking is considered synchronous

- Blocking send has the sender block until the msg is recieved.



(Donot drop until msg will be delivered).

- Blocking recieve has the reciever block until the msg. is available.



(Server) & client

If (waiting for msgs. like gifts eagerly).

→ Non-Blocking is considered Asynchronous

- Non-Blocking send has the sender send the msg. and continue.
- Non-Blocking recieve has the reciever recieve a valid message or null.

Further, in

Process Synchronization → More than one process many processes remain in main m/m. Resources are shared (Printer). Diff. processes can use the printer in mutual exclusion fashion.

One one time if more than two processes use the shared resource at a time then inconsistency occurs.

Simple Example :-

a = 10

```
P ( )
{
   R(a) ← P1
   a = a+1 ↑ P2
   w(a)
}
```

P1 P2

$P_1 = 11$
$P_2 = 12$ } In systematic way.

If we use switching b/w the instructions -

$P_1 = 10$ $P_2 = 11$

& After P_2, P_1 will also write 11. Hence the conditional result is different.

"Race Condition - When order of execution can change result."

If Process operate on his/ its private resources that don't matter but in whole execution if there is shared section like

```
P( )
{
  :
  [C.S] ← S.R
  :
}
```

Hence the area in process execution, where process shared h/w or s/w is known as "Critical Section."

(Ex) - [Gate Questions] -

```
P1( )                    P2( )
{                        {
① C = B-1;               ③ D = 2*B;
② B = 2*C;               ④ B = D-1;
}                        }
```

B is a shared variable with initial value 2. How many diff values of B can have?

a) - 3   b) - 2   c) - 5   d) - 4.

Solⁿ-

Case 1 (order - 1,2,3,4)	Case 2 (3,4,1,2)
C = 2-1 = 1	D = 2*2 = 4
B = 2*1 = 2	B = 4-1 = 3
D = 2*2 = 4	C = 3-1 = 2
B = 4-1 = ③	B = 2*2 = ④

Case-3 (1, 3, 4, 2)
C = 2-1 = 1
D = 2*2 = 4
B = 4-1 = 3
B = 2*1 = ②

Case-4 (3, 1, 2, 4)
D = 2*2 = 4
C = 2-1 = 1
B = 2*1 = 2
B = 4-1 = ③

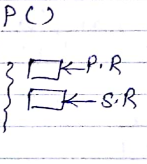
Case-5 (1, 3, 2, 4)
C = 2-1 = 1
D = 2*2 = 4
B = 2*1 = 2
B = 4-1 = ③

Case-6 (3, 1, 4, 2)
D = 2*2 = 4
C = 2-1 = 1
B = 4-1 = 3
B = 2*1 = ②

Hence option 'A' is correct.

Critical Section Problem :-

That part of a code where a process access shared resource.



Problem - If any process access C.S than no inconsistency will occur.

Criteria's should be satisfied by the solutions. Criterias are following -

① - **Mutual Exclusion** - C.S must be accessed only one process at a single unit time. It's a mandatory criteria.

② - **Progress** - Can we execute processes in RR fashion. Yes we can but this is not a good solution beez may be some processes don't want to go in C.S.
So only those process should compete for C.S, those wants to go in C.S. It is also mandatory criteria.

③ - Bounded Wait :- There must be a max. bound upto a process can wait. (Time limit).

After that time limit, the process should be bounded to enter in C.S. But this is not mandatory. (optional).

Two Process Solution :- We will check all the criteria should be fulfilled.

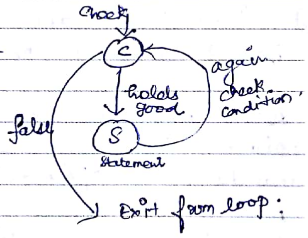
P_0 , P_1 Independant Processes.

```
P0           P0                                   P1
{            while ( 1 )                          while ( 1 )
Entry        {                                    {
section      while ( turn != 0 );  <- trap        while ( turn != 1 );
C.S            critical section <- P0               C.S
exit sec.    turn = 1;                             turn = 0;
Remainder    remainder section                      R.S
section      }                                    }
}
```

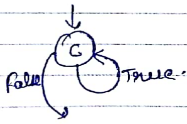
turn is a boolean value either 0 or 1.

In first case turn = 0 , P0 check whether(turn != 0)
But it is 0, the condition is false so come to end of loop. reached to C.S.

P0 while loop works like –



But



Hence P0 is preempt in C.S and give the chance to P_1 but P_1 tries for a no. of attempt. Hence again it gives the charge again to P_0.

P0 will continue and make it turn = 1. Now if P0 wants to go again C.S still it cannot go to C.S , now the turn of P_1 holds good.

Hence at any instance of time only 1 process ~~at~~ will execute.

① – M.E ✓
Strictly alteration → ② – Progress $P_0 \rightarrow P_1$ X
No choice is here ③ – B.W.

This is not a good solution.

Now Case 2 :– We din't ask any process, whether it wants to go in C.S or not. Now we use it – Flag

Flag	F	F
	0	1

If any process wants to go in C.S, Hence the value of flag should be True.

```
P0                              P1
while ( 1 )                     while (1)
{ flag [0] = T                  {
  while ( flag[1] );              flag [1] = T;
    C.S                         while (flag[0]);
  flag [0] = F;                     C.S
}                                 flag [1] = F;
                                }
```

P_0 wants to go in C.S–

	0	1
flag	T	F

It will check P_1 wants to go while (flag [1]); So it is false. Hence P_0 goes to C.S. Now Context switch P_0. Run P_1.

	0	1
flag	T	T

Hence P_1 will remain in loop. and give charge to P_0.

	0	1
flag	F	T

Now P_1 goes to C.S. and context switch P_1 and then P_0 still cannot go to C.S.

So ① mutual Exclusion ✓
② - Progress ✗
③ - B.W.

If P_1 is not interested in C.S.

	0	1
Flag	T	F

P_0 only go to C.S any no. of time. only those processes are competing for C.S.

Now if we contexet switch P_0 at flag [0] = T and give charge to P_1 and P_1 then flag [1] = T.

In this condition, No process will go in C.S. Hence surprisingly, in case of progress system goes to deadlock. So this is not so good strategy.

Peterson's Solution Case 3

```
P0
while (1)
{
   flag [0] = T
    turn = 1
 while (turn == 1 && flag [1] == T);
    C.S
   flag [0] = F
}

P1
while (1)
{
flag [1] = T
turn = = Φ
while (turn !== 0 &&
   flag [0] != T);
 C.S
flag [1] = F
}
```

turn = 0/1

	0	1
flag	F	F

	0	1
	T	F

★ P_0 check & turn = 1 and check while loop and result is False Hence

P_0 comes to C.S.

★ Now P_1 comes check flag & turn

flag $\begin{array}{|c|c|} \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \hline T & T \\ \hline \end{array}$, Turn = 0 and checks while loop and result will be True hence P_1 remain in loop and won't enter into C.S.

So mutual exclusion is there.

★ Now P_0 comes out from C.S and make flag [0] = F then P_1 will goes to C.S.

Now point is Progress ⇒

★ Restart flag $\begin{array}{|c|c|} \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \hline F & F \\ \hline \end{array}$ turn = 0/1

In this if P_0 wants a no. of times to access the C.S, can access while P_1 is not interested.

★ Suppose P_1 wants to go in C.S, it sets flag [1] = T and turn = 0 then context switch and give turn to P_0.

★ Now P_0 check flag [0] = T and turn = 1, then while loop gives true and buzzy wait for C.S.

★ Again charge will goes to P_1 and now turn = 1 and flag [0] = T, Hence P_1 will go in C.S So system will not go to deadlock condition.

Hence Progress ensures in Peterson's Algorithm.

Bounded Wait :- It holds good. If suppose P_0 goes to C.S and while C.S, P_0 context switch and P_1 tries but cannot go to C.S until P_0 finish. then P_0 comes out from critical section and again try but this time P_0 ~~will~~ can not go to C.S. Hence P_1 will take charge & goes to C.S.

So for a single turn you have to wait not so much long time.

For n processes.

Semaphores — A semaphore is an integer variable that apart from initialization, is accessed only through two standards atomic operations.

① - Wait ()

② - Signal ()

Semaphore could be like int S;

Semaphores can also use for other applications like- resource mgmt', order of execution of processes and critical section.

So in critical section, it will always initialize by '1'.

(1) - wait (S) ⇒ S--
(2) - Signal (S) ⇒ S++

```
wait (s)
 {
 while (S<=0);   ← false
  S = S-1;
 }

Signal (s)
 {
   S = S+1;
 }
```

General structure -

Let P0 P1 P2 - - - - Pn.

Simple semaphore

```
        Pi
do
{
   wait (S);
// C.S
   Signal (S);
// remainder section
}
while (T);
```

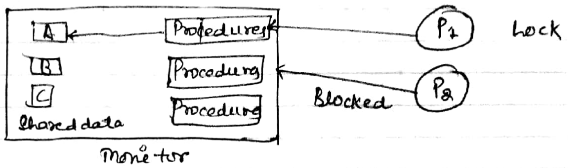
Properly satisfy M.E. ✓

✓ progress (Jisko Andar Jana hai, Cango) [No ordering defined. System does not force anyone)

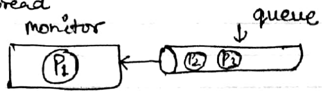
Bounded wait - No B.W, we can ~~say~~ assigned to any process] Still semaphore is a good selection becz third one is optional.

Monitors → A monitor is a module that encapsulates

- Shared data structures
- Procedures that operates on the shared data.
- Synchronization b/w concurrent procedure invocation



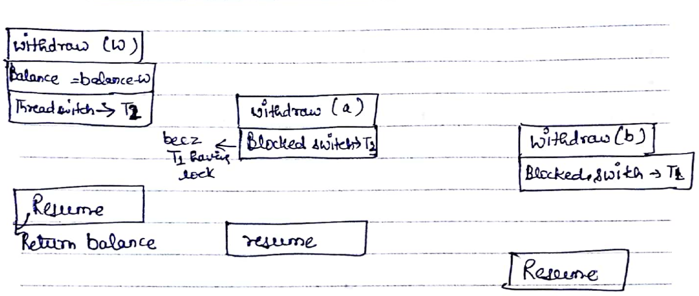
Monitor will ensure that only at one time only one process/thread will enter into monitor.



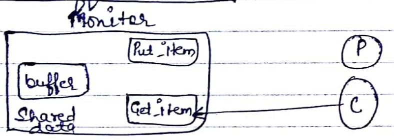
Example - Monitor Account

```
Monitor Account
 {
   Double balance;   → Shared data
   withdraw (amount)
    {
     balance = balance - Amount;
      return balance;
    }
 }
```

T1
T2
T3



Bounded Buffer Problem



Initially buffer is empty & Ⓒ will get monitor and see buffer is empty then starts waiting.

If Ⓟ now produce something, it can't becz monitor won't allow it to enter

So conditional variable comes. It provides synchronization inside the monitor.

- ☆ If a process wants to sleep inside the monitor or it allow a waiting process to continue, in that case conditional variables are used in monitors.
- ☆ Three operations can be performed on conditional variables –
  wait, signal, broadcast

Wait Operation – If resources is/are currently not available, current process put to sleep. It release the lock for monitor.

Signal Operation – Signal operation wakes up one process which is sleeping as a result of call to wait. This causes a waiting process to resume immediately. The lock is automatically passed to the waiter, the original process blocks.

broadcast – Broadcast operation signal to all waiting processes.

[Gate 2013]

Q. A shared variable a, initialized to zero, is operated by four processes W, X, Y, Z. Process W and X increament a by one, while process Y, Z decreament a by two. Each process before reading perform 'wait' on a semaphore 's' and signal on 'S' after store. If semaphore 'S' is initiazed to two. find what is max. possible value of a after all processes complete execution?

a) → −2   b) → −1   c) → 1   d) 2 ✓

W	X	Y	Z
wait (S)	wait(S)	wait (S)	wait (S)
R(a)	R(a)	R(a)	R(a)
a = a+1	a = a+1	a = a−2	a = a−2
W(a)	W(a)	W(a)	W(a)
Signal(S)	Signal(s)	Signal (s)	Signal (s)

S = 2   a = 0

→ W ⇒ S = 1 Read $(a)^{=0}$ and content switch to Z.

→ Z = S = 0 Read (a) = 0, a = −2 and signal S = 1.

$\boxed{S=1}$

→ Y ⇒ S = 0 Read (a) = −2 and a = −4 and S = 1 Now give chance W.

⇒ W ⇒ 0+1 = a = 1 and signal will become S = 2. (Lost update problem)

⇒ X ⇒ S = 1, a = 2 and S = 2.

If we ask for minimum value of a = −4 (Same trick).

But logically value should be −2