OS (Qus)

Date: Page:

Q What is OS? Purpose of OS? Types of OS?

→ OS is a system software which acts as an interface b/w user & H/W
→ resource allocator & manager

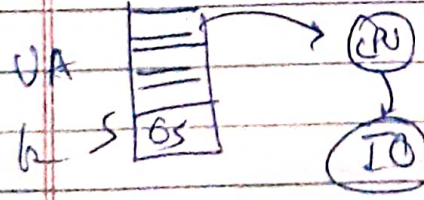
Purpose of OS
- Convinience - User friendly
- Effective utilization of resources
  - CPU scheduling
  - memory management (avoiding fragmentation)

Batch OS
manual feeding of Job cards in main memory

CPU idle
No job schedules unless previous gets completed.

Uniprograming OS
manual X first come first serve

Multiprograming OS



J1 → goes to IO
new Job is Scheduled to CPU

Multiprocessing
multiple CPU's

Multitasking
Multiprograming + preemption

Q. Socket, Kernel, Monolithic Kernel?

Date:
Page:

Monolithic

Users
↓
Shells & Commands
↓
Kernel
↓
Hardware

Kernel → main component of OS loads 1st into memory when OS is booted & remains until OS is shut down.

Monolithic Kernel → all OS services operate in kernel space.

Sockets allow you to exchange info b/w processes on same machine or across a network

Q. Program vs Process vs Threads?

Program	Process	Thread
set of intructions to perform a job (passive entity)	Program under execution. (active entity)	small executable unit of process. Threads share same memory of process. (fast communication)

Notepad.exe → double click → process created → main thread executed

Notepad.exe → program

Types of Processes

Event specific → CPU bound, I/O Bound

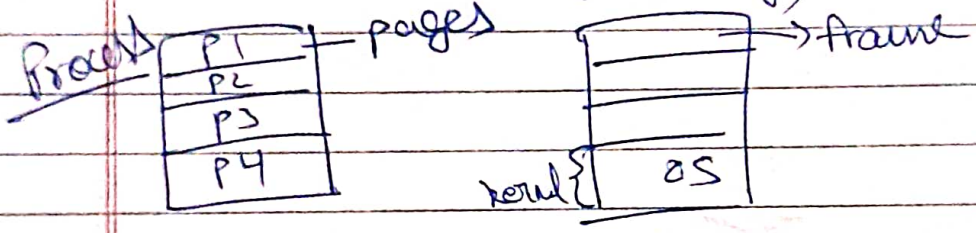
nature → Independent, Cooperative

Date: _____

Page: _____

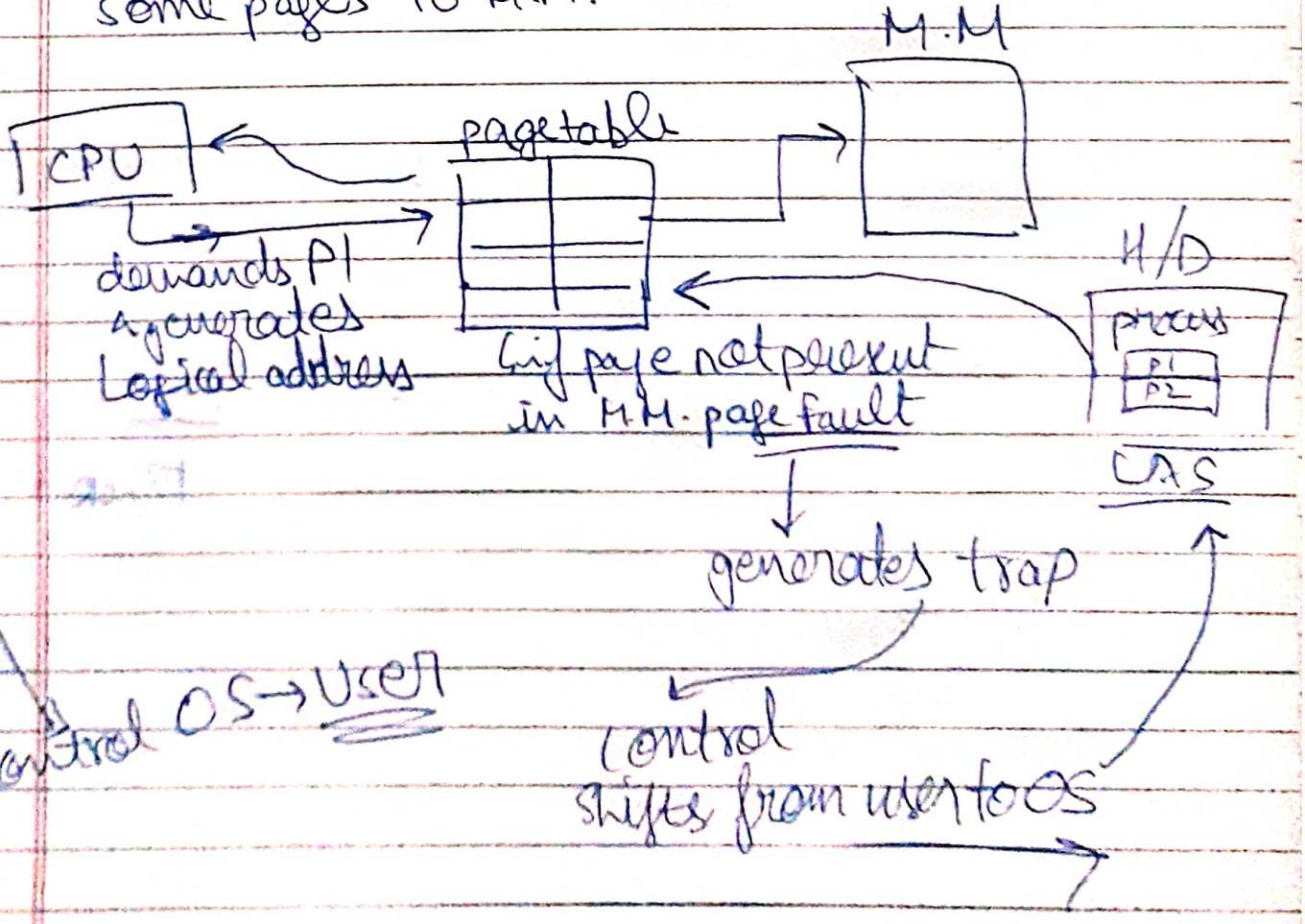
Q Virtual Memory & Thrashing?

Programs of size greater than main memory are executed by virtual memory.
(increased Multiprogramming)



pages.size = frame.size

some pages to H.M.



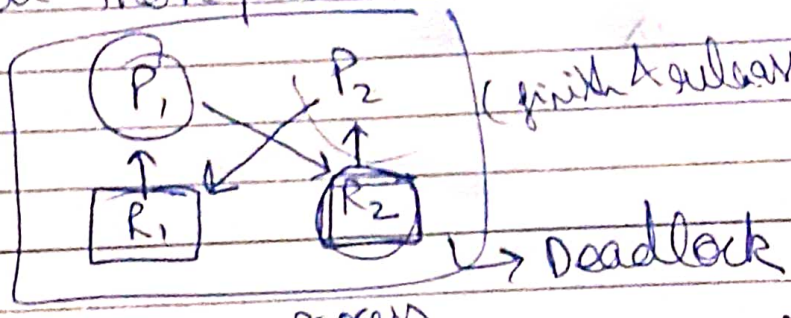
Thrashing → When page fault occur frequently at higher rate

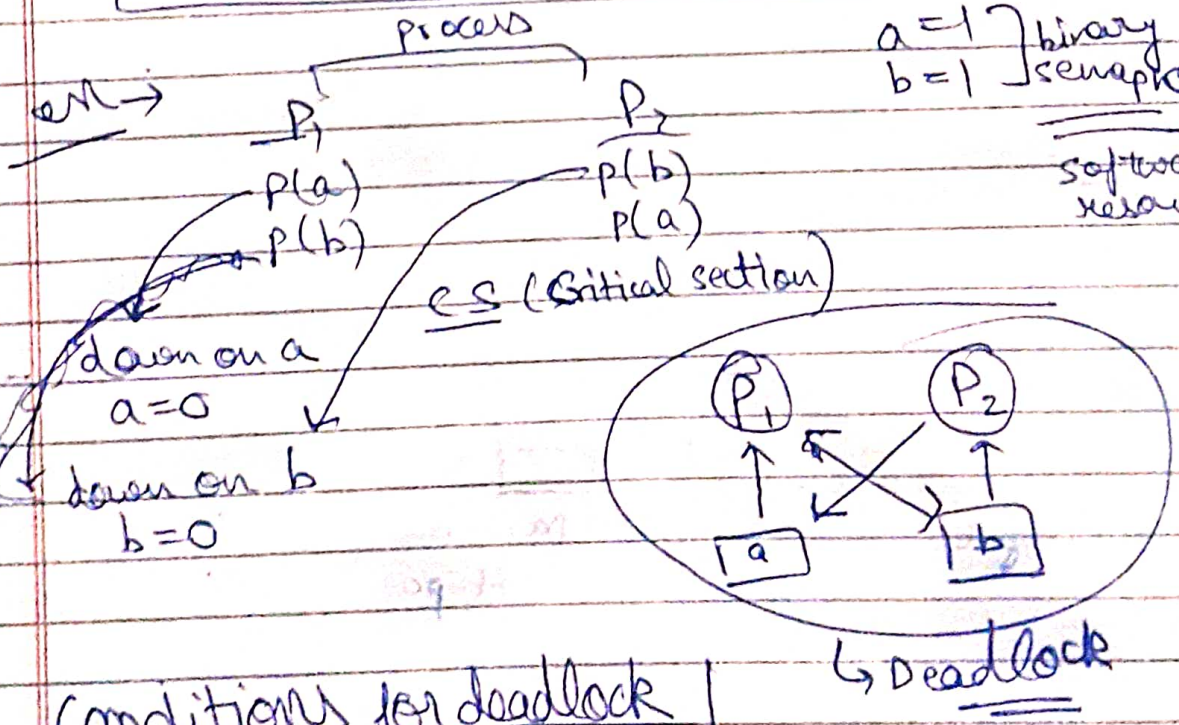
Date:
Page:

3) RAID (Redundant array of independent disks)

4) Deadlock

↳ 2 or more processes are waiting for some event to happen which never occurs then those processes are said to be in deadlock





Conditions for deadlock

- Mutual exclusion
- Hold & wait
- No preemption
- Circular wait

→ Mutual exclusion
↳ One resource used by one process at a time.

(P) → [R]

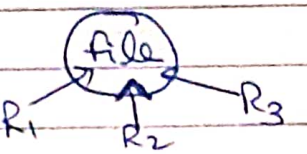
→ Hold & Wait
process holding resources & waiting for other resources

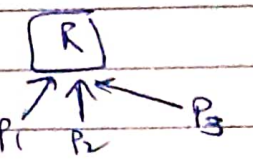
→ No preemption
process will release the resources voluntarily

→ Circular Wait
processes waiting on each other in cyclic way.

Deadlock Prevention (any one satisfy)

→ Mutual Exclusion (false)





→ Hold & wait →
give all resources req. to process before its execution

→ No preemption →
if process is in block state preempt to release resource req. by other process.

→ Circular wait →
→ Assign number to each resource
→ Only request resources at higher levels than it holds

r_i → holding resource
r_j → requesting resource

$r_i < r_j$

Q Spooling?

Data temporarily held to be used & executed by device, system.

Date:
Page:

Q Semaphores?

variable for managing concurrent processes by integer value

2 Types

- Binary (0, 1)
- & Counting

Binary

```
P(Semaphore s)
{
  while (s==0);
  s=s-1;
}
```

```
V(semaphore s)
{
  s=s+1;
}
```

P1, P2 — caught in while

P(s) → -S → -V(s) remaining

S = X 0 1 0 Now P2

Mutual exclusion

P1 counting

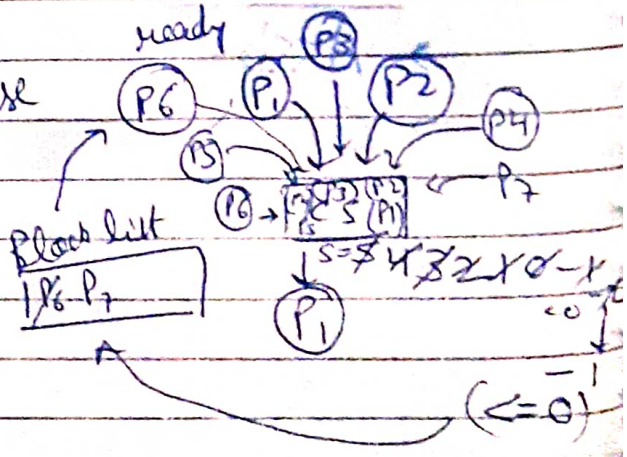
S = Instances of Resource

ex 5 printers

S=5

P(S) entry code

V(S) exit code



Producer / consumer Problem

Date:
Page:

```
int N=5;
int buffer[] = new int[N];
int count=0;
void Producer()
{
  while(1)
{
  int itemP, in;
  ProduceItem(itemP);
  while(count==N);
  Buffer[in] = itemp;
   in=(in+1)%N;
   count= count+1;
```

```
void consumer()
{ int itemc, out;
  while(1)
{ while(count==0);
  itemc = buffer[out];
  out = (out+1)%N
  count = count-1;
  ProcessItem(itemC);
```

lock variable

```
while(1)
{
while(lock!=0);
  lock=1;
  <C-S>
  lock=0;
}
```

Solution

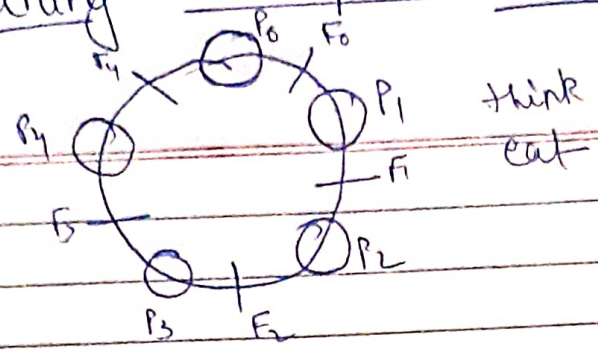
counting semaphore
Binary Semaphore s=1,

```
produceItem(itemp)
  down(empty),
  down(s);
  Buffer[in] = itemp;
   in=(in+1)%n;
   up(full);
   up(s);
}
```

```
consumer
down(full);
down(s);
item = buffer[out];
out = (out+1)%N;
  up(S);
  up(empty);
}
```

Dining Philosophers Problem

Date:
Page:



think
eat

```
void Philosopher (){
while (true)
{
  thinking();
  take_fork(i);
  take_fork((i+1)%N) - Right
  Eat();
  put fork(i);
  put_fork((i+1)%N);
}
}
```

N - no. of forks

CS

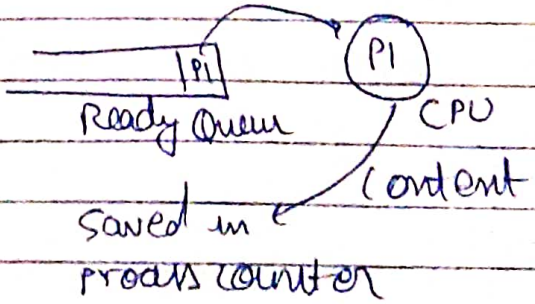
thinking(); S[i]
down(tak_fork(k);
down (tak fork S[(i+1)%n])
Eat();
up(put fork(Si));
up(put_fork S[(i+1)%n])

S_0 S_1 S_2 S_3 S_4 5 semaphores
| | | | |

Mutex is an object
Semaphore is a variable

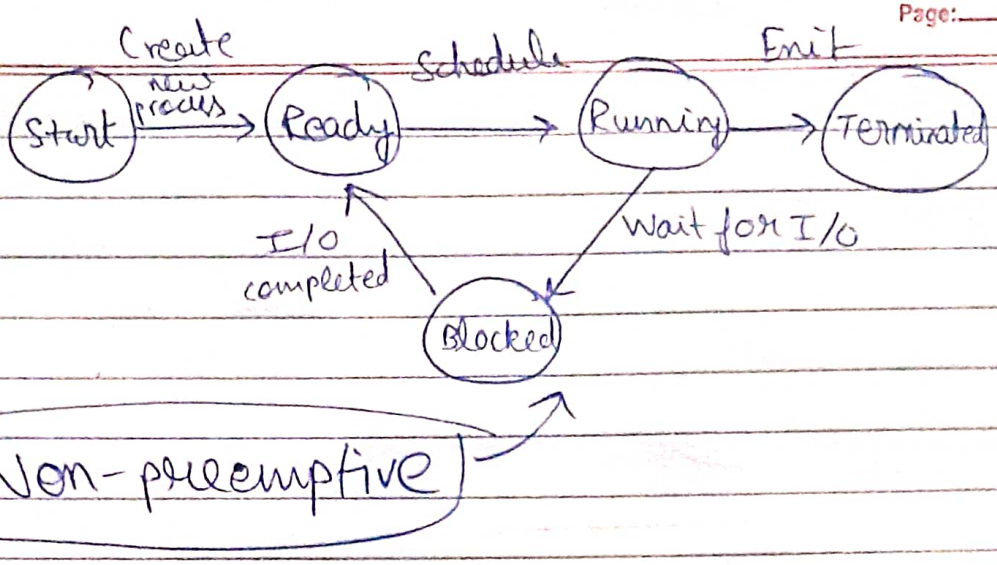
Attributes of the process collectively called as content of the process

- process id
- state
- priority
- program counter



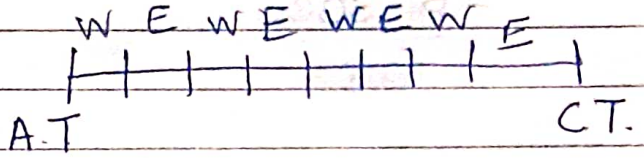
Process State transition Diagram

Date:
Page:



Non-preemptive

CPU Scheduling



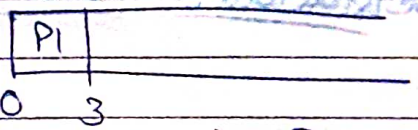
TAT = lifetime

TAT = CT - AT

WT = CT - AT - BT

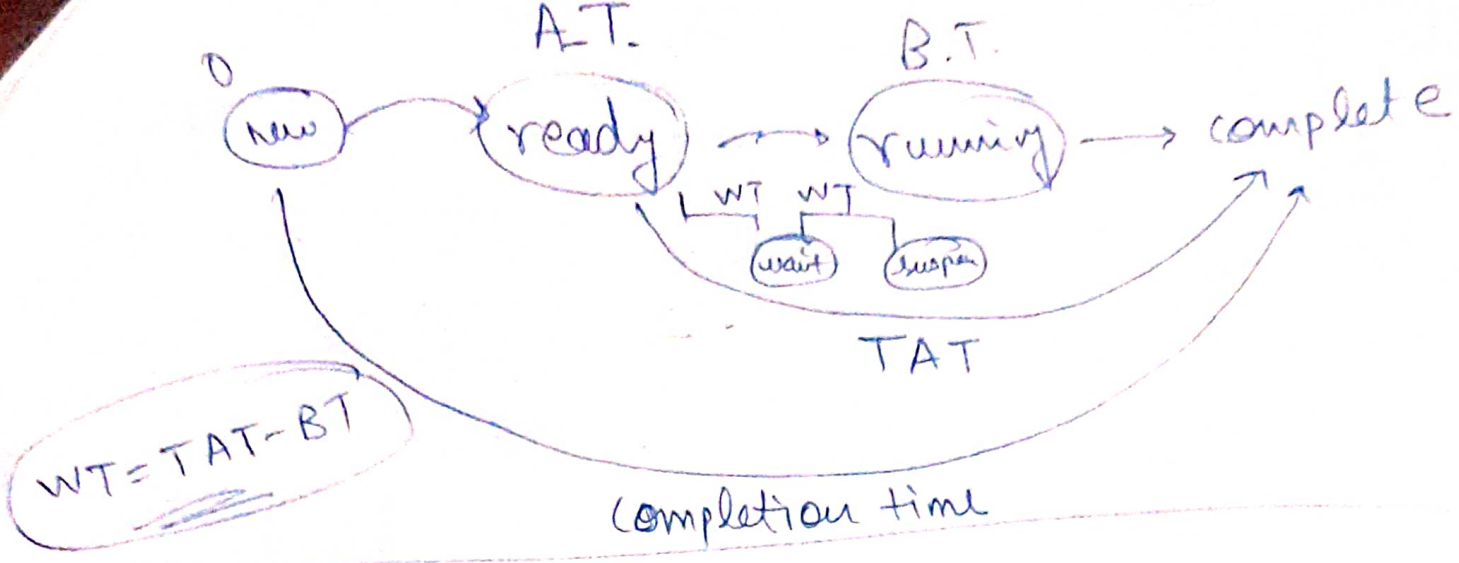
WT = TAT - BT

FCFS



Convoy Effect → High BT process to executed at last in FCFS for lesser ~~A~~ AWT
Av. waiting time

Starvation (unfair) removed lower priority & then high priority + so more high priority such that low priority kept waiting
Starvation

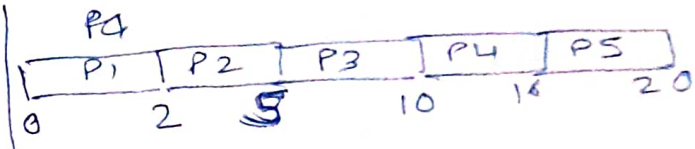


CPU scheduling

First come First serve

PID	AT	BT	CT	TAT	WT
P1	0	2	2	2	0
P2	1	3	5	4	1
P3	2	5	10	8	3
P4	3	6	16	13	7
P5	4	4	20	16	12

Gantt chart



completion clock
- arrival — Rank
TAT
TAT - BT Rank
WT

If more than 1 process has same arrival time then process with lower process ID will be selected

Convoy Effect

If 1st process are CPU bound then I/O bound
Waiting Time ↑ increases.

If 1st process is I/O bound and rest are CPU bound. Waiting time ↓

Shortest Job First (Non-pre-empt)

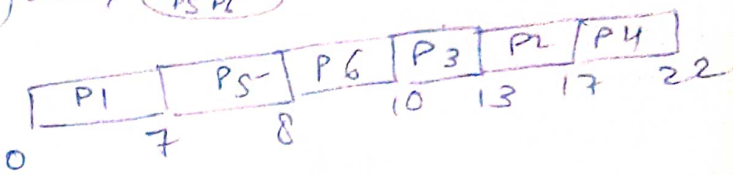
Depends on B.T. → (ready)

→ Should be in ready state

→ Least BT executed 1st

ready 0-7 → ready P2 P3 P4 P5 P6

	AT	BT	CT	TAT	WT
P1	0	7	7	7	0
P2	1	4	17	16	12
P3	2	3	13	11	8
P4	3	5	22	19	14
P5	4	1	8	4	3
P6	5	2	10	5	3



Shortest Remaining Time First (preempt)

BT after every 1sec compare

PID → AT same → BT same

Longest Job First (Non-pre-emtive method)

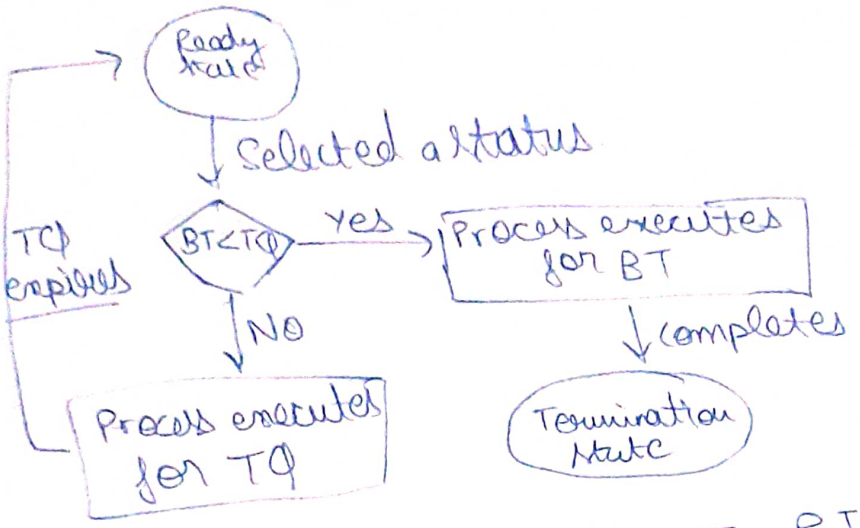
BT↑

Longest Remaining Time First (pre-emptive)

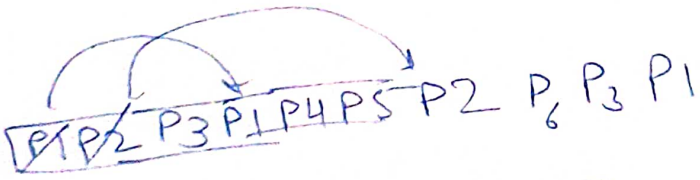
after every 1sec compare

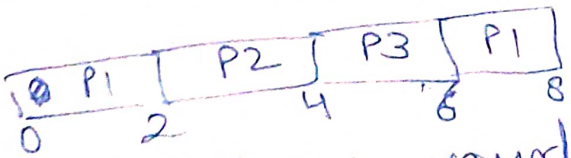
Round Robin Scheduling

Time Quantum (TQ) ← max time for CPU utilization



AT	BT	CT	TAT	WT	RT
0	6				
1	4				
2	5				
3	1				
4	2				
5	3				





if TQ is less in round robin
→ more context switching
→ less response time

if TQ ↑
→ context switching less
→ Response time more