GATE CSE NOTES

by

UseMyNotes

Introduction

* **Operating System:** Program that controls the execution of application programs & acts as an interface between the user of a computer & the computer hardware.

→ OS controls the allocation & use of the computing system's resources among the various user & tasks.

→ It provides an interface between the computer hardware & the programme that simplifies & makes feasible for coding, creation, debugging of application programs.

* **Operating System Services.**

Functions that are helpful to the user:

a) User interface: Varies between command-line (CLI), GUI, batch.

b) Program execution: Loading program & run it.

c) I/O operations:

d) File system manipulation

e) communications. (Shared memory or packets)

f) Error detection: Taking action & debugging

Functions for efficient operation:

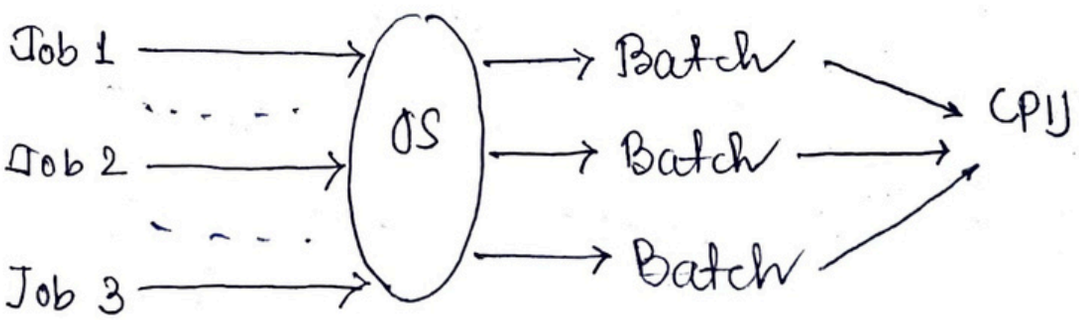
a) Resource allocation

b) Accounting (Keep track of which users use how much & what kinds of computer resources.)

c) Protection & Security

* Types of OS.

1. Batch OS. : Operator takes similar jobs having same requirement & group them into batches.



Adv: Multiple users can share the batch systems.

The idle time for batch system is very less.

It's easy to manage large work repeatedly in batch systems.

Disadv.: Priority can't be set for jobs.

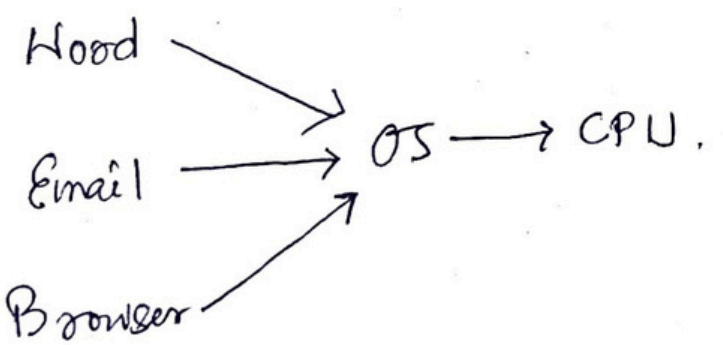
May lead to starvation because of a batch taking long time for execution.

Hard to debug.

eg. Payroll system, Bank statements etc.

2. Time Sharing OS.

Each task is given some time quantum to execute.

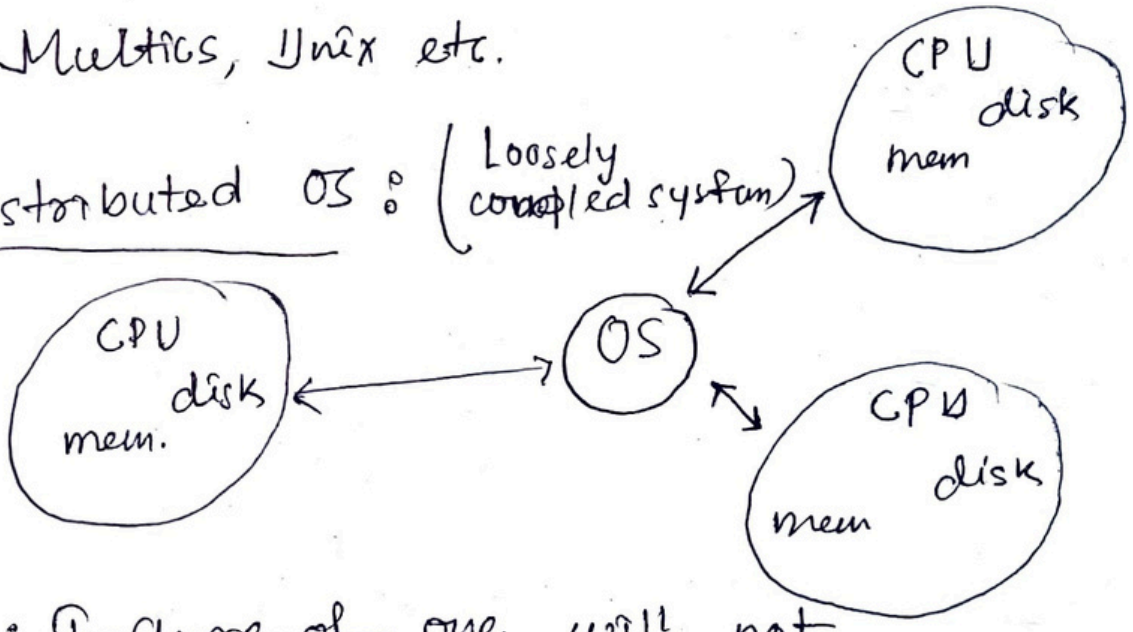


Adv.: Each task gets equal opportunity, less chances of duplication of S/W, CPU idle time can be reduced.

Disadv.: Reliability problem, data communication problem.

eg. Multics, Unix etc.

3. Distributed OS: (Loosely coupled system)



Adv.: Failure of one will not affect the other network communication.

Computation highly fast (as resources are shared).

Scalable & less load on host computer.

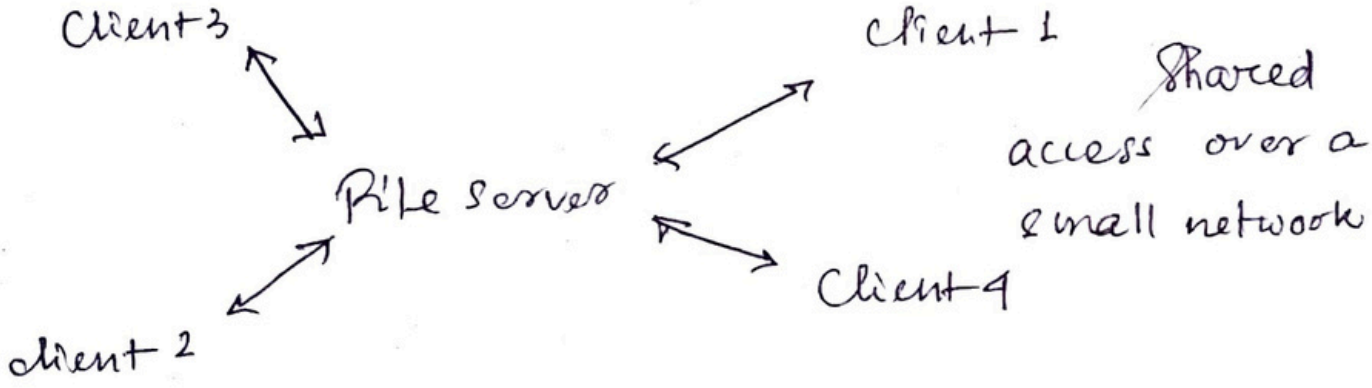
Delay on data processing reduces.

Disadv.: Failure of main network will stop the entire communication.

Very expensive.

eg. LOCUS.

4. Network OS. : (Tightly coupled system).



Adv.: Highly stable centralised servers.

Security concerned handled through servers

Server accesses are possible remotely.

Disadv.: Costly servers.

Maintenance & updates required regularly.

Eg.: Ms Windows server 2003, Windows server 200

BSD etc.

5. Real time OS (RTOS):

Time interval & response time are very small.

2 types:

i) Hard RTOS: Time constraints are very strict. Virtual memory never found.

ii) Soft RTOS: Time constraint less strict.

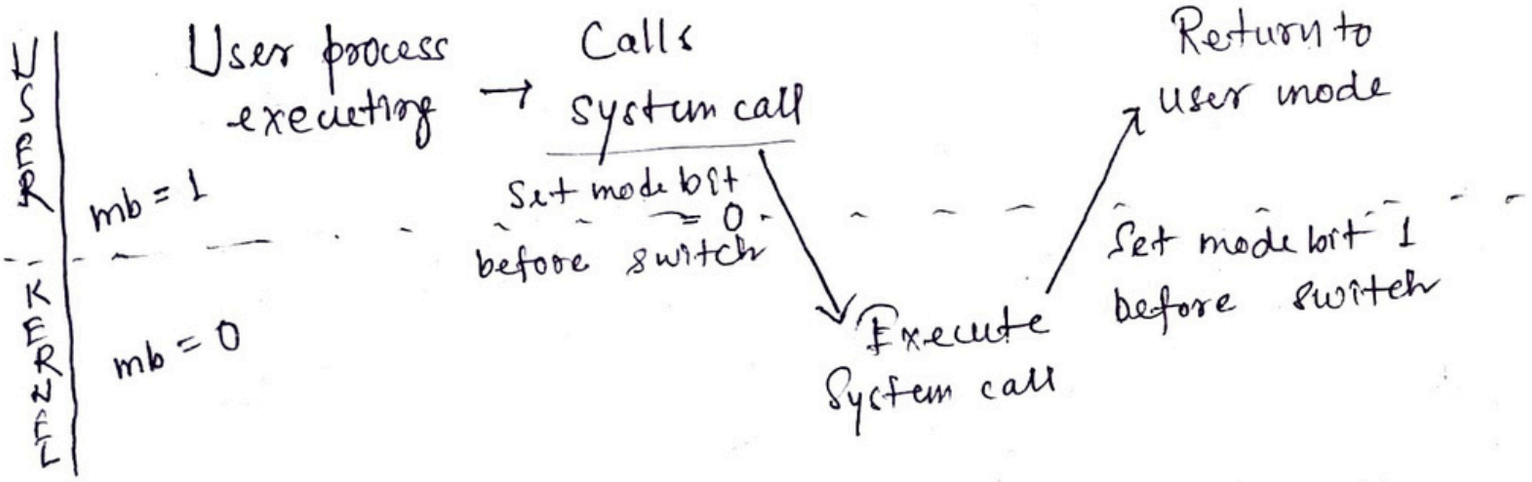
Adv.: Maximum consumption, task shifting, focus on application, RTOS can be used in embedded systems, error free, memory allocation best managed.

Disadv.: Limited tasks, heavy system resources usage, complex algo., needs specific device drivers & interrupt signals.

eg.: Weapon systems, air traffic control system, QNX, VxWorks, RTLinux, FreeRTOS

* Dual Mode operations in OS :

1) **User Mode :** When the computer system runs user app. then the system is in user mode. When the user app. requests for a service from the OS or an interrupt occurs in system or system call, then there will be transition from user mode (1) to kernel mode (0).



2) **Kernel Mode :** When system boots then H/W starts in kernel mode & when OS is loaded then it starts user app. in user mode. We have privileged instructions that execute in kernel mode.

↳ [handling interrupts, I/O mgmt.]

* Privileged instructions.

Can run only in kernel mode.

→ Any attempt to execute p-ins^{n}s in user mode, is treated as an illegal ins^{n}. H/W traps it to the O/S.

→ Any ins^{n} that can modify the contents of the timer is a privileged ins^{n}.

→ P. ins^{n}s are used by the OS in order to achieve correct operation.

→ eg. I/O ins^{n}s & halt ins^{n}s, turn off all interrupts, setting timer, context switching, clearing memory, modifying entries in device-status table.

* Non-privileged instructions.

Can run only in user mode.

eg. Reading status of processes, reading system time, generating any trap ins^{n}, sending final printout of printer.

* Functions of an OS

Processor management, Device management, Buffering, Spooling (Simultaneous peripheral operation on line), memory mgmt., partitioning Virtual memory, File mgmt.

* System Calls.

Programming interface to the services provided by the OS. System calls are accessed by programs via a high level API.

Types:

a) Process control
b) File mgmt.
c) Device mgmt.
d) Information maintenance
e) Communications

a) fork(), exit(), wait()

b) open(), read(), write(), close()

c) ioctl(), read(), write()

SetConsoleMode | read console ....

d) getpid(), alarm(), sleep()

e) pipe(), shmget(), mmap()

create file mapping

map view of file.

Protection —

chmod(), set file security

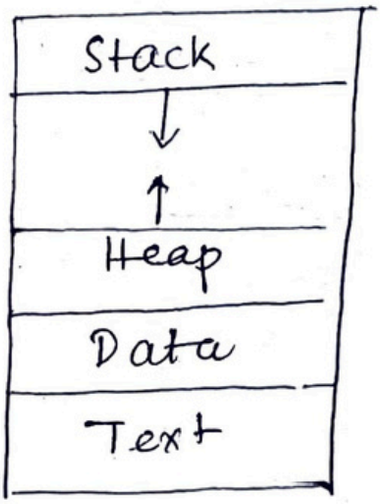
chown()

Set security descriptor group

Process Management

*Process : A program in execution.
(Active entity)

*Process in memory



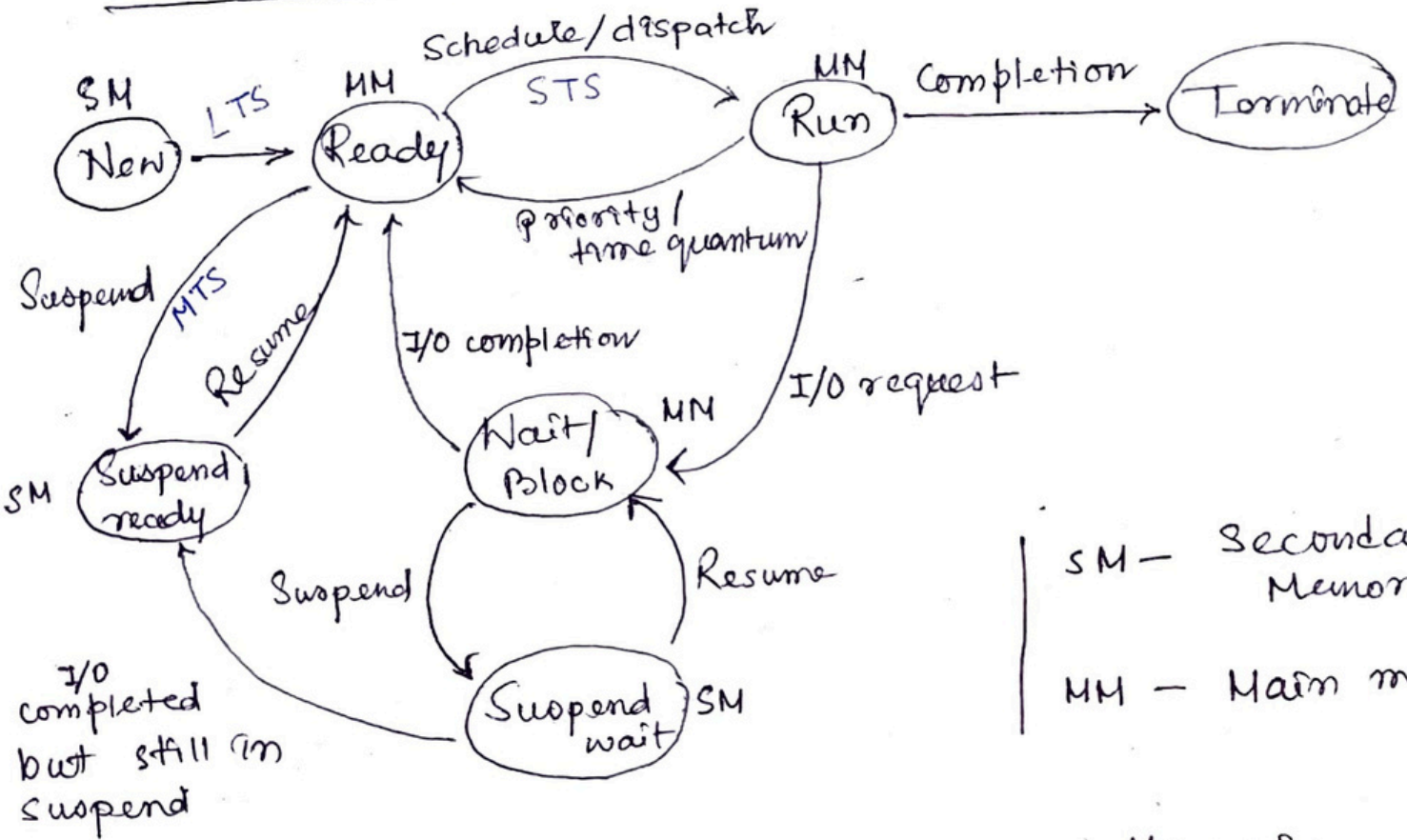
*Process attributes.

i) Identifier : Unique id for process
ii) State
iii) Priority
iv) Program counter
v) Memory pointers.
vi) Context data
vii) I/O status information
viii) Accounting information

Suspend ready: Process that was initially in the ready state but were swapped out of MM & placed onto external storage by scheduler are said to be in suspend ready. They will transition back to ready whenever the process is again brought onto MM.

Suspend wait: Process was performing I/O & lack of MM caused them to move to SM

* Process States.



→ A process necessarily goes through minimum 4 states. (new, ready, run, terminate). Process requiring I/O, min. no. of states is 5.

→ A single processor can execute only one process at a time.

→ Moving a process from wait to suspend wait is a better alternative, as the process is already waiting for some blocked resource.

* Context Switching

Process of saving context of one process & loading the context of another process.

Context switching happens when -

a) a high priority process comes to ready state

b) an interrupt occurs

c) User & kernel mode switch

d) preemptive CPU scheduling used.

* CPU - bound & I/O - bound process

CPU - bound processes require more CPU time or spends more time in running state. (Intensive in CPU operation)

I/O - bound processes require more I/O time. More time in waiting state. (Intensive on I/O operations)

* Mode switch time < Context switch time

* Multiprogramming : Multiple processes in ready state.

1. With preemption –

Forcefully removed process from CPU.

(Time sharing, multitasking)

2. Non-preemption –

Not removed until execution.

Degree of multiprogramming is the maximum no. of processes that can be present in the ready state.

→ Optimal degree of multiprogramming means average rate of process creation is equal to the average departure rate of processes from main memory.

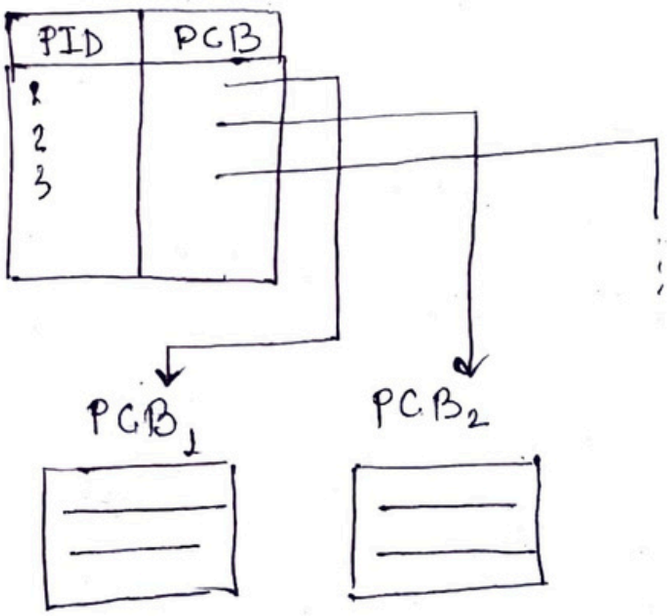
* Process table

Data structure maintained by the OS to facilitate context switching & scheduling

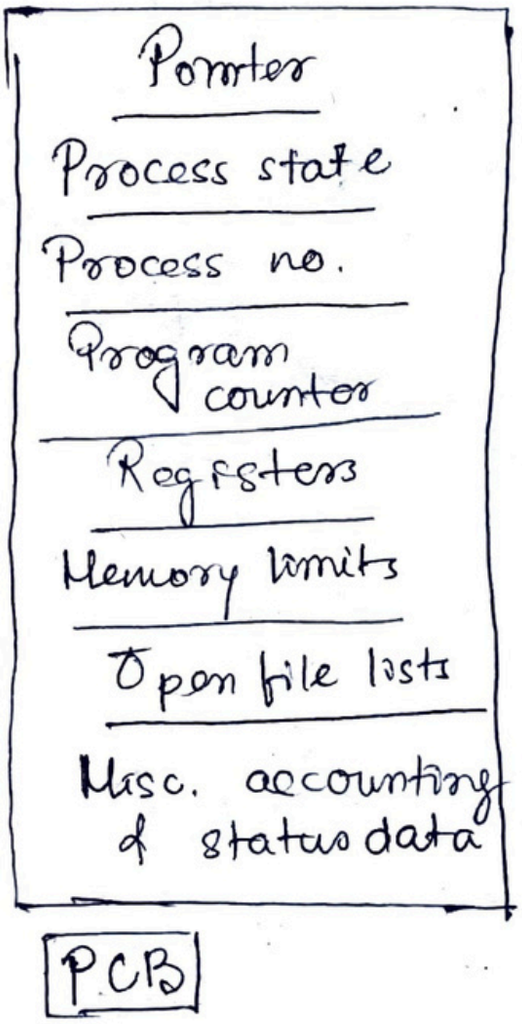
It contains all the current processes' information or PCBs in the system.

*Process Control Block.

Data structure that contains information of the process related to it.



→PCB of each process resides in MM.



PCB

* Schedulers.

Help in scheduling the processes in various ways. Responsible for selecting the jobs to be submitted into the system & deciding which process to run.

→ Types of schedulers.

1. Long term scheduler : Also job scheduler.
performance
It selects a balanced mix of I/O bound & CPU-bound processes from the secondary memory (new). Then it loads the selected processes onto the MM (ready) for execution.

Primary objective of LTS is to maintain a good degree of multiprogramming. (New → Ready)

2. Short term scheduler : Also CPU scheduler.
context switching time.
It decides which process to execute next from the ready queue. After STS decides the process, dispatcher assigns the decided process to the CPU for execution. (Ready → Run)

Primary objective of STS is to increase the system performance.

Software that moves processes from ready to run state and vice versa.

3. Medium term scheduler.

swapping time

Swaps out the processes from MM to secondary memory to free up the MM when required. It reduces degree of multiprogramming. After some time when MM becomes available, MTS swaps in the swapped out processes to the MM & its execution is resumed from where it left off.

Primary objective is swapping. Swapping may be required to improve the process mix.

→ Dispatcher.

A dispatcher is a special program which comes into play after the scheduler. Dispatcher takes the process to the desired state/queue after scheduler selects the process.

It involves - context switch, switching to user mode, jumping to the

proper location on the user program to restart that program.

Dispatcher	Scheduler.
i) Module that gives control of CPU to the process selected by STS.	i) Selects process among various processes.
ii) It's a code segment. No types.	ii) 3 type, L, M, S.
iii) Dependent on scheduler for working.	iii) Works independently.
iv) No specific algo.	iv) Works on various algo - SJF, FCFS etc.
v) Time taken is dispatch latency.	v) Time taken is negligible.
vi) Also responsible for context switches, switching to user mode.	vi) Only work is selection of processes.

* Various times related to process.

1. Arrival time: Process enters the ready queue.

2. Waiting time: Time spent by the process waiting in the ready queue for getting CPU.

Waiting time = Turn around time − Burst time

3. Response time: Time after which a process gets the CPU for the first time after entering the ready queue.

Response time = Time at which process 1st gets CPU − arrival time

4. Burst time: Time required by a process for executing on CPU.

Also called running time. It can only be known after the process has been executed.

5. Completion time:

Time at which a process completes its execution on the CPU & takes exit from the system.

6. Turn around time:

Total time spent by a process in the system.

- Turn around time = Burst time + Waiting time

or

- Turn around time = Completion time − Arrival time

*Objective of Process Scheduling Algorithm.

1. Maximum CPU utilisation
2. Fair allocation of CPU
3. Maximum throughput [No. of processes that complete their execution per time unit]
4. Minimum Turn around time
5. Minimum waiting time
6. Minimum response time.

*Different Scheduling Algorithms.

- i) First come first serve (FCFS)
- ii) Shortest job first (SJF)
- iii) Longest job first (LJF)
- iv) Shortest remaining time first (SRTF)
- v) Longest remaining time first (LRTF).
- vi) Round robin scheduling
- vii) Priority based scheduling (Non-preemptive)
- viii) Highest response ratio next (HRRN)
- ix) Multilevel queue scheduling
- x) Multilevel feedback queue scheduling.

* Preemptive Scheduling.

Used when a process switches from running state to ready state or from waiting state to ready state. Resources are allocated to the process for the limited amount of time & then is taken away & the process is again placed back in the ready queue if it has burst time remaining.

Algo based on this —

Round Robin, SJF, priority etc.

* Non-preemptive Scheduling

Used when a process terminates or a process switches from running to waiting state. Once the resources are allocated to a process, process holds the CPU till it gets terminated or it reaches a waiting state.

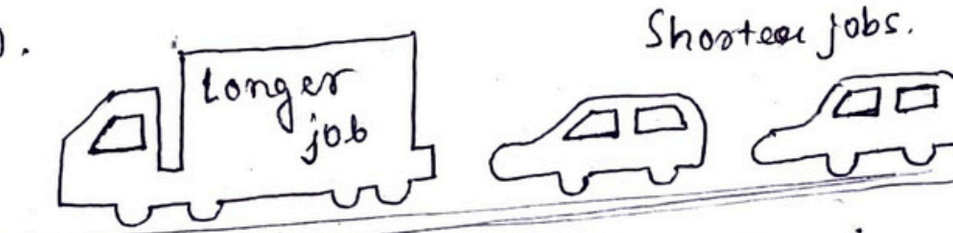
Algo based on this -
SRTF, priority etc.

* First Come First Serve (FCFS) Scheduling

Process arriving first in the ready queue is firstly assigned the CPU. In case of a tie, process with smaller process id is executed first. It's always non-preemptive in nature..

Adv.: Simple & easy, doesn't lead to starvation.

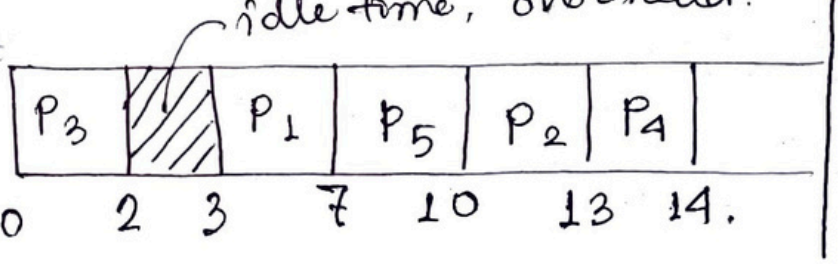
Disadv.: • Doesn't consider priority or burst time of process.
• Convoy effect - Processes with higher burst time arrived before the processes with smaller burst time. Then, smaller processes have to wait for a long time for longer processes to release the CPU.



eg.

PID	Arrival Time	Burst time	Exit time	Turn Around time	Waiting time
P1	3	4	7	4	0
P2	5	3	13	8	5
P3	0	2	2	2	0
P4	5	1	14	9	8
P5	4	3	10	6	3

Gantt chart.



avg. TAT = $\frac{4+8+2+9+6}{5}$

= 5.8

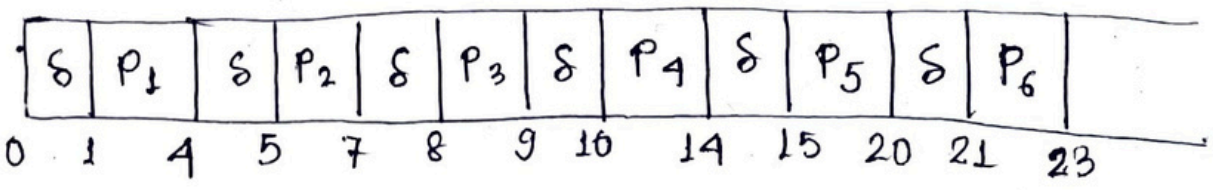
avg WT = 3.2

eg. 1 unit of overhead in scheduling the processes. Find out efficiency of algo.

PID	AT	BT
1	0	3
2	1	2
3	2	1
4	3	4
5	4	5
6	5	2

$\eta = \left(1 - \frac{6}{23}\right) \times 100\%$.

$= 73.91\%$



* SJF, SRTF Scheduling.

SJF - CPU is assigned to the process having lowest burst time.
In case of a tie, it is broken by FCFS.

SJF

- Non-preemptive mode
- Preemptive Mode (SRTF)

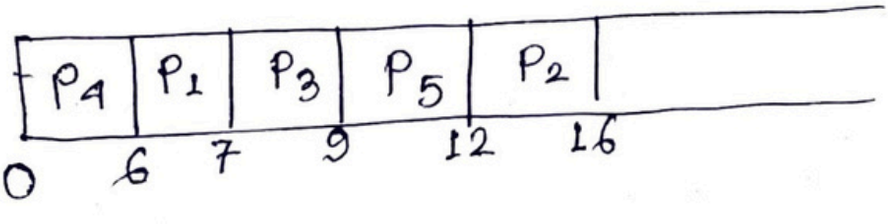
Adv.: SRTF is optimal & guarantees the minimum average waiting time.

Disadv.: Can't be implemented practically since burst time can't be known in advance. It leads to starvation of processes with larger burst time. Priorities can't be set for the processes. Processes with larger burst time have poor response time.

eg.

PID	AT	BT	Exit time	TAT	WT
1	3	1	7	4	3
2	1	4	16	15	11
3	4	2	9	5	3
4	0	6	6	6	0
5	2	3	12	10	7.

SJF

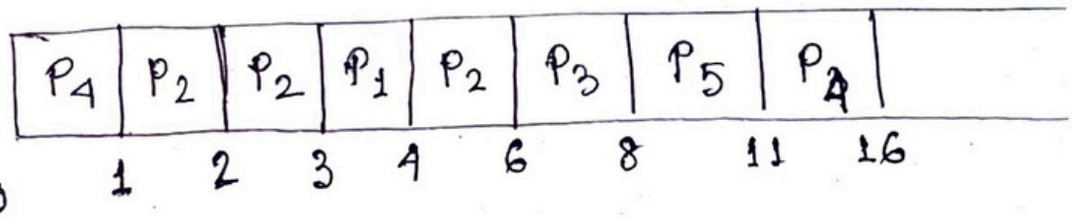


eg

PID	AT	BT	ET	TAT	WT
1	3	1	4	1	0
2	1	4	6	5	1
3	4	2	8	4	2
4	0	6	16	16	10
5	2	3	11	9	6

SRTF

preemptive SJF



Implementation.

Practically can't be implemented, but theoretically can be implemented.

Min heap can be used, where root element contains the least burst time process

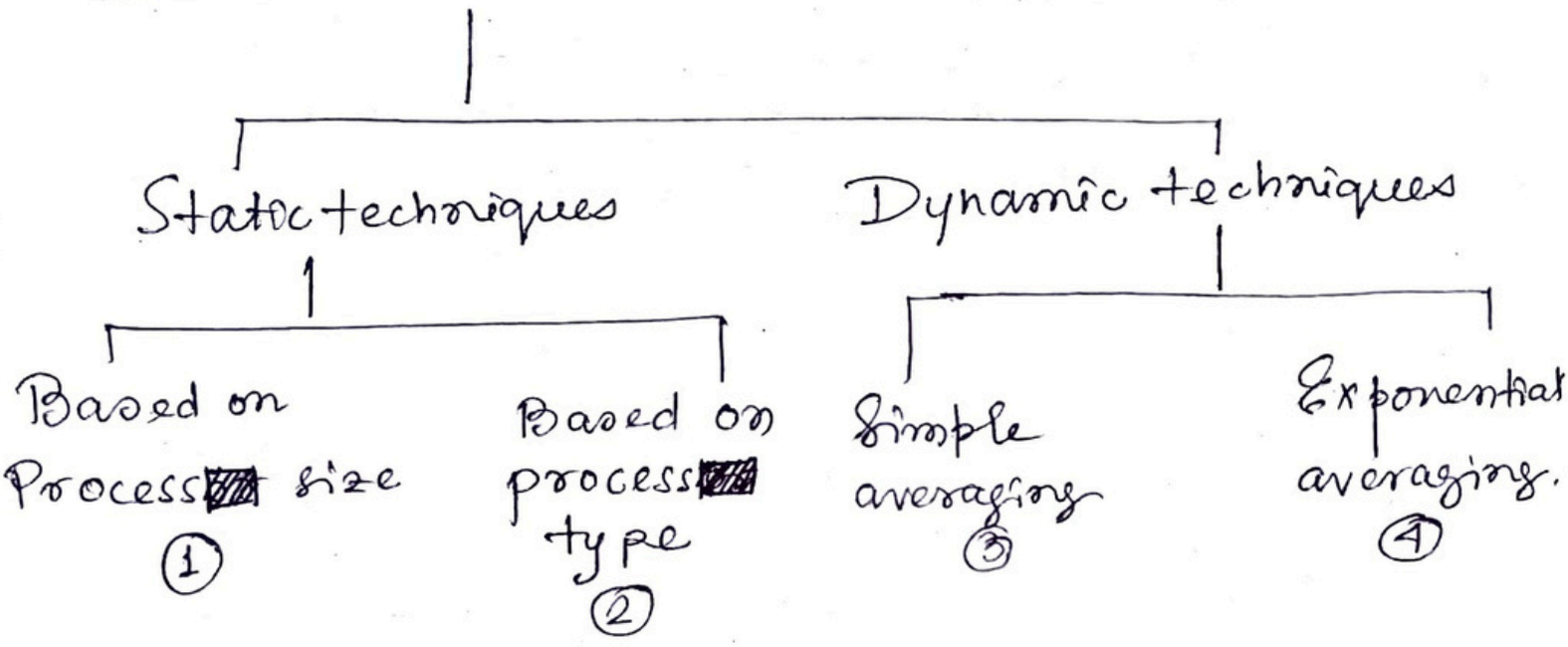
Time complexity for n processes

$$n \times (\log n + \log n) \equiv n \log n$$

adding element in min heap

deleting element from min heap

* Techniques to predict burst time.

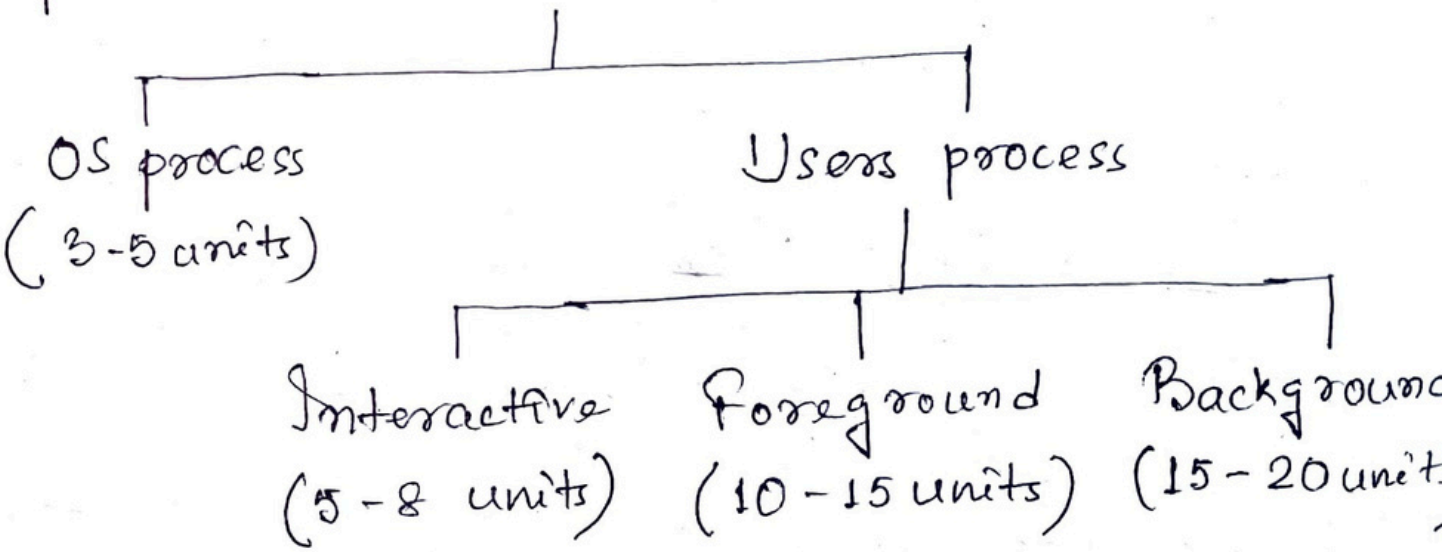


① Process size : This technique predicts the burst time for a process based on its size. Burst time of the already executed process of similar size is taken as the burst time for the process to be executed.

Predicted burst time may not always be right. This is because the burst time of a process also depends on what kind of a process it is.

(2) Process type:

Predicts the burst time for a process based on its type.



(3) Simple averaging

Given n processes of burst time of each process P_i is t_i, then predicted burst time for process P_{n+1} is

$$T_{n+1} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

avg of all burst times till now.

(4) Exponential averaging

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n$$

α – Smoothening factor $(0 \leq \alpha \leq 1)$

t_n – actual burst time of process P_n

T_n – predicted burst time of P_n

eg. $T_5 = ?$ | $t_1 = 4$, $t_2 = 8$, $t_3 = 6$, $t_4 = 7$ | $T_1 = 10$, $\alpha = 0.5$

$T_2 = 0.5 \times 4 + 0.5 \times 10 = 7$

$T_3 = 0.5 \times 8 + 0.5 \times 7 = 7.5$

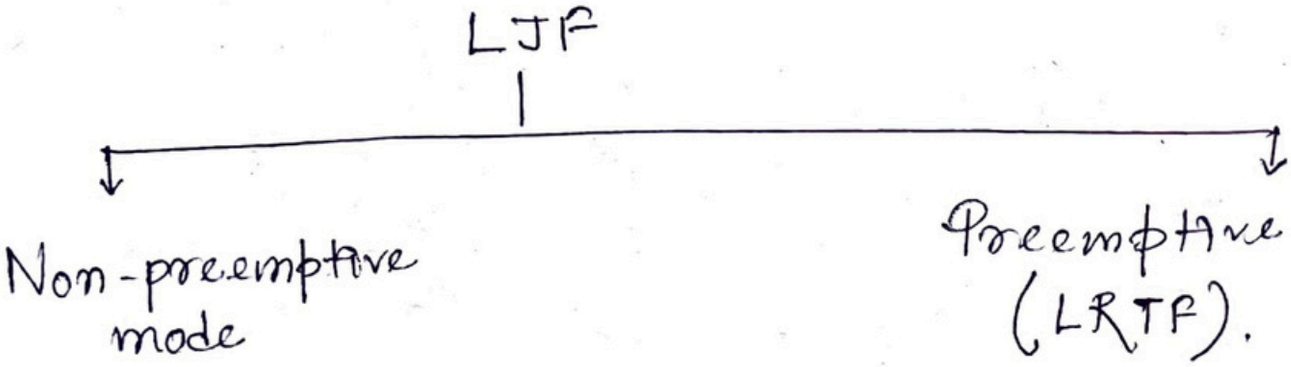
$T_4 = 0.5 \times 6 + 0.5 \times 7.5 = 6.75$

$T_5 = 0.5 \times 7 + 0.5 \times 6.75 = 6.875$ (Ans)

* LJF, LRTF Scheduling.

LJF - CPU is assigned to the processes, having largest burst time.

In case of a tie, it is broken by FCFS.



Adv.: No process can complete until the longest job also reaches its completion

All the processes approx. finishes at same time.

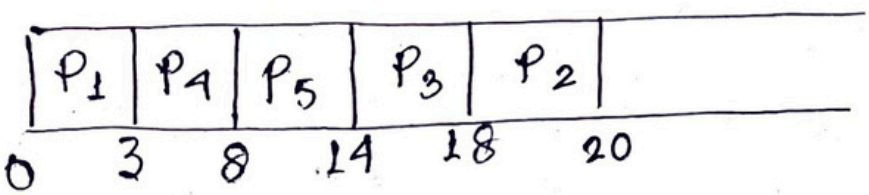
Disadv.: The WT is high.

Processes with smaller BT may starve for CPU.

eg.

LJF Non-preemptive

PID	AT	BT	ET	TAT	WT
1	0	3	3	3	0
2	1	2	20	19	17
3	2	4	18	16	12
4	3	5	8	5	0
5	4	6	14	10	4.

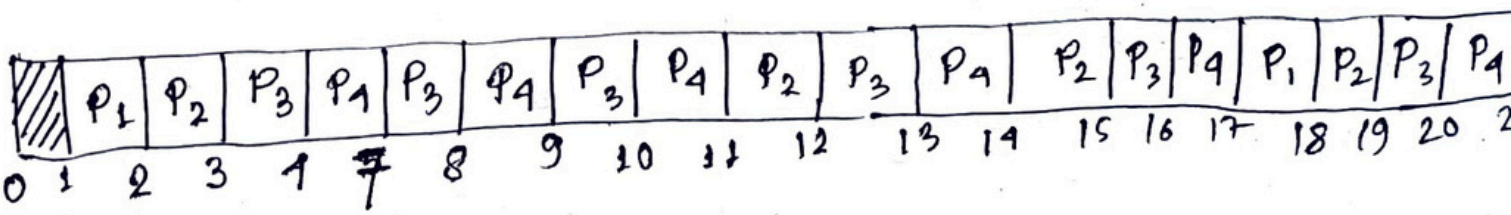


** eg.

LJF preemptive.

LRTF

PID	AT	BT	ET	TAT	WT
1	1	~~2~~ 1	18	17	15
2	2	~~4~~ ~~2~~ 1	19	17	13
3	3	~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ 1	20	17	11
4	4	~~8~~ ~~7~~ ~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ 1	21	17	9.



* HRRN Scheduling.

$$\text{Response ratio} = \frac{WT + BT}{BT}$$

CPU is assigned to the process having highest response ratio. In case of tie, it is broken by FCFS. Operates only in non-preemptive mode.

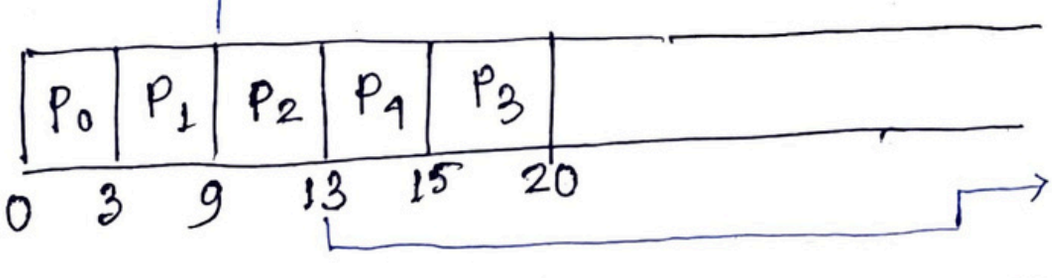
Adv.: It performs better than SJF. It not only favours the shorter jobs but also limits the waiting time of longer jobs.

Disadv.: Can't be implemented practically, as burst time can't be known in advance.

eg.

PID	AT	BT	ET	TAT	WT
0	0	3	3	3	0
1	2	6	9	7	1
2	4	4	13	9	5
3	6	5	20	14	9
4	8	2	15	7	5

Non-preemptive



i) $RR_2 = \frac{5+4}{1} = 2.25$

$RR_3 = \frac{3+5}{5} = 1.6$

$RR_4 = \frac{1+2}{2} = 1.5$

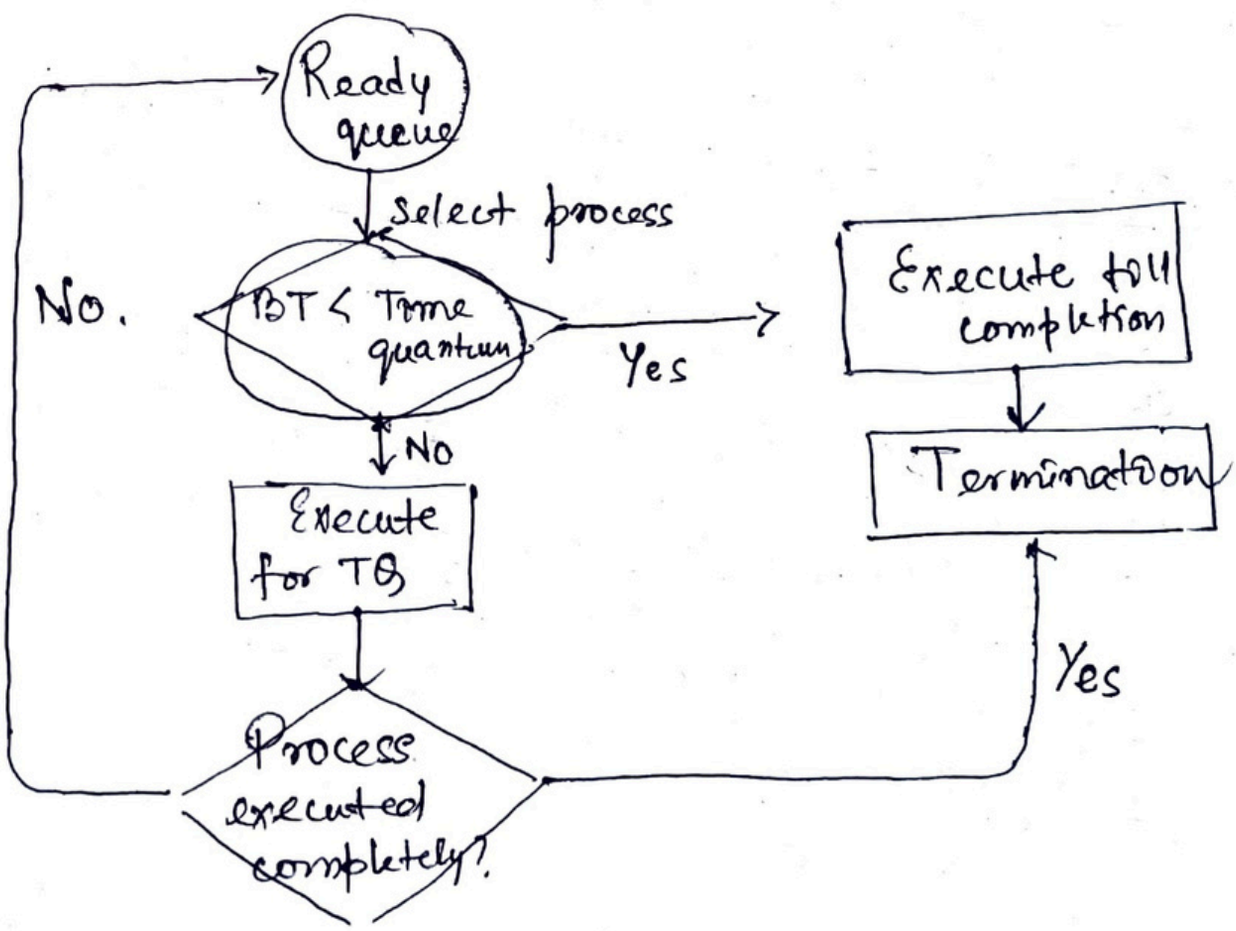
ii) $RR_3 = \frac{7+5}{5} = 2.4$

$RR_4 = \frac{5+2}{2} = 3.5$

* Round Robin Scheduling.

CPU assigned to the process on the basis of FCFS for a fixed amount of time. (time quantum). After the time quantum expires, the process is preempted & sent to the ready queue. It's always preemptive in nature.

RR scheduling is FCFS with preemptive mode.



Adv.: It gives the best performance in terms of average response time. It's best suited for time sharing system, client server architecture & interactive system.

Disadv.: It leads to starvation for processes with larger burst time as they have to repeat the cycle many times. Performance depends on TQ. Priorities can't be set for the processes.

→ With decreasing value of TQ,

i) No. of context switch increases

ii) Response time decreases.

iii) Chances of starvation decreases.

Smaller TQ is better in terms of response time.

→ Higher value of TQ is better in terms of no. of context switches.

→ With increasing value of TQ, RR tends to become FCFS.

When TQ → ∞, RR becomes FCFS.

→ Value of TQ should be neither too big nor too small.

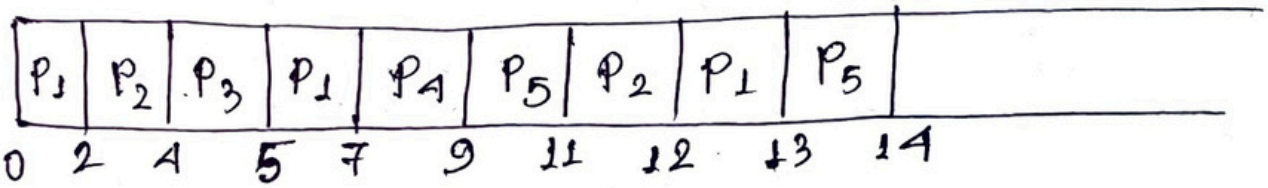
Eg.

PID	AT	BT	ET	TAT	WT
1	0	5 ~~3~~ 1	13	13	8
2	1	~~3~~ 1	12	11	8
3	2	~~1~~	5	3	2
4	3	~~2~~	9	6	4
5	4	~~3~~ 1	14	10	7.

TQ = 2

Avg. TAT = $\frac{13+11+3+6+10}{5}$

$= \frac{43}{5} = 8.6.$



Ready queue: ~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P4~~ ~~P5~~ ~~P2~~ P1 P5

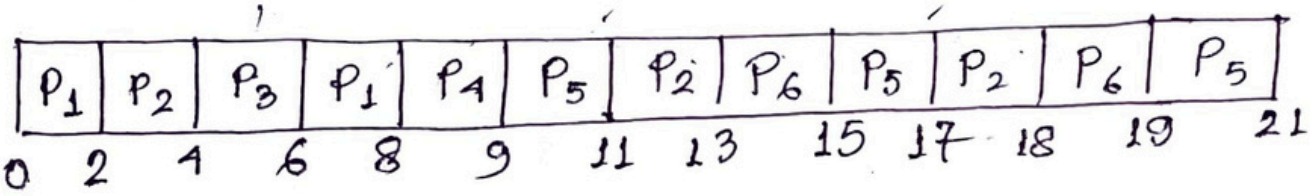
eg.

PID	AT	BT
1	0	~~4~~ ~~2~~
2	1	~~5~~ ~~3~~ ~~1~~
3	2	~~2~~
4	3	~~1~~
5	4	~~6~~ ~~4~~ ~~2~~
6	6	~~3~~ ~~1~~

TQ = 2

FCFS with preemption

In queue, first add processes which has come, then add the concerned process if BT is remaining.



Queue. ~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P4~~ ~~P5~~ ~~P2~~ ~~P6~~ ~~P5~~ ~~P2~~ ~~P6~~ ~~P5~~

* Selfish Round Robin Scheduling. Better service to processes that have been executing for a while than to newcomers.

Implementation: i) Processes in ready list are partitioned into NEW & ACCEPTED lists. ii) New processes wait while accepted processes are serviced by RR. iii) Priority of a new process increases at a rate 'a' while the priority of an accepted process increases at a rate 'b'. iv) When the priority of a new process reaches the priority of an accepted process, that new process becomes accepted. v) If all accepted processes finish, the highest priority new process is accepted.

Algorithm: i) No ready processes initially, when 1st process A arrives. It has priority 0 to begin with. Since, there are no other accepted processes, A is accepted immediately. ii) After a while another process B arrives. As long as $b < a$, B's priority will eventually catch up to A's, so it's accepted. Now, both A & B have same priority. iii) All accepted processes share a common priority (which rises at rate b); that makes this policy easy to implement- i.e. any new process's priority is bound to get accepted at some point. So, no process has to experience starvation. iv) Even if $b > a$, A will eventually finish & then B can be accepted.

Adjusting a & b:

- if $b/a \geq 1$, a new process isn't accepted until all the accepted processes have finished
  SRR becomes FCFS.
- if $b/a = 0$ all processes accepted immediately
  SRR becomes RR
- if $0 < b/a < 1$ accepted processes are selfish, but not completely.

* Priority based scheduling: (Preemptive & non-preemptive)

CPU is assigned to the process having highest priority. In case of a tie, it's broken by FCFS.

Adv.: Considers priority of the processes & allows the important processes to run first. (Priority scheduling in preemptive mode is best suited for real time OS.)

Disadv.: Processes with lesser priority may starve. There's no idea of response time & waiting time.

→ Waiting time for the process having the highest priority will always be zero in preemptive mode, & may not be zero in non-preemptive mode.

→ Priority scheduling in preemptive & non-preemptive mode behaves exactly same under following conditions:

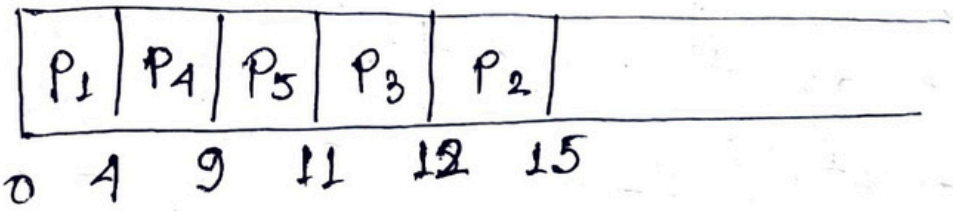
i) arrival time of all processes same,

ii) all processes become available.

eg.

non-preemptive

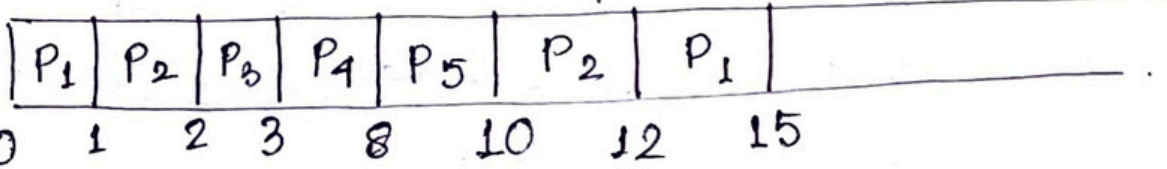
PID	AT	BT	Prio
1	0	4	2
2	1	3	3
3	2	1	4
4	3	5	5
5	4	2	5



eg.

PID	AT	BT	Prio
1	0	13	2
2	1	32	3
3	2	1	4
4	3	5	5
5	4	2	5

preemptive



* Each queue gets a certain portion of CPU time & it can then schedule among its various processes eg. foreground q can be given 80% of CPU time for RR sched, whereas background q receives 20% of CPU to give to its processes on an FCFS basis.

* Multilevel Queue Scheduling:

- Ready queue divided into separate queues for each class of processes (eg. foreground (interactive) & background processes (batch))

High priority

System processes → Queue 1

Interactive processes → Queue 2

Batch process → Queue 3

Low priority

Each queue has its own scheduling algorithm.

Different sched. needs. Foreground processes may have priority (externally defined) over background processes.

Each queue has own sched. algo. Processes permanently assigned to one queue. Foreground queue - RR sched; Background - FCFS. There must be scheduling among queues (fixed prio. preemptive)
- Time slice among queues. →

→ For scheduling among the queues –

i) Fixed priority preemptive scheduling method.

ii) Time Slicing.

Each queue has absolute priority over lower priority queue.

MCQS has low sched. overhead but it's inflexible.

Each queue gets certain portion of CPU time & can use it to schedule its own processes.

eg.

PID	AT	BT	Q No.
1	0	4	1
2	0	3	1
3	0	8	2
4	10	5	1

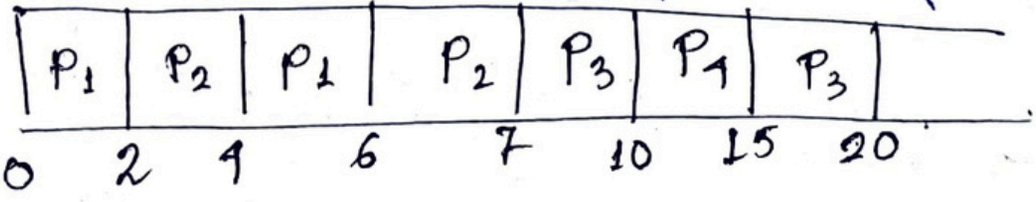
Queue 1 has higher priority.

ref. gfg

Queue 1 – RR (TQ = 2)

Queue 2 – FCFS.

because of fixed priority preemption among queues.



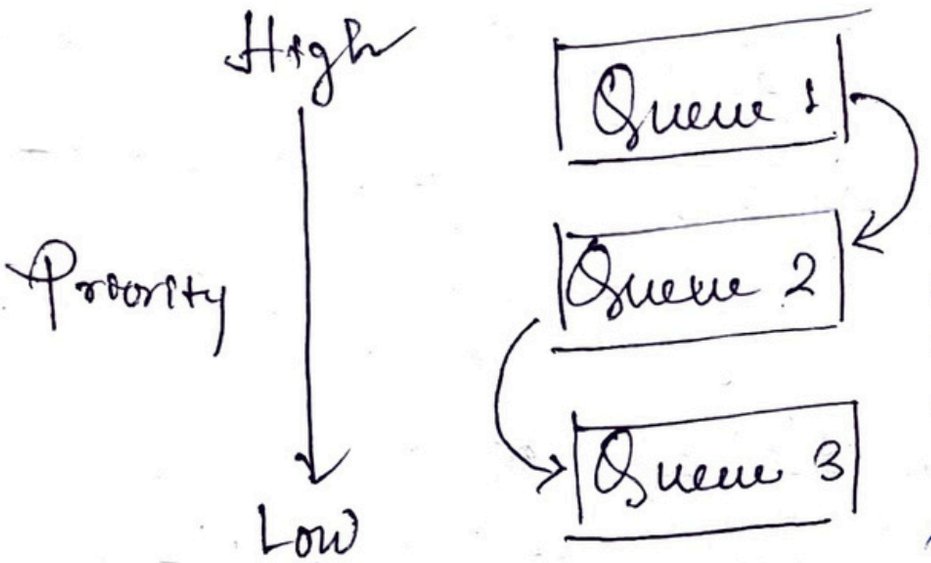
* Multilevel Feedback Queue Scheduling.

(MLFQ).

Process can move between queues.

MLFQ keeps analysing the behaviour of processes & according to which it changes its priority.

ref. gfg



If a process uses too much CPU time, it will be moved to a lower-prio queue. This scheme leaves I/O & interactive processes in the higher prio queues.

- MFQS defined by parameters -
  i) No. of queues
  ii) Sched. algo for each queue
  iii) Methods to determine when to upgrade/demote a process to higher/lower prio queue.

Process that waits for too long in a lower prio queue may be moved to a higher prio queue. This form of aging prevents starvation.

- Most general CPU scheduling algo.
- Can be configured to match a specific system under design.
- Most complex algo too.

* Starvation.

Indefinite blocking in which a process ready to run for CPU can wait indefinitely because of low priority (or some other reasons).

Solution to starvation:

Aging: Gradually increasing the priority of processes that wait in the system for a long time.

→ When deadlock occurs no process can make progress, while in starvation apart from the victim process other processes can progress or proceed.

* Thread :

A thread is a path of execution within a process. A process can contain multiple threads.

→ Multithreading

Parallelism by dividing process into threads.

→ Process vs Thread.

Threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are.

Like process, a thread has its own program counter, register set, & stack space.

Advantages of thread over process. ~

gfg

i) Responsiveness

ii) Faster context switch

iii) Effective utilisation of multiprocessor systems.

iv) Resource sharing — Economy

v) Communication.

vi) Enhanced throughput of system.

Types of threads:

User level thread,

Kernel level thread.

Similarity between threads & processes

Only one thread or process is active at a time. Within process both execute sequentially. Both can create children.

(quasi-parallel)

Differences between threads & processes

Threads are not independent, processes are, Threads are designed to assist each other, processes may or may not do it.

User level thread (ULT)

Implemented on the user level library, not created using sys. calls. Kernel doesn't know about the ULT & manages them as if they were single threaded processes.

Adv.: Can be implemented on an os that doesn't support multithreading.

Simple representation.

Simple to create.

Thread switching is fast.

Disadv.: No or less coordination among threads & kernel.

If one thread causes a page fault, the entire process blocks.

Kernel Level Thread (KLT).

Kernel knows & manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. OS kernel provides sys. call to create & manage threads.

Adv.: Since kernel has full knowledge about the threads, scheduler may decide to give more time to processes having large no. of threads.

Good for application that frequently block.

Disadv.: Slow & inefficient. It requires thread control block so it's an overhead.

* Microkernel.

The user services & kernel services are implemented in different address space. The user services are kept in user address space & kernel services in kernel address space, thus also reduces the size of kernel & size of OS as well.

μkernel is responsible for-

i) Inter process comm"
ii) Memory management
iii) CPU scheduling

Adv.: i) Architecture of this kernel is small & isolated hence it can function better.

ii) Expansion of system is easier, it's simply added in the system application without disturbing the kernel.

eg. Eclipse IDE.

*Layered Architecture - Groups components that perform similar functions into layers. Each layer communicated only with layers immediately above & below it. Processes' requests might pass through many layers before completion.

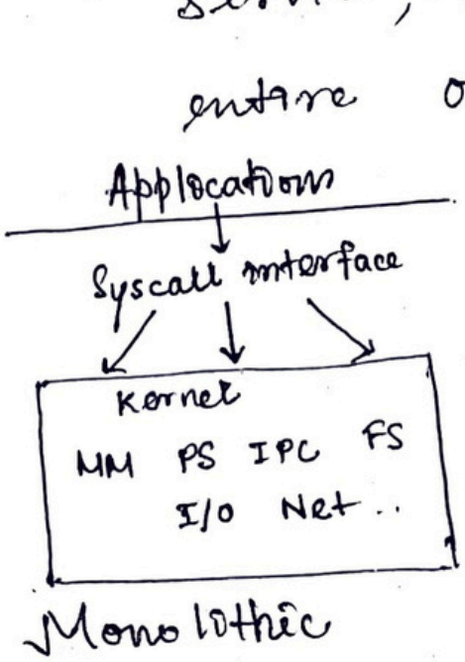
* Monolithic Kernel (Linux, BSD, Solaris)

Manages system resources between application & hardware, but user services & kernel services are implemented under same address space. It increases the size of the kernel, thus increases size of OS as well.

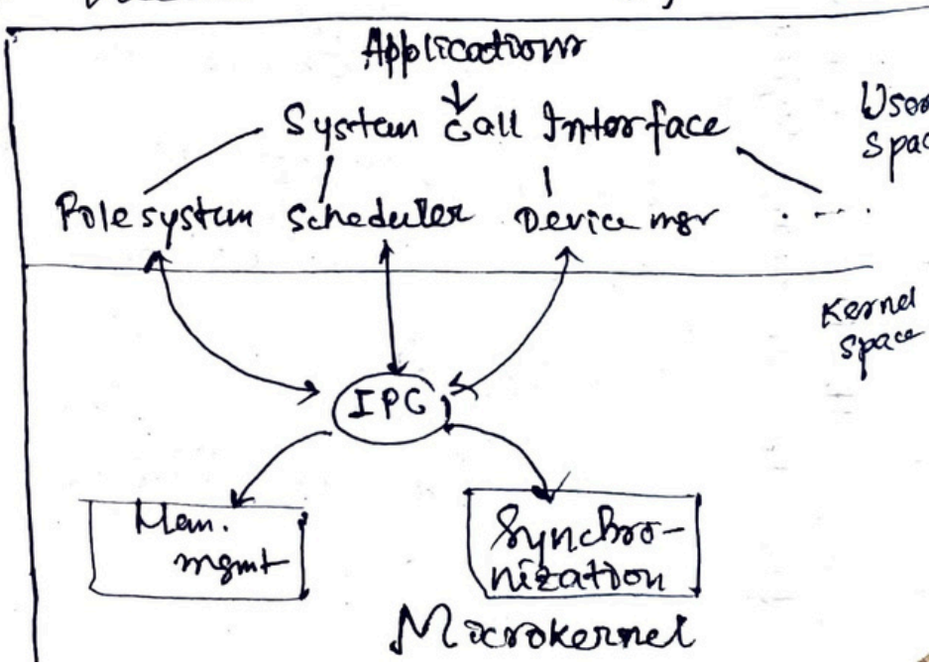
Adv.: It provides CPU scheduling, memory mgmt., file mgmt.; it's a single static binary file.

Disadv.: If any one service fails it leads to entire system failure.

If user has to add any new service, user needs to modify entire OS.



Monolithic



Microkernel

* fork() in C.

System call to create a new process (child process) & it runs concurrently with process (which called system call fork), parent process. After a new process (child) created, both processes will execute the next instn following the fork() sys. call. A child process uses the same PC, same CPU registers, same open files which use in the parent process. It takes no parameters & returns an integer value.

Return values by fork().

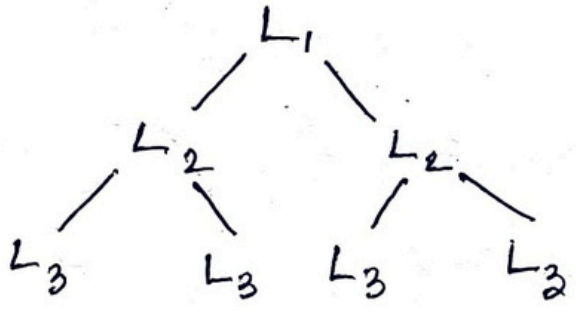
i) -ve value: Creation of a child process was unsuccessful.

ii) Zero: Returned to the newly created child process.

iii) +ve value: Returned to parent or caller. Value contains PID of newly created child process

Total no. of processes = 2^n where n is number of fork sys. calls.

```
fork();
fork();
fork();
```

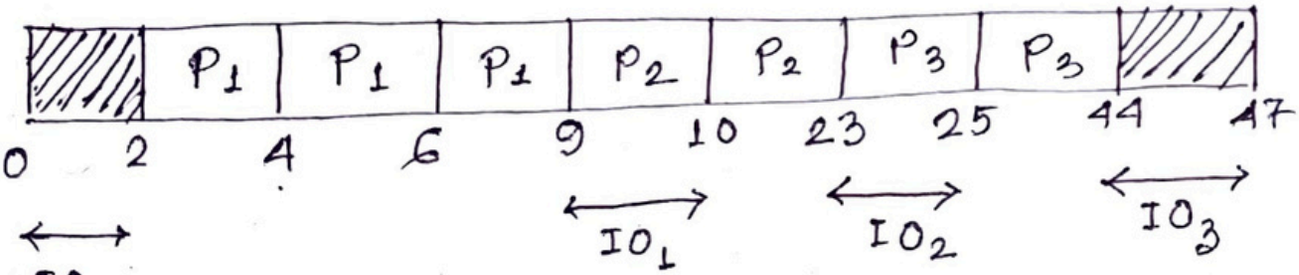


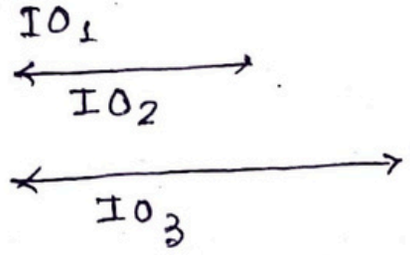
CPU scheduling problems.

eg. ***

PID	Total BT	I/O BT	CPU BT	I/O BT.
P_1	10	2	7	1
P_2	20	4	14	2
P_3	30	6	21	3.

SRTF.



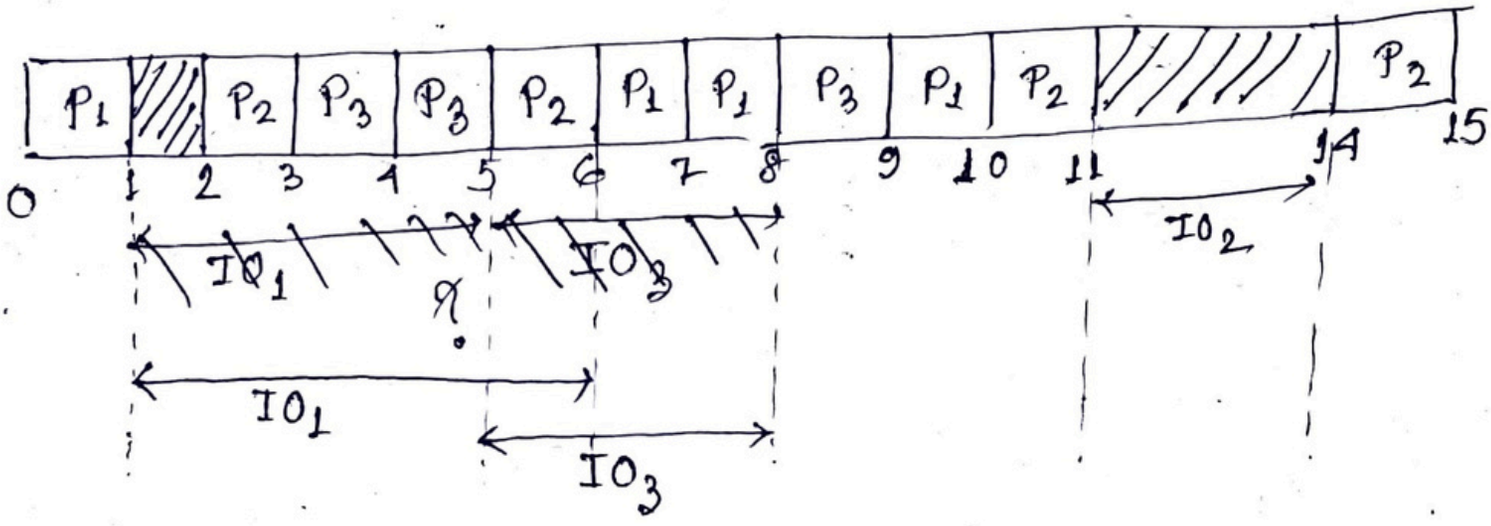


% idle CPU time = $\frac{5}{47} \times 100\%$

eg. *** Imp:

PID	AT	Prio	BT		
			CPU	I/O	CPU
1	0	2	1	5	3
2	2	3	3	3	1
3	3	1	2	3	1

Priority scheduling



Process Synchronisation.

* Process synchronisation controls the execution of processes running concurrently to ensure that consistent results are produced.

Need of sync.:
When multiple processes execute concurrently sharing some system resources.

* On the basis of synchronisation, processes are of 2 types:

i) Independent process. : Execution of one doesn't affect the exec^n of other.

ii) Cooperative process : Execution of one affects other's exec^n.

* Critical Section: Critical section is a section of the program code where a process access the shared resources during its exec^n. It can be accessed by only one process at a time. CS contains shared variables that need to be synchronised to maintain consistency of data variables. (CS needs to be executed atomically).