→ Web link :-

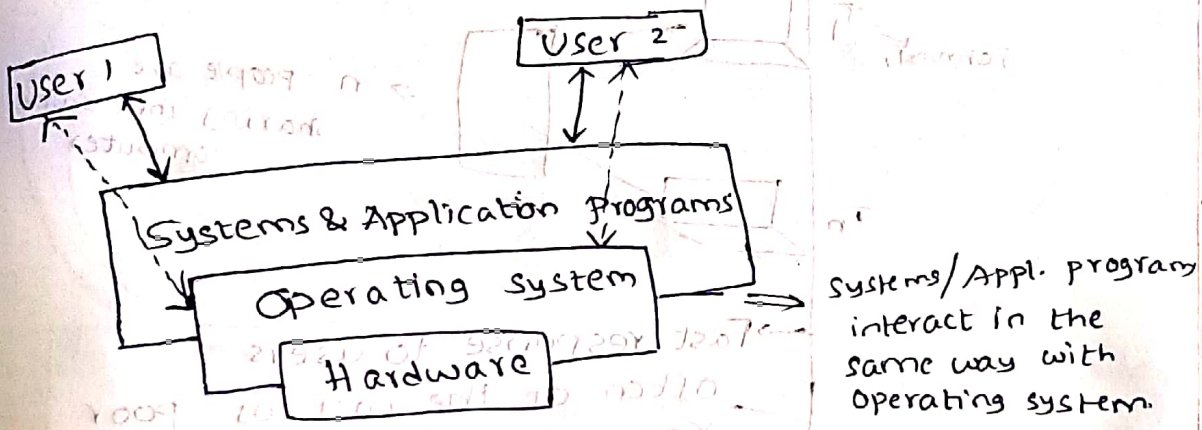
http://www.facweb.iitkgp.ac.in/~isg/os

→ Books:-

→ Operating System Concepts, 9th edition
- Silverschatz, Galvin and

* What is an OS?

→ A program that acts as an interface between user program and computer hardware.



→ Systems programs → programs that assist the user in creating/running programs

Ex: Compiler, loader/linker, editor...

→ Application programs

Ex: Calculator, word processor....

→ Primary goals of an OS

→ Make a computer system easy to use.

→ Ensures efficient utilization of the resources → CPU, Time, Memory, files, I/O devices

* Evolution :-

→ Mini Computers and Mainframes (1970-80)

↓ Ex: IBM 1130

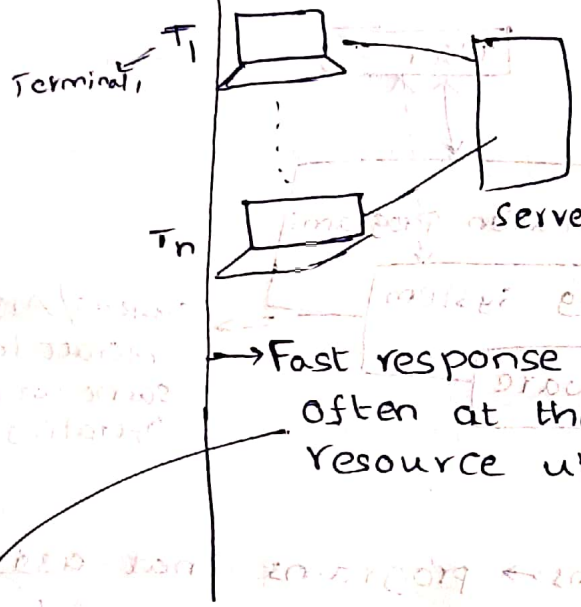
[Hallorith code → in Punched Cards]

→ Users used to share a computer

→ Best utilization of resources

→ User experience has compromised

→ Workstations and Terminals (1980 - present)



⇒ 'n' people are sharing the same computer

→ Fast response to users often at the cost of poor resource utilization

Ex: program P1 P2 P3

Read

↓

10 min. ↺ compute

↓

Print

→ while P_1 is in compute, if P_2 is called, then P_1 has to saved & stopped temporarily ⇒ more # registers are used.

→ Desktops (PCs) and Laptops (1985 - present)

→ Flexible user interface, easy to use

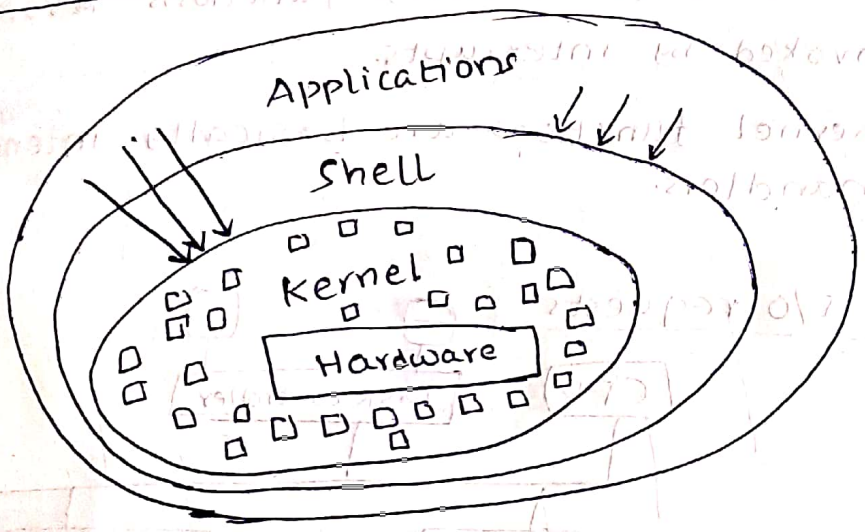
→ Single-user system (fully personal)

→ Mobile Systems

→ Constrained battery life

→ User experience

*A Typical structure of OS (w.r.t UNIX) terminology

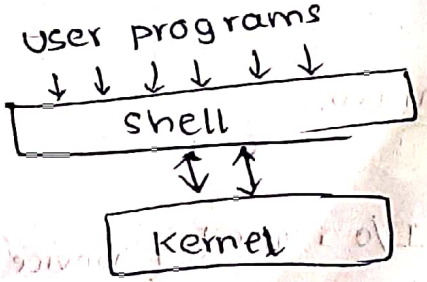


□ → function

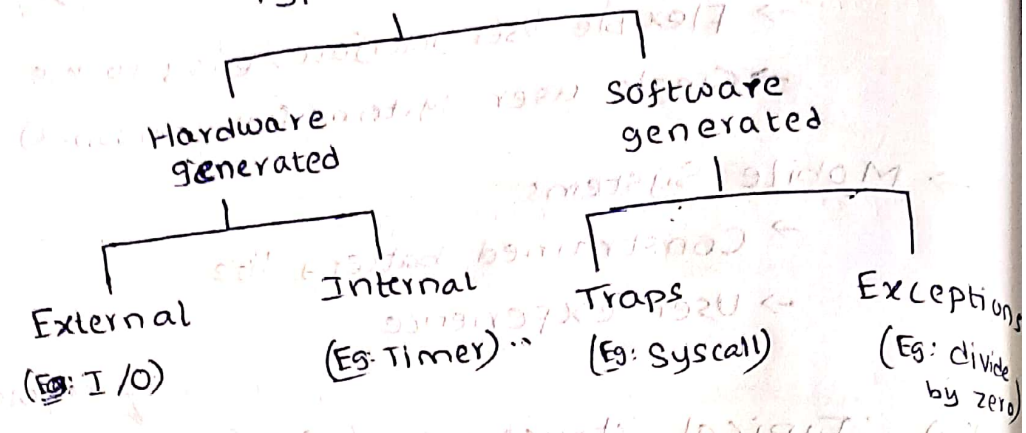
→ Kernel → A set of functions that provide the essential functionalities of the OS, and the associated data structures.

→ kernel runs at all times → INCORRECT Statement

→ Shell → A program which allows user programs to access the functions provided by the kernel.

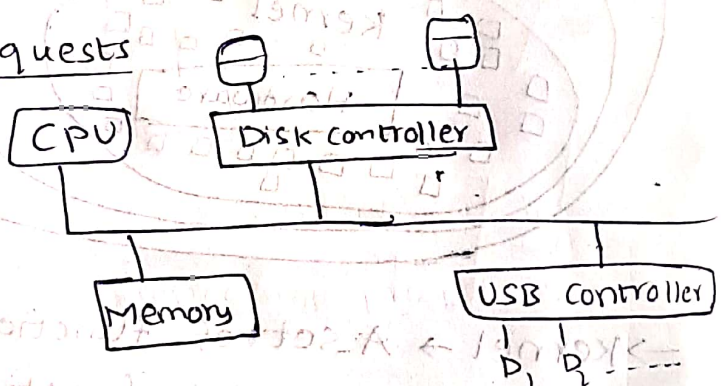


* Types of Interrupt



* How are the OS (i.e kernel) functions invoked?
  - → invoked by interrupts.
  - → Kernel functions are basically interrupt handlers.

* Handling I/O requests

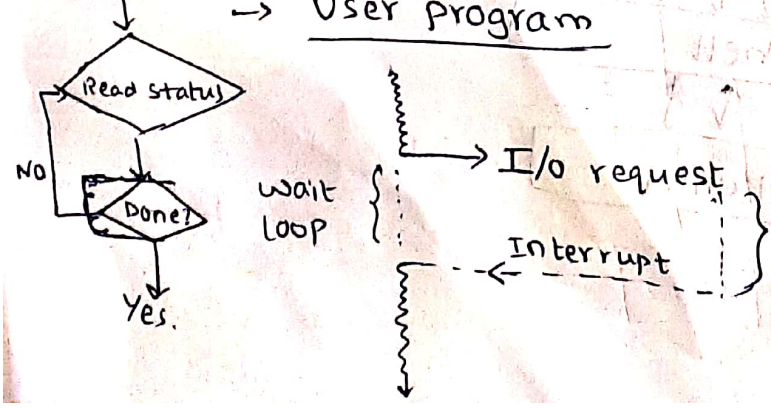


→ If there is no Disk controller, CPU has to spend so much time in disk access.

→ Disk controller → has a dedicated processor to carryout disk access and store the read data in a buffer.

* Alternate 1 :-

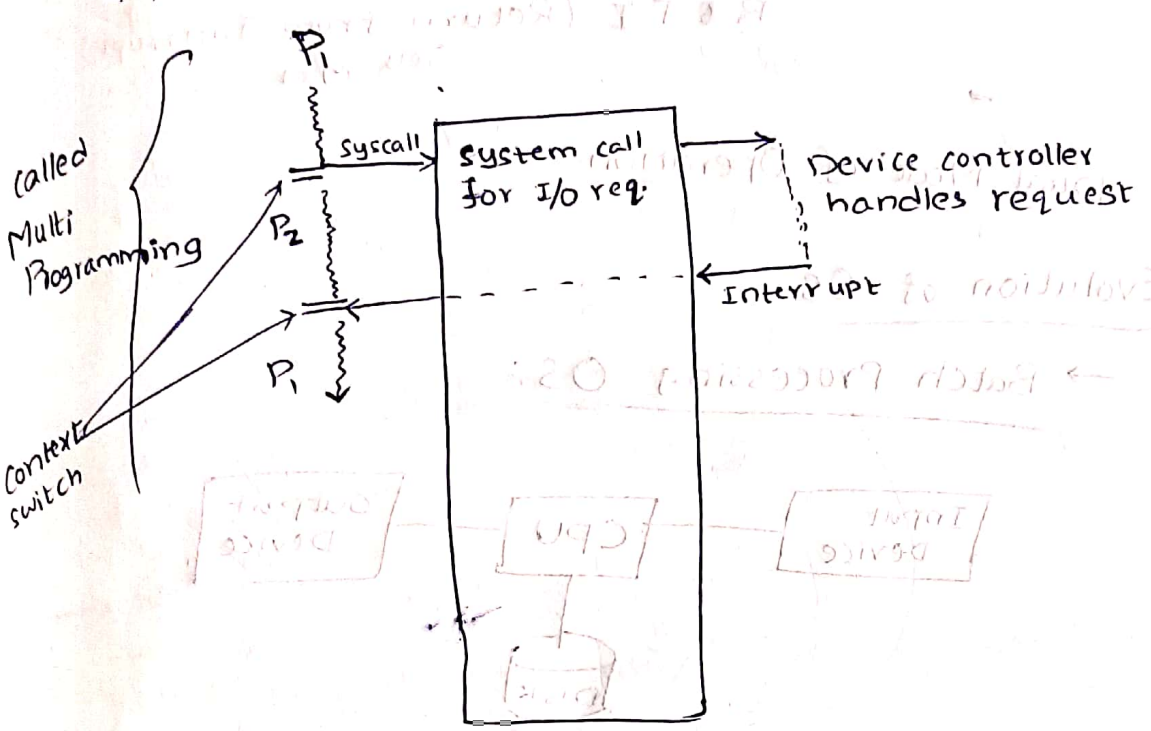
→ User program



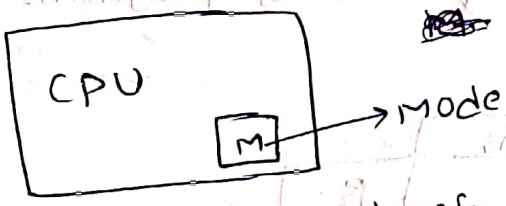
→ The above scheme is called "BUSY WAITING".

*Alternative 2 :-

→ Active programs P_1, P_2, P_3..

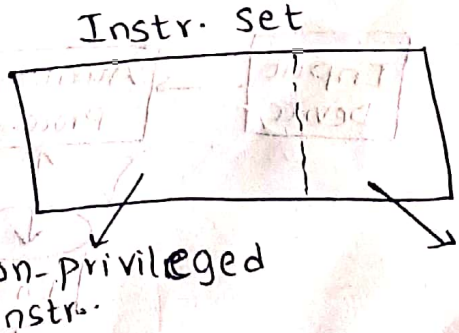


*Syscall



M=1 ⇒ User mode of execution

M=0 ⇒ Supervisor/ Kernel/ privilege Mode of Execution.



* → CPU can execute a privileged instr. only when M=0
  ↳ else it will generate an exception.

→ Syscall ⇒ Set M=0 · Interrupt Handler

Transfer control to IH (inside kernel)

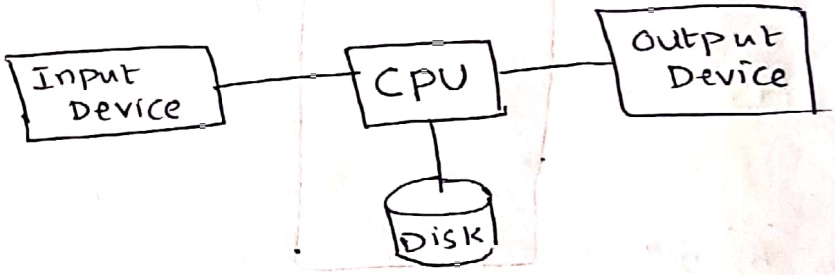
IH executes

RTI (Return from Interrupt) Sets M=1

Dual-Mode of Operation.

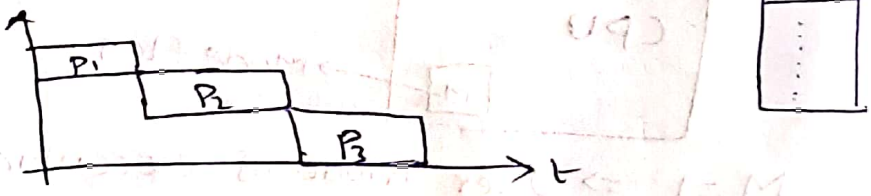
*Evolution of OS :-

→ Batch Processing OS :-

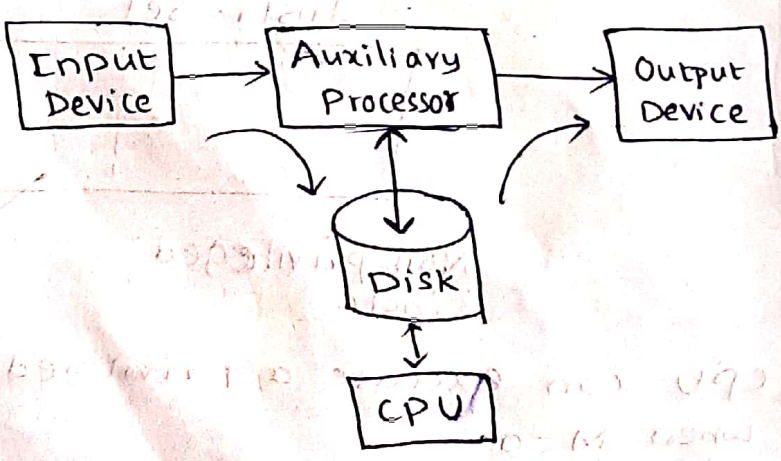


→ programs execute one at a time.

→ Batch : a sequence of programs

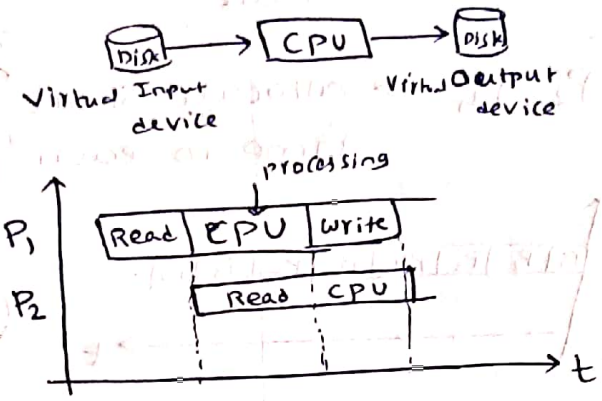


→ Simultaneous Peripheral Operations On Line (SPOOL)



→ Auxiliary processor
  → just reads data from input device & ~~puts it~~ sends to disk
  → (or) reads data from disk and sends to output device

→ Concept of virtual I/O devices



→ Multiprogrammed OS :-

→ Multiple programs loaded in memory at the same time.
→ You can switch execution from one program to another.
→ Parallelism in CPU & I/O operation.
→ concepts of CPU-bound jobs & IO-bound jobs
  ↙ programs which do more computation than I/O.

Ex: 1. Matrix multiply (n×n)
  I/O : $O(n^2)$
  computation : $O(n^3)$ ⇒ CPU-bound job
2. Payroll → IO-bound job

Objective ↓ Maximum resource utilization ↓ by a good mix of CPU & IO-bound jobs

→ Timesharing/Multi-tasking OS :-

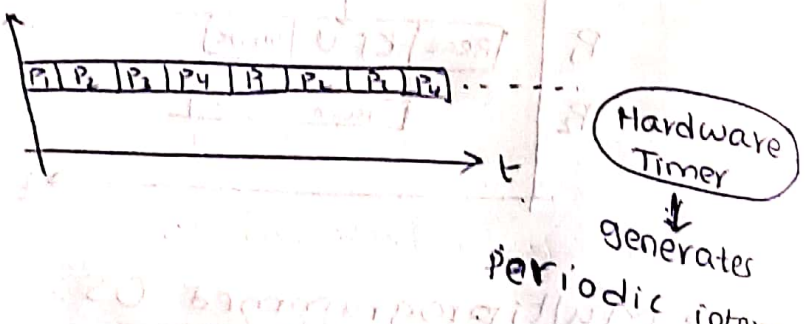
→ users sit on terminals & interact in real time.
→ Objective → Maximum user satisfaction ↓ minimization of response time

→ very frequent context switching between programs

→ Let's say there are 50 users

if we follow Multiprogramming, one job is being done & remaining 49 users have to wait.

→ say, P_1, P_2, P_3, P_4 → allocate short bursts (5-10ms) of time to each prog.



→ **Real-time OS :-**

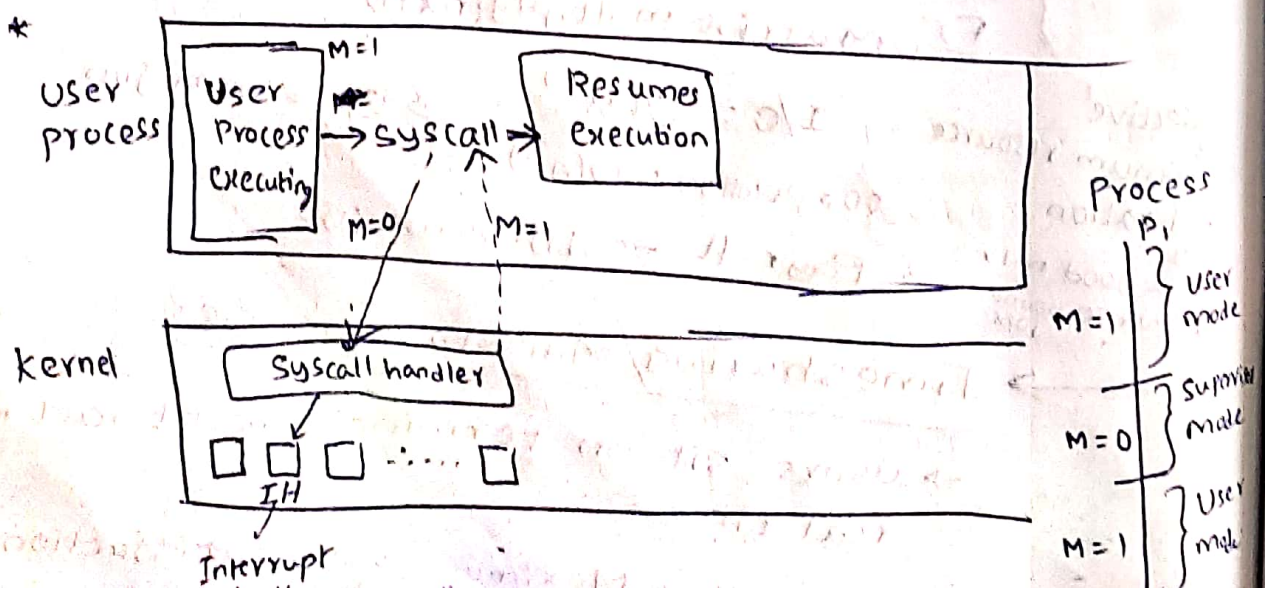
→ process - user requests based on deadlines
come as interrupts

→ **Distributed OS :-**

→ part of computation happens in one place and other part in other place.

* process → a program in execution

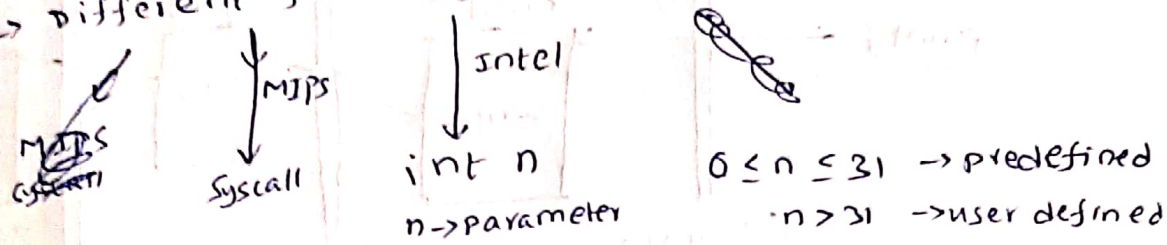
*



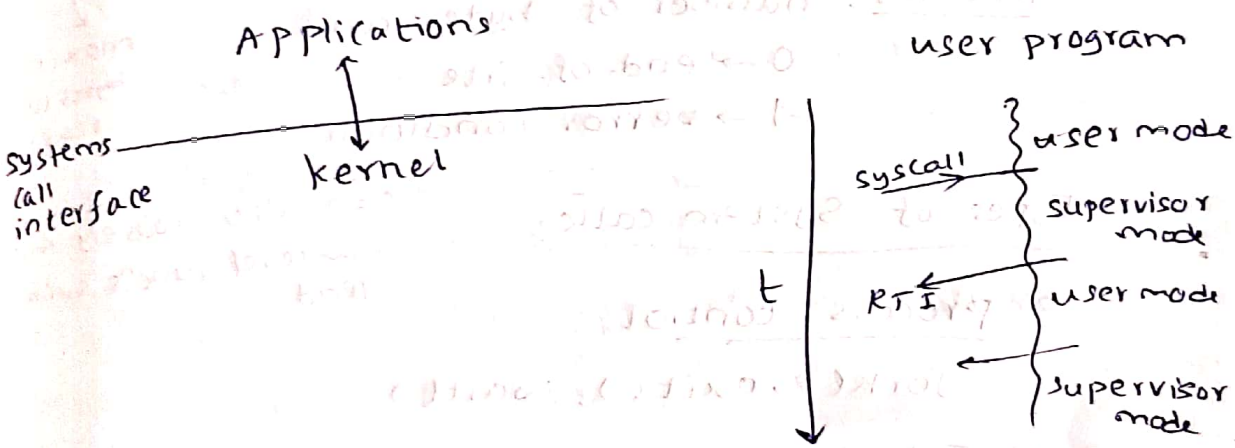
* System Calls

→ It is a machine instr.

→ different for different ISA.

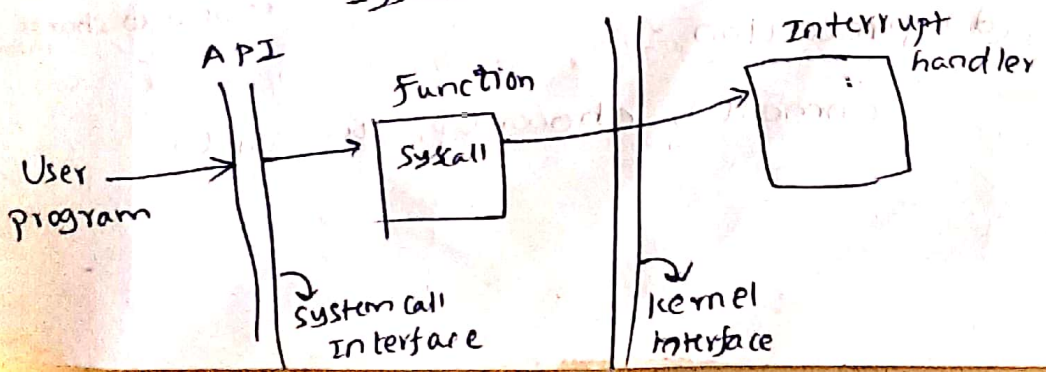


→ system call instr. is executed
- → software interrupt
- → Mode = supervisor
- → Invoke interrupt handler
- → Return from interrupt (Mode = user)

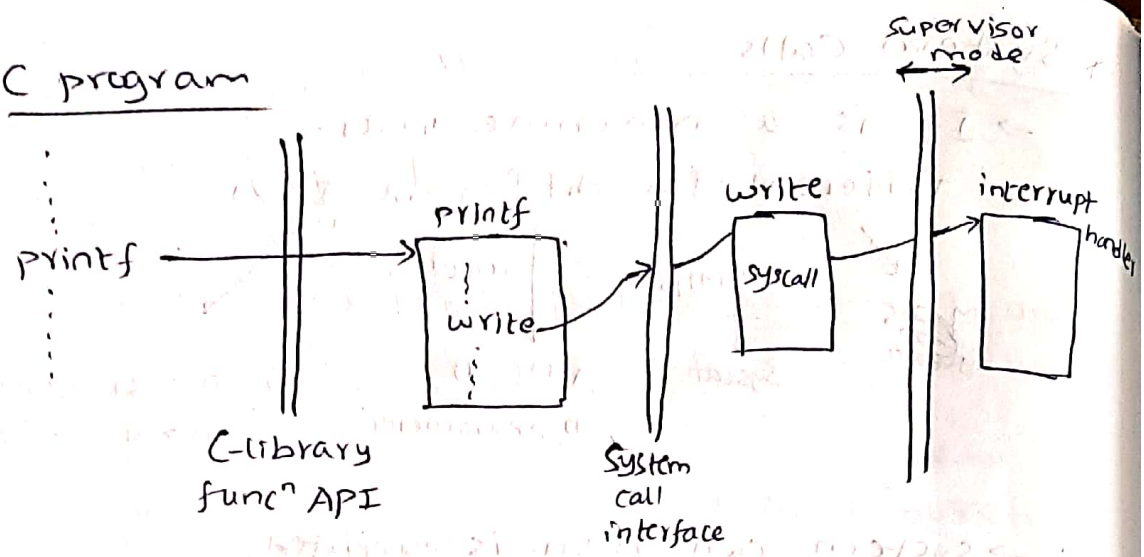


→ system call interface :-

→ provided by the OS
- → A set of low-level func^ns that can be called from a user program.
- → These low-level func^ns execute some system call.



* C program



* Example API :

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

max. no-of bytes to read

Returns : number of bytes read
0 → end-of-file
-1 → error condition.

→ Types of System calls :- (can also loosely mean low-level func^s such as read.)

(a) Process Control :-

fork(), exit(), wait().

(b) File manipulation :-

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

(c) Inter-process communication :-

Pipe(), semget(), semop(), semctl(), (related to semaphores)
shmget(), shmat(), shmdt(), shmctl(), (related to shared memory)
kill(), signal().

(pipe in terminal)

(d) Protection :-

chmod(), chown(), umask()

* Problems

1. Related to SPOOLing

data → card Reader → CPU → Printer

→ Payroll system (1000 employees)

→ Reading a card – 100 ms

→ Printing a line – 100 ms

→ Read/write from disk – 10 ms

→ processing time – 5 ms/employee

(a) No SPOOLing

elapsed time = 1000 (100 + 5 + 100) ms (only one line is to be printed)

= 205 s.

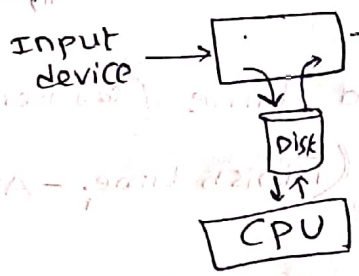
CPU idle time = 205 – 1000 × 5 ms

= 200 s.

card reader reads one card every (100 + 5 + 100) ms ≈ 5 cards/second

CPU utilisation = $\frac{5}{205} \times 100 = 2.5\%$

(b) SPOOLing



Elapsed time = 1000 (10 + 5 + 10) ms (10: reading from disk; 10: writing to disk)

= 25 s

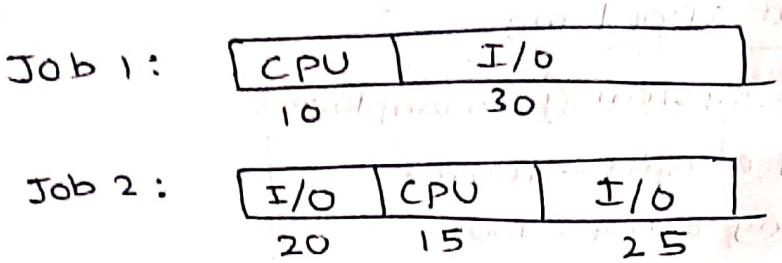
CPU idle time = 25 – 1000 × 5 = 20 s.

card reader reads one card every (100 + 10) ms ≈ 9 cards/second

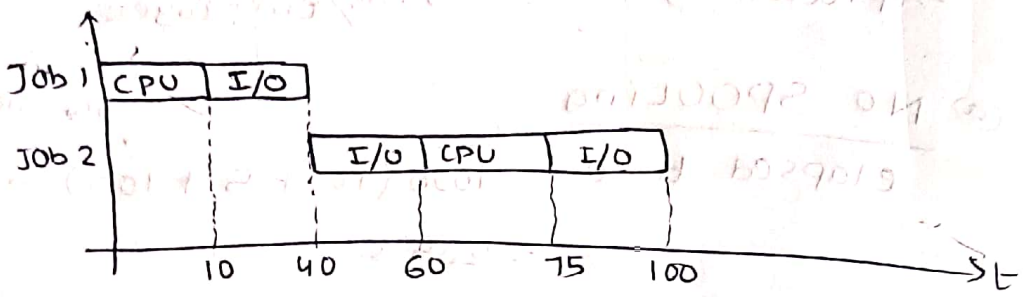
(100: reading from card; 10: writing to disk)

CPU utilisation = $\frac{5}{25} \times 100 = 20\%$

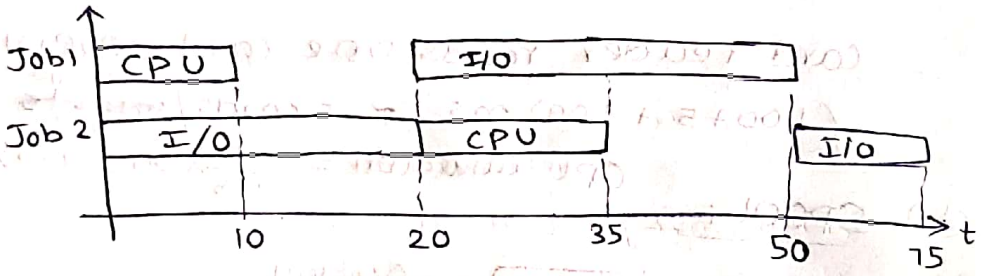
2. Related to multiprogramming :-



(a) Batch execution :-



(b) Multiprogramming :-



→ Average turnaround time (say there are 'n' jobs)

$$= \frac{1}{n} \sum_{i=1}^{n} (\text{Finish time}_i - \text{Arrival time}_i)$$

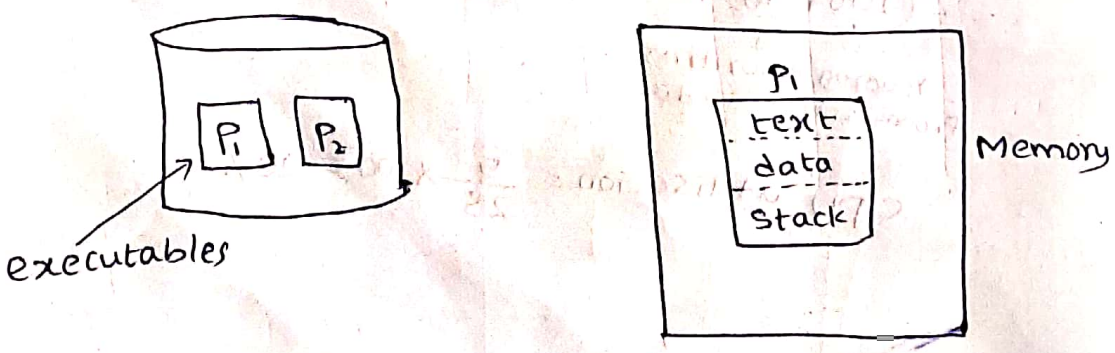
→ Average throughput

= No. of jobs completed per unit time

* Process :-

→ A program in execution

(terminologies of UNIX)



→ A process consists of several regions

→ text → contains executable code.

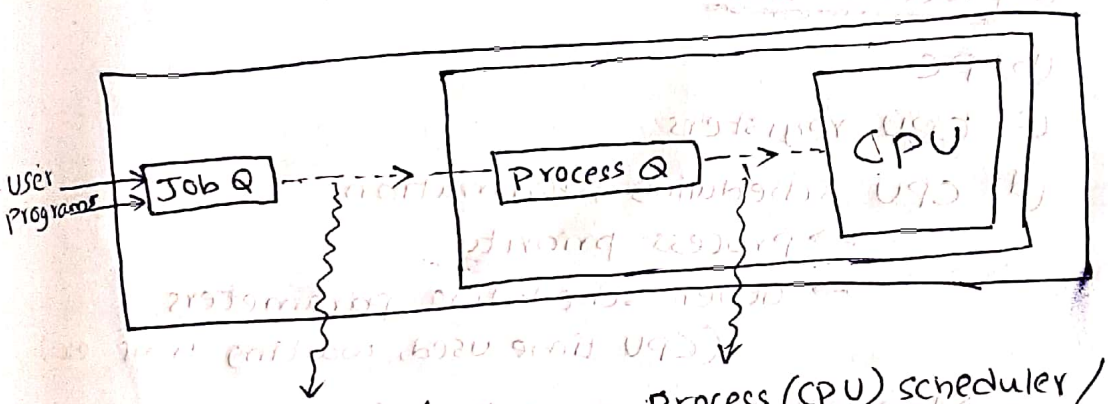
→ data → contains global data (static data)

→ stack → contains dynamic data (eg: recursion)

→ heap → dynamic data controlled by the program.
(eg. malloc).

→ shared → region shared by more than one process.

→ Batch-processing and multiprogramming
↳ user programs were called jobs



Job scheduler / long-term scheduler

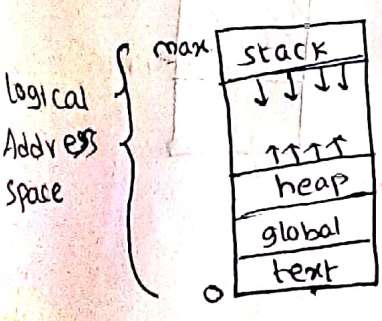
Process (CPU) scheduler / short-term scheduler
↓
decides which process should run now

When the Job gets loaded into memory, it becomes process.

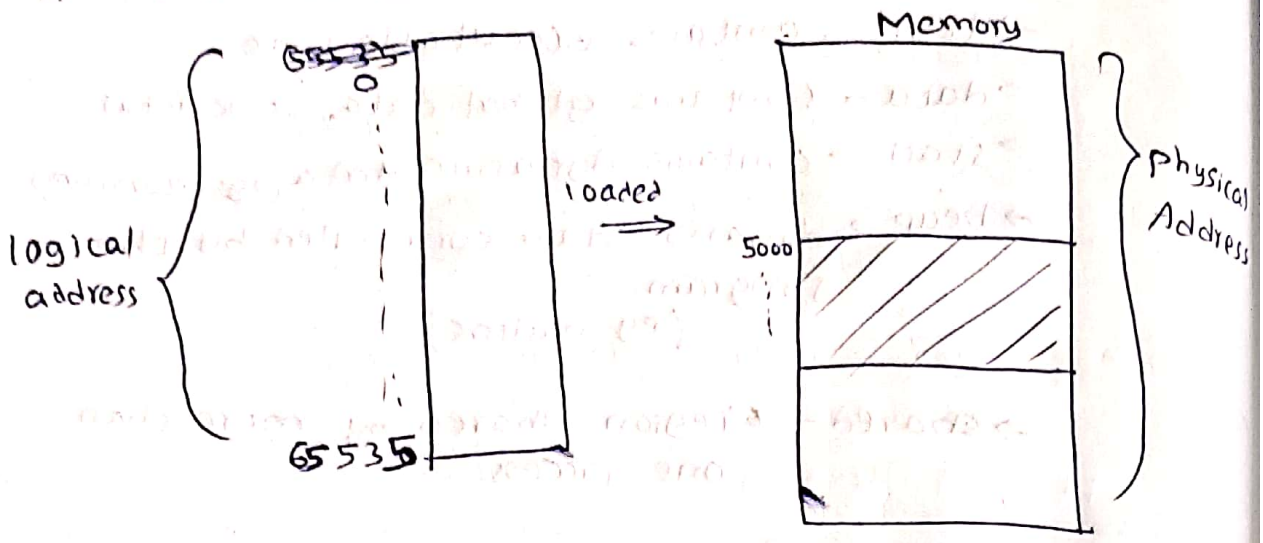
→ Structure of a process :-

→ Each process is represented in the kernel using a data structure called Process Control Block (PCB).

↓
Contains relevant information about a process.



* → Say Address - 16 bits



* Typical PCB entries:-

(a) process state

(b) PC

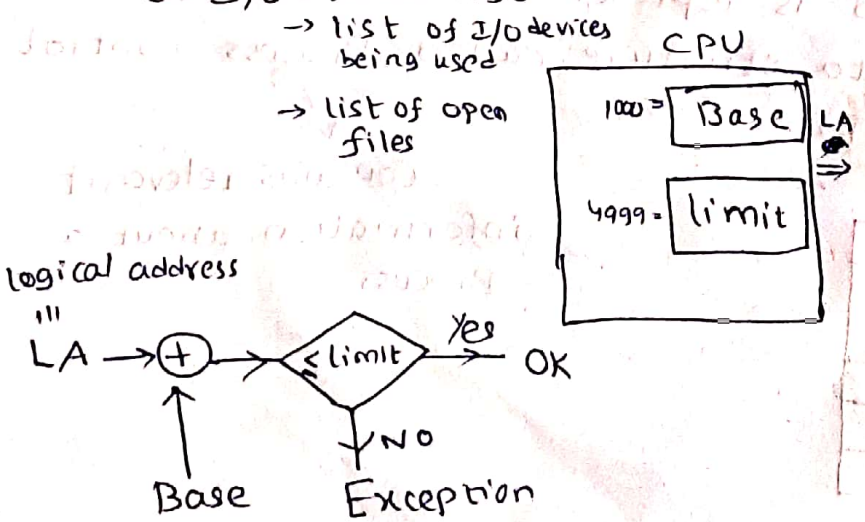
(c) CPU registers

(d) CPU scheduling information
- → process priority
- → other scheduling parameters (CPU time used, waiting time etc)

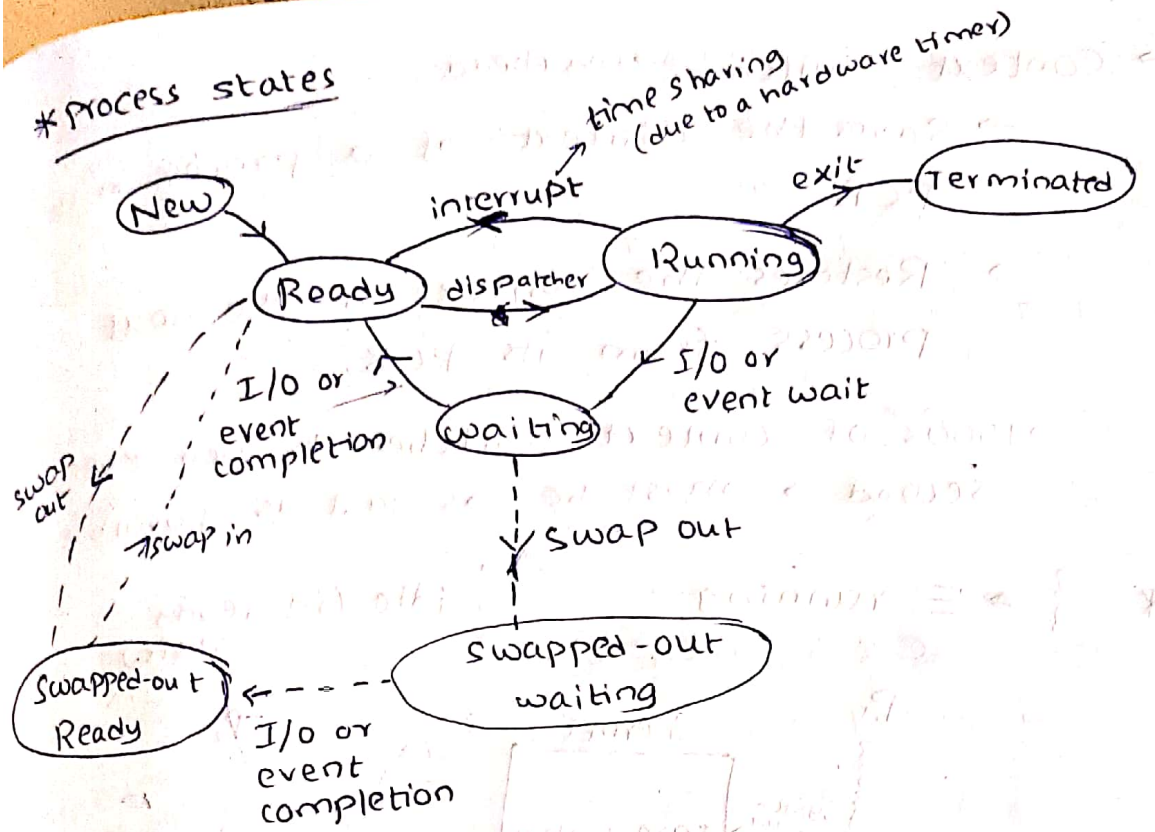
(e) Memory management information
- → Base and limit registers
- → Page table.

(f) Accounting information
- → Amount of CPU time used
- → Total waiting time
- → process id

(g) I/O status info
- → list of I/O devices being used
- → list of open files



*Process states



→ New : A process is being created

→ Ready : The process is ready to executed (waiting for CPU).

→ Running : The process is running on the CPU.

→ waiting } Blocked : The process is waiting for some event to occur. (I/O operation, signal from other process etc.).

→ Terminated : Finished execution and has lost the system.

* Memory is limited
  ⇒ No. of processes that can be accommodated is limited.

access to this is faster than the rest of the disk.

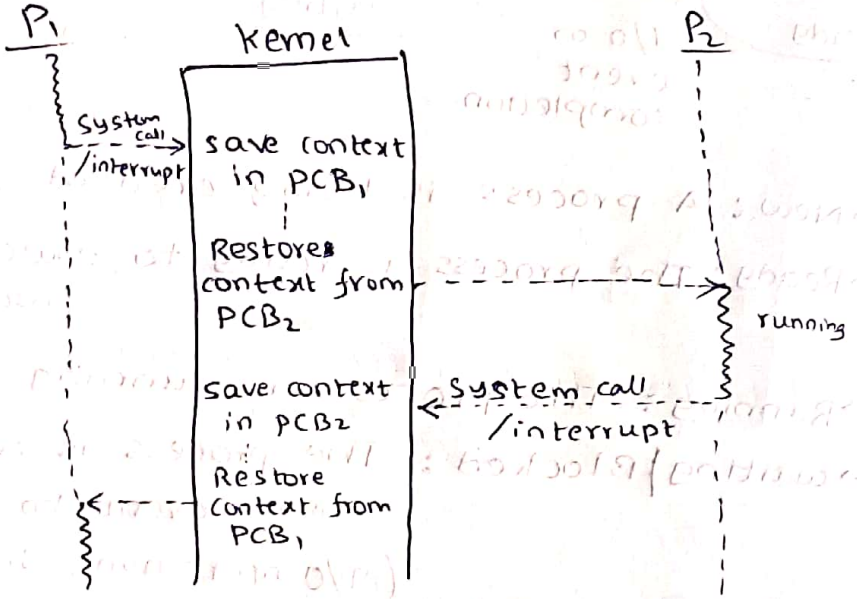
→ Context switch → overhead

→ Saves the context of a process in its PCB

→ Restores the context of the next process from its PCB.

→ 1000's of context switches happen every second ⇒ must be as fast as possible.

* } → ≡ running ¦ ≡ idle (in ready state)



P_1 & P_2 are concurrent processes

* Process scheduling :

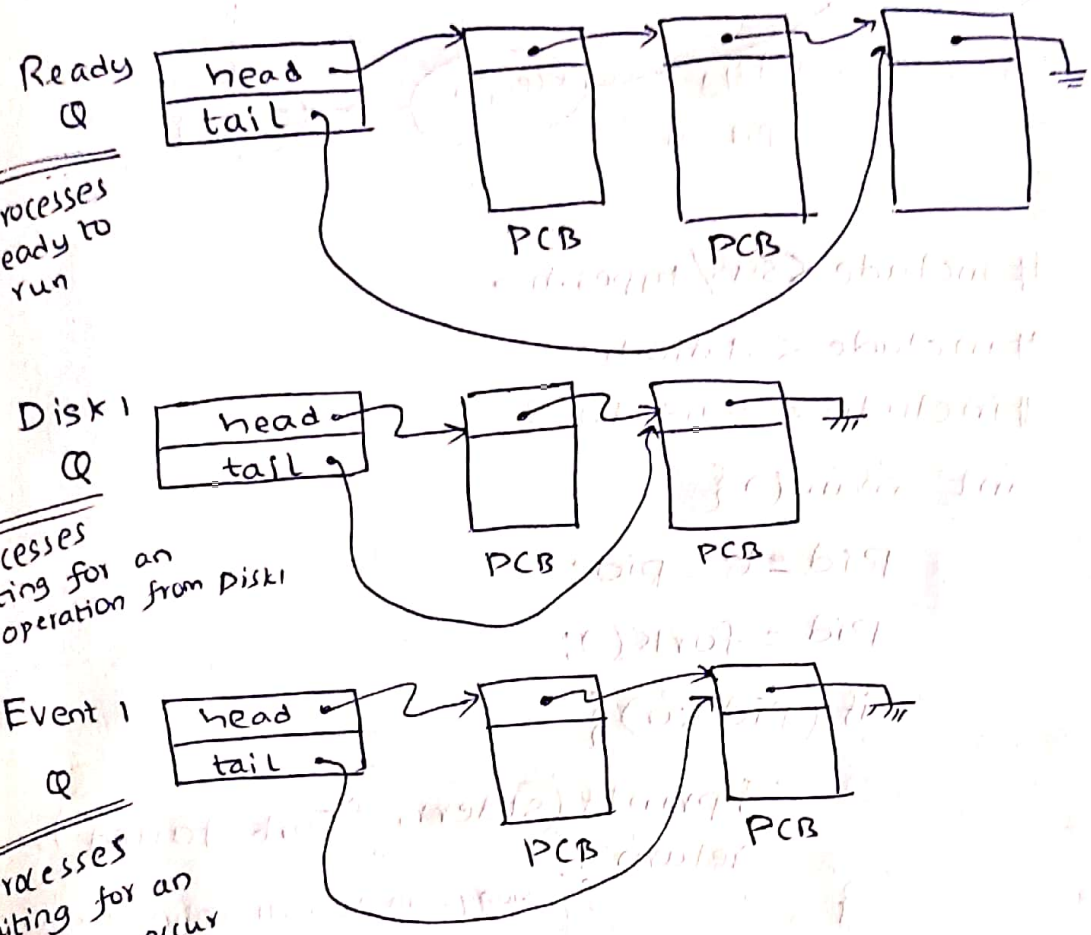
→ Ready state → queue of processes

when the running process is interrupted, there must be some algorithm to select a process from the queue to make it running

* Process scheduler or dispatcher selects a process from the ready queue for execution whenever there is a context switch.

* One of the many scheduling queues maintained by kernel.

→ linked list of PCBs



* Operations on Processes:

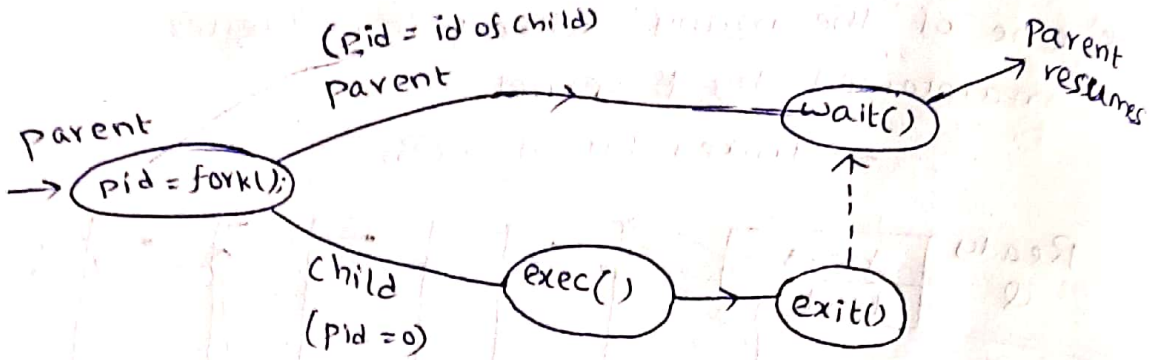
→ Process creation:-

→ 'ps' command lists all the processes active.

→ every process has a unique id (pid).

→ A single process that gets created in the beginning → called "init" with pid = 0
i.e when booted

→ all other processes are created using fork.




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    Pid_t pid;
    Pid = fork();
    if (Pid < 0){
        fprintf(stderr, "Fork failed");
        return 1;
    }
    else if (pid == 0){
        execlp("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

↳ non-zero return value ⇒ some error.

→ Process Termination:-

→ The exit() and wait() system calls can be used.

→ child process

```
exit(2);
```

Parent process

```
pid_t pid;
int status;
pid = wait(&status);
```

pid: id. of child who has terminated.
[∵ parent may fork many processes, it can find which process terminated]

status: Value passed as argument to exit() in child.
i.e, Here it is 2.

→ Normal process termination
- → child executes exit()
- → parent executes wait()

If both these happen, the OS considers that it happened successfully & all entries are deleted from process tables.

→ A child process terminates but parent has not called wait().
- ⇒ The child becomes zombie process
  - i.e, some entries will remain though it's not active

→ A child process terminates & parent also terminates but without calling wait.
- ⇒ The child process becomes Orphan process

* All orphans are assigned to "init" as default parent (i.e init is parent) & init calls wait() periodically.

→ ∃ multiple process in various states of execution

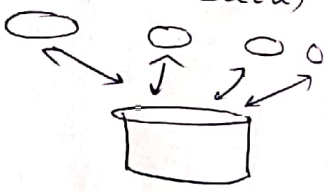
↳ concurrent process

independent

A process that doesn't affect or be affected by other process running in system (can't share any data with other processes).

co-operative

Process that are not dependent. (share some data)

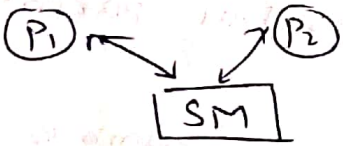


* **Co-operating Processes :-**

→ Some mechanism using which the processes can exchange data.

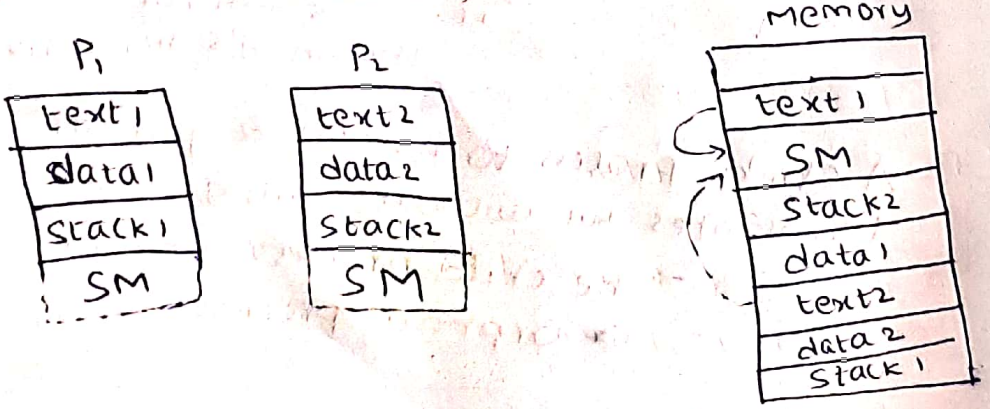
Inter Process Communication (IPC)

Shared Memory (SM)



Message passing

(a) **Shared Memory (SM)**



→ System calls:

→ shmget ⇒ creates a SM segment in Memory

→ shmat ⇒ attaches the SM segment to the address space of a process

→ shmdt ⇒ detaches the SM segment from a process.

→ shmctl ⇒ remove the SM segment from memory

*
```
int pid;
char * myseg;
key_t key; int shmid;
key = 758;
shmid = shmget(key, 250, IPC_CREAT|0666);
pid = fork();
if (pid == 0){
    myseg = shmat(shmid, NULL, 0);
    shmdt(shmid);
}
else{
    myseg = shmat(shmid, NULL, 0);
    shmdt(shmid);
}
shmctl(shmid, IPC_RMID, NULL);
```

no. of bytes

Create a new SM segment with key if it does not exist

access permissions (octal)

666 ≡ 110 110 110 ≡ rwx rwx rwx

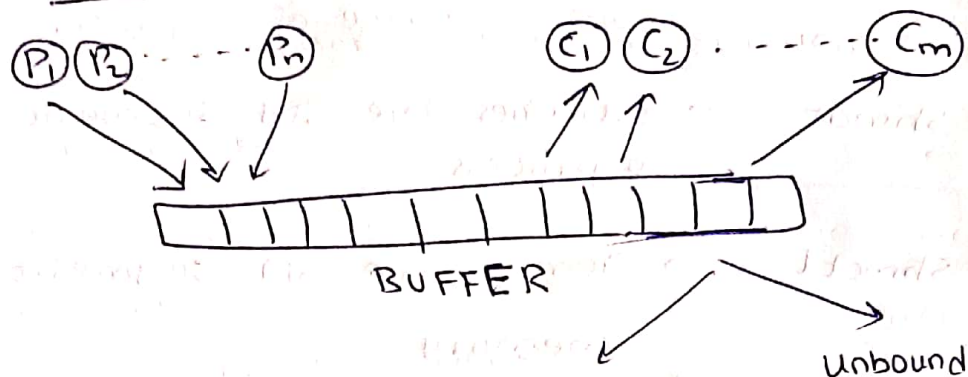
for user — for group — other

every user has a group id also

* Example:-

Producer - Consumer Problem:

→ Producers



Bounded Buffer

Unbounded Buffer

If P_i sees that the Buffer is full, then P_i has to wait until next write.

* Pseudo code:

```
#define BUFF_SIZE 50
typedef struct {
        int value;
}item;
item buffer[BUFF_SIZE];
int in=0;
int out=0;
```

out ≡ pointer to the first data item in the buffer

in ≡ pointer to next free Location in Buffer

→ located in Shared memory

Producer

```
while(true){
    <produce an item in next_p>
    while(((in+1)% BUFF_SIZE) == out);
    buffer[in] = next_p;
    in = (in+1)% BUFF_SIZE;
}
```

Consumer

```
item next_c;
while(true) {
    while(in == out);
    next_c = buffer[out];
    out = (out + 1) % BUFF_SIZE;
    <consume item in next_c>
}
```

⇓

Max. data items that can be put in Buffer

~~= BUFF_SIZE~~

= BUFF_SIZE - 1

(b) Message - Passing :-

send(message)

receive(message)

Example :

2 processes trying to update a variable is SM.

SM [a = 100]

P_1	P_2
a = a + 1	a = a - 1

	P_1	P_2
in low level lang.	1. LOAD R_1, a	4. LOAD R_2, a
	2. ADD R_1, R_1, 1	5. SUB R_2, R_2, 1
	3. STORE R_1, a	6. STORE R_2, a

let's say St.1 was executing & there's a Context switch

The sequence is like say

1 4 5 6 2 3 ⇒ Final value of a = 101

for 4 5 1 2 3 6 ⇒ " " " = 99

The above is called RACE condition.

↓

also happens when I/I is given to S-R Latch.

Example :- Producer - Consumer problem

Producer

```
message next_P;
while(true) {
    <produce item in nextP>
    send (next_P);
}
```

Consumer

```
message next_C;
while(true){
    receive(next_C);
    <consume item
        in next_C>
}
```

→ Two approaches :

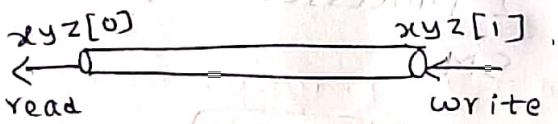
(a) unnamed pipes

(b) Named pipes/message Q.

* **Unnamed pipe :-**

→ pipe() system call.

```
int xyz[2];
pipe(xyz);
```




```
pid = fork();
if (pid == 0) {
    write (xyz[1], .....)
    :
}
else {
    read (xyz[0], ...)
    :
}
```

→ Parent & child can communicate since they have access to both ends of the pipe.

→ unnamed pipe can be used by processes that are generated by a common parent.

* Named Pipe:-

→ They use a common temporary file.

```
char *myfifo = "/tmp/myfifo";   → files in /tmp/ are accessible to every user

mkfifo(myfifo, 0666);
```

more

parent
```
fd = open(myfifo, O_WRONLY);
:
write(fd, ....);
close(fd);
```

child
```
fd1 = open(myfifo, O_RDONLY);
:
read(fd1, ....);
:
close(fd1);
```

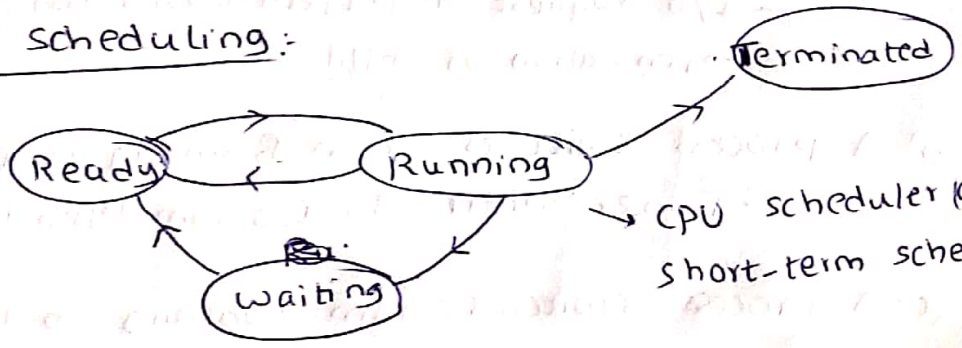
→ If even one of the processes calls mkfifo, then all the other processes can use it as named pipe.

(c) Process Comm^n across machines

→ Remote procedure call

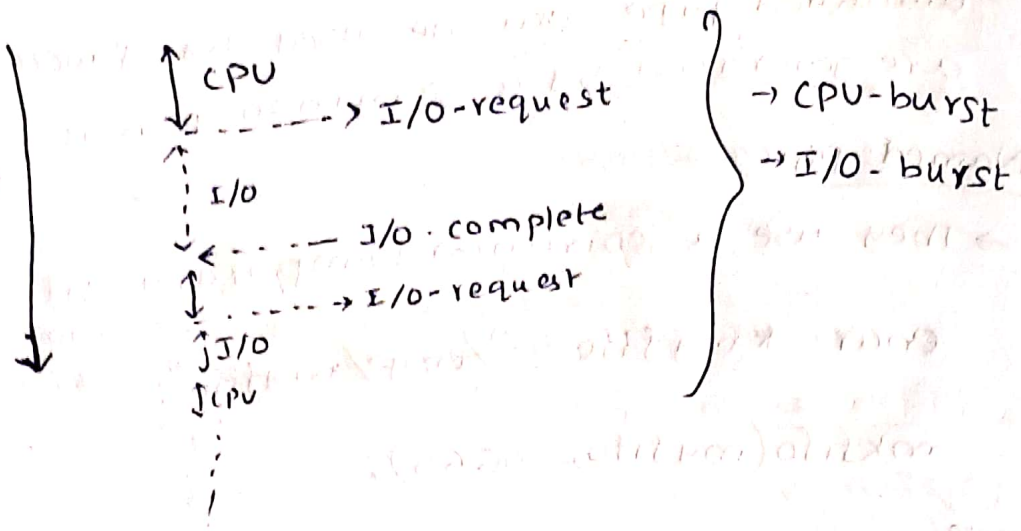
→ Socket.

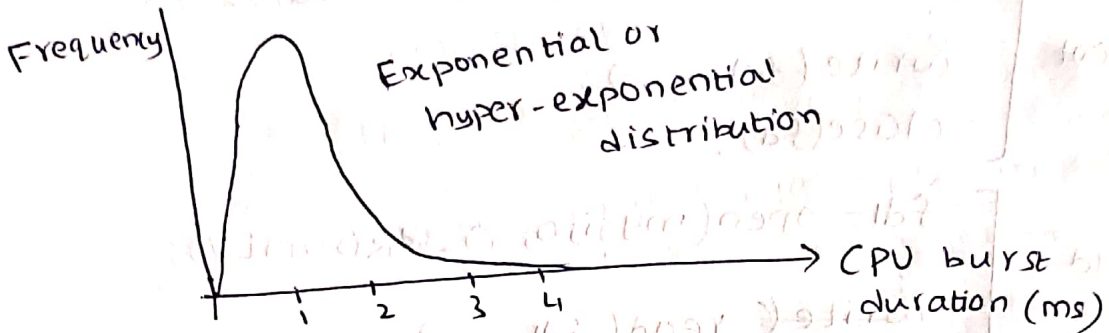
* CPU scheduling:-



→ CPU scheduler (or) short-term scheduler.

* Typical process execution:-





Typical curve

There are large no. of CPU bursts with very small duration.
& Small no. of CPU bursts with large duration.

* Scheduling decisions:

(a) A process switches from running to waiting
- I/O request (or) Parent invokes wait() for termination of child.

(b) A process switches from Running to Ready.
- Timer interrupt, I/O-completion interrupt

(c) A process switches from waiting to Ready.
- I/O completion interrupt, child has terminated.
→ then, Parent switches to ready.

(d) A process terminates
- exit() system call.

→ (a) and (d)

→ Non-Preemptive scheduling

↳ Allowing a process to continue for as long as it wants.

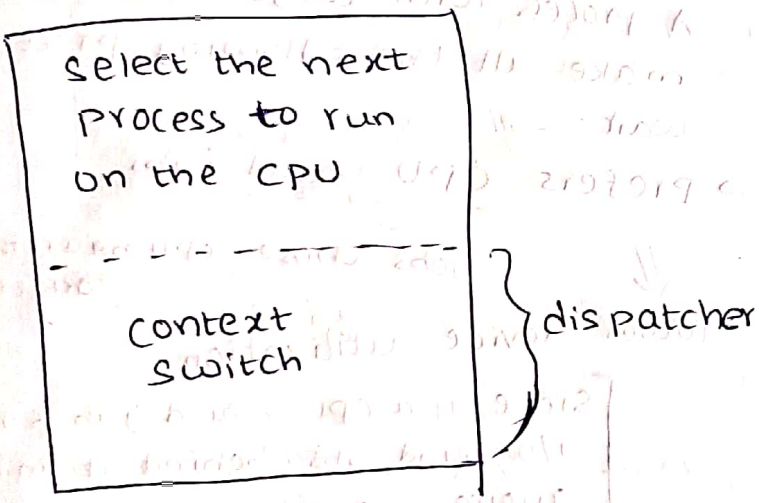
Here, the processes are coming out of CPU on their own.

→ (b) & (c)

→ Preemptive scheduling

↳ forcibly taking the CPU away from a process.

* CPU Scheduler



* Class Test 1

7th February 7-8 PM

Venue:- CSE Dept

Syllabus:- Up to CPU scheduling

* Scheduling Criteria

→ (a) CPU utilization

→ (b) Throughput – No. of processes finishing per unit time.

→ (c) Turnaround time – (Finishing time – Arrival time) ↑ when job first arrives.

→ (d) Waiting time – Time taken in the ready and waiting states.

→(e) Response time – After how much time a response is received

* CPU Scheduling Algorithms

→ (a) First-Come First Serve (FCFS)

→ It is a non-preemptive algorithm.

→ Ready list is maintained as a FIFO queue.

*** Drawback:**

→ Convoy effect

↓

Traffic has to wait when convoy passes

⇓

A process with large CPU burst will make all the following processes wait.

→ prefers CPU-bound jobs

↙

Jobs whose CPU-bursts are longer.

⇓

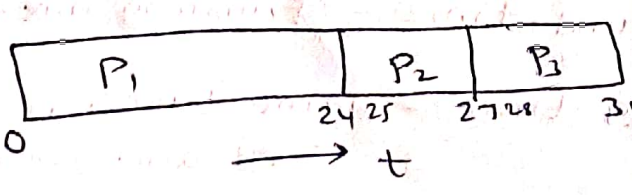
lower device utilization.

[Since, if a CPU-bound job is running, I/O-bound jobs behind it in the queue are not even started, then I/O-devices will lie idle]

Ex1:-

Processes	P_1	P_2	P_3
Arrival time	0	0	0
CPU Burst	24ms	3ms	3ms

(a) $P_1 - P_2 - P_3$



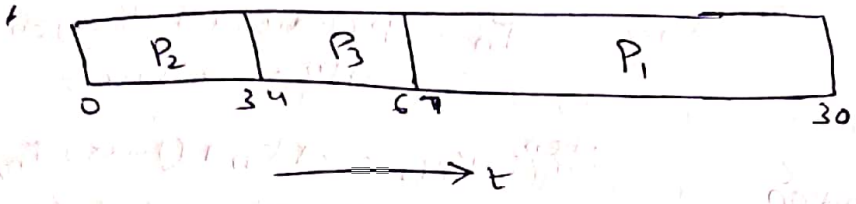
Avg. turnaround time = $\frac{(24-0)+(27-0)+(30-0)}{3}$

= 27 ms

Avg. waiting time = $\frac{0+(27-3)+(30-3)}{3}$

= 17 ms

(b) $P_2 - P_3 - P_1$



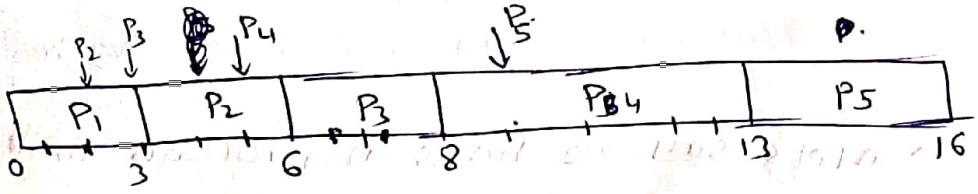
Avg. turnaround time = $\frac{(30-0)+(6-0)+(3-0)}{3}$

= 13 ms

AWT = $\frac{(30-24)+(3-3)+(6-3)}{3}$ = 3 ms

Ex2:

Process	P_1	P_2	P_3	P_4	P_5
Arrival time (in ms)	0	2	3	5	9
CPU Burst	3	3	2	5	3



AWT = $\frac{0+1+3+3+4}{5}$ = 2.2 ms

(b) Shortest Job First/Next (SJF/SJN)

→ The process with the smallest next CPU burst is allocated the CPU.
  ↳ non-preemptive.

→ Predicting the next CPU burst time
  → exponential averaging
    t_n – length of n^{th} CPU burst
    τ_n – predicted length of n^{th} CPU burst

based on history

then, $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$,
where $0 \le \alpha \le 1$

$\Rightarrow \tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2}$
$+ \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots$
$\cdots + (1-\alpha)^{n+1} \tau_0$ → initial value

$\alpha = 0 \Rightarrow \tau_{n+1} = \tau_n = \tau_0$
⇒ recent history doesn't have any effect.

$\alpha = 1 \Rightarrow \tau_{n+1} = t_n$
⇒ only recent CPU burst matters

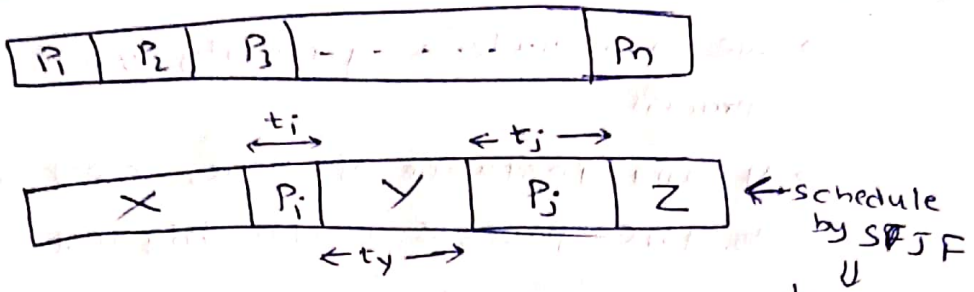
→ The SJF algorithm minimizes the AWT
(AWT ✓ Avg. Waiting Time)

→ Let's say we have 'n' processes that all arrive at $t = 0$.
CPU burst times are $t_1 \le t_2 \le \cdots \le t_n$.
$P_1 \quad P_2 \cdots P_n$

Proof that in this case, SJF minimizes AWT. :

Proof by contradiction.

→ SJF Schedule



Exchange P_i & P_j

⇒ X : no change in waiting time

Z : no change in waiting time

Y : All the processes ~~with~~ have their waiting times increased.

P_j : waiting time decreases by $t_y + t_i$

P_j : waiting time increases by t_y.

(c) Shortest Remaining Time First (SRT) :-

→ preemptive version of SJN

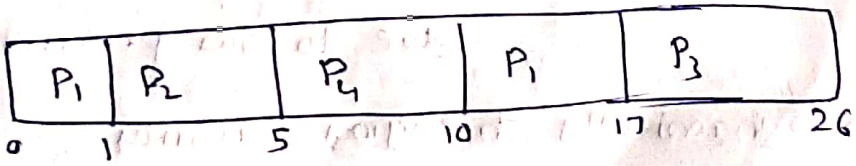
↳ A process with a shorter CPU burst can preempt a running process

→ we decide when

↳ A new process arrives

Ex)

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

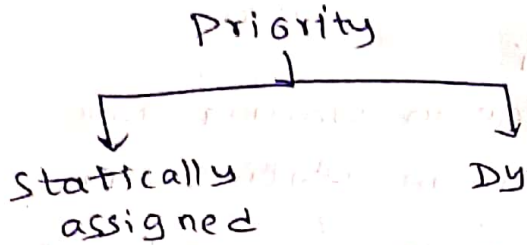


$$AWT = \frac{9+0+15+2}{4} = 6.5$$

(d) Priority Scheduling

→ We associate a priority value with each process

→ At any instance of time, CPU is assigned to the process with highest priority



→ Static priority

→ Many versions of linux

→ Each process is assigned a priority value

−20 ⇒ highest priority

+19 ⇒ lowest priority

↓

Default priority = 10.

→ Commands

→ nice ⇒ decreases the priority of a process

→ renice ⇒ increases " "

(only with "root" access)

→ Problem

Starvation or indefinite waiting

↓

A process is waiting indefinitely due to low priority

→ Dynamically changing priority

→ Try to overcome starvation problem.

↓

As a process waits, its priority increases

→ Response Ratio,

Jobs arriving first, but are being delayed have high RR.

$$RR = \frac{\text{Time since arrival} + \text{CPU burst time}}{\text{CPU burst time.}}$$

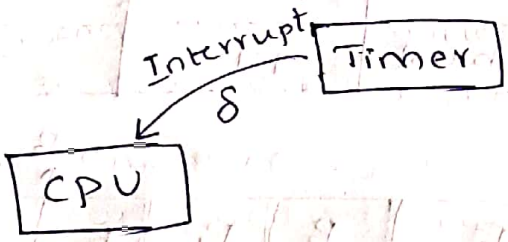
→ RR can indicate priority

→ whenever there's a clash of priority, we can choose the one with highest RR

(e) Round-robin scheduling :

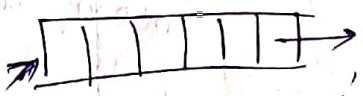
→ Designed primarily for interactive time-stationary systems

→ let's say a small time quantum, δ is predefined ($\delta \approx 1-100$ ms).

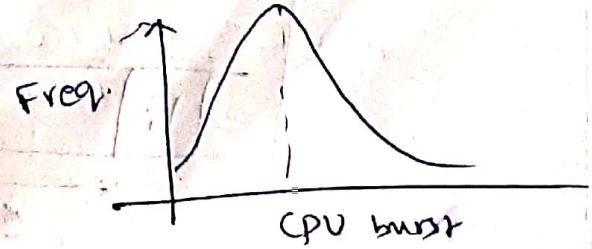


→ OS loads the timer
→ Decrements with a clock
→ Interrupts when I/O.

Ready list
↳ stored as circular queue.



→ The instruction to set the timer is priveleged instruction.



CPU burst

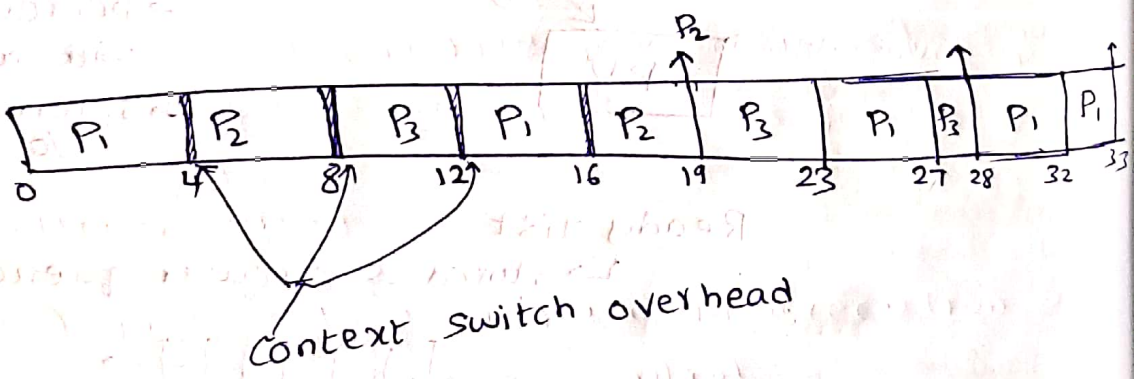
$< \delta$: process invokes a system call or an external interrupt arrives (say I/O)
⇓
process may go to waiting or ready state respectively
(i.e process releases the CPU.)
⇓
Context switch

$> \delta$: Timer interrupt
⇒ process goes to ready state
⇓
Context switch

→ Example :-

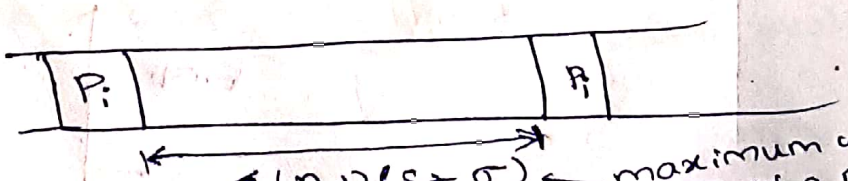
$\delta = 4$ ms

Process	P_1	P_2	P_3
Arrival Time	0	0	0
CPU burst	17	7	9



→ A Simple Analysis

→ n processes in READY queue.
→ δ is the time quantum.
→ σ is the context switch overhead.



$(n-1)(\delta + \sigma)$ ← maximum waiting time of a process before it gets back to CPU.

Very high δ ⇒ similar to FCFS.

Very small δ ⇒ large no. of context switches ⇒ overhead keeps adding up.

δ >> σ

δ ✓ typically 10 ms

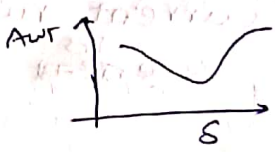
σ ↓ typically 10 μs.

Q.

Process	P_1	P_2	P_3	P_4
Arrival time	0	0	0	0
CPU Burst	6	3	1	7

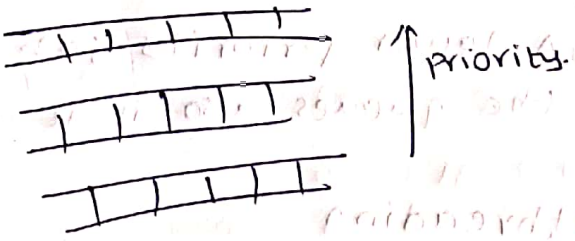
Compute AWT for $\delta = 1, 2, 3, 4, 5, 6, 7$.

Ans. we get graph like



(f) Multi-level queue scheduling

→ We'll have multiple Ready queues



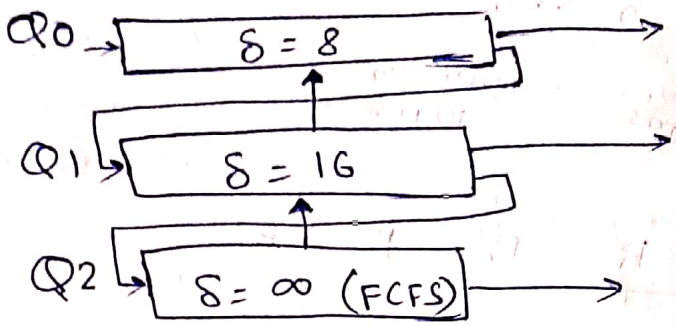
→ Multiple ready queues with different priorities and scheduling ~~processes~~ policies.

→ A process is assigned to one of the queues → priority
- → Foreground
- → Background

→ * processes do not move across queues.
  - ↳ It can lead to starvation.

(9) Multilevel feed back queue scheduling

→ 3 queues



→ Q_0 has the highest priority & Q_2 has the lowest priority.

→ A new process is assigned to Q_0.

→ Let's say Current running process is in Q_0 & its CPU burst < 8 ms

↓

Then, the process remains in Q_0.

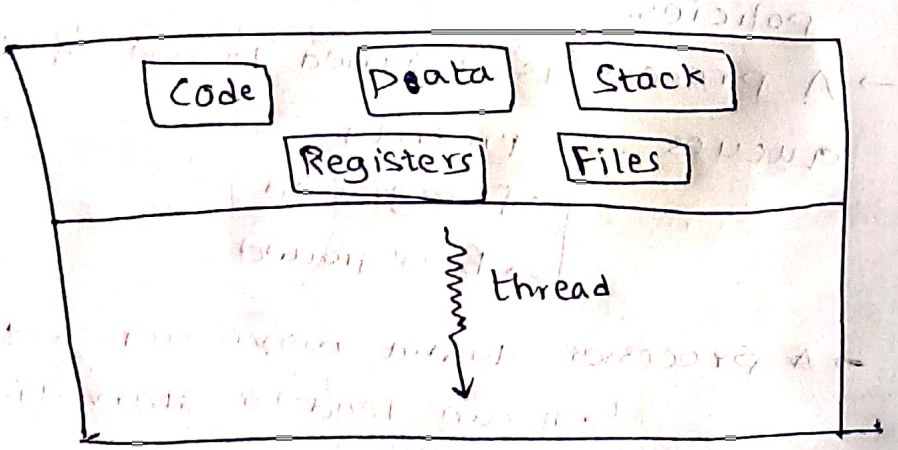
if CPU burst > 8 ms,

Move the process to Q_1.

→ A lower priority job can move up the queues also if it waits for too long

* Multithreading

→ A process is a program in execution with a single thread of control

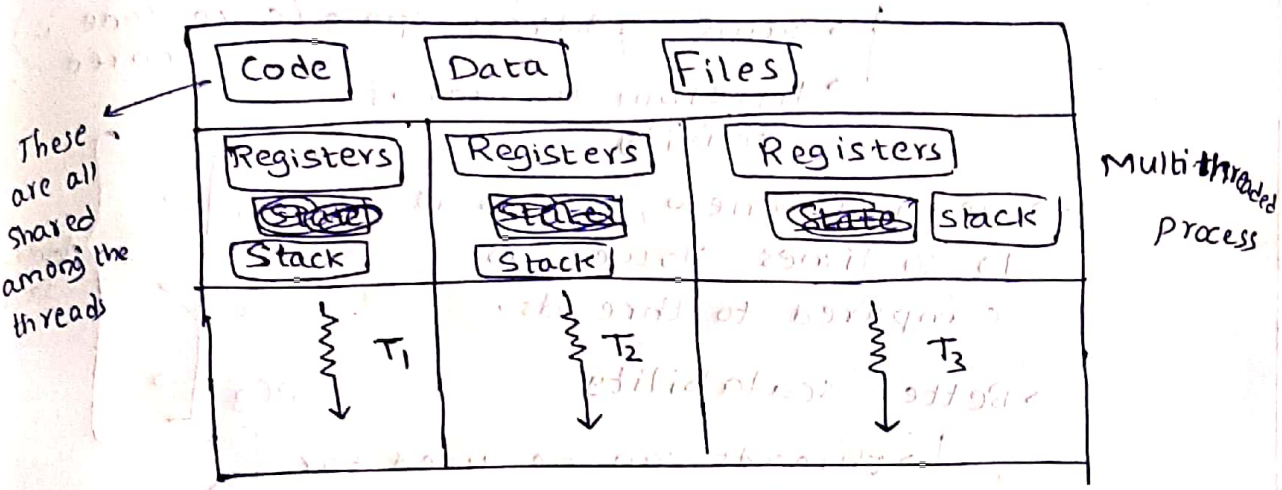


→ What is a thread?

→ Also referred to as light-weight process

→ Contains a thread ID, Program Counter (PC), registers and stack.

→ shares the code, data, files etc. with the peer threads.

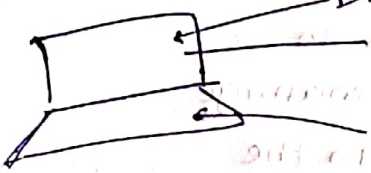


→ Why do we require multiple threads?

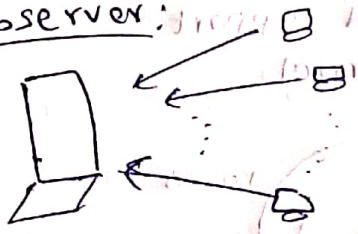
→ Context switch overhead is significantly reduced. [since only Registers & stack need to be saved, remaining all are same].

Ex:/

→ Web client:-



→ Webserver:



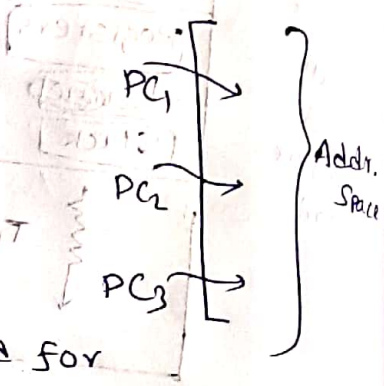
→ Multithreading benefits

→ context switch among threads is faster.

→ Better responsiveness in interactive applications.

→ Better resource utilization among peer threads
  - same address space (since code is shared)
  - different threads of activity

→ creating a new process is 10-30 times slower as compared to threads.



→ Better scalability
  - threads can be used for multiprocessor environments.

→ Multithreading models

Types of threads

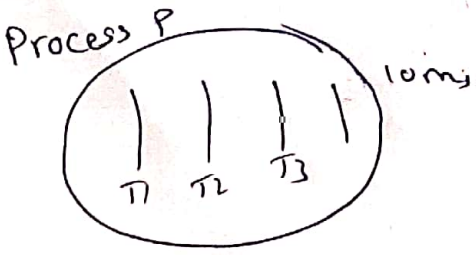
User-level

use APIs to create & manage threads in the user space

Don't need support from kernel.

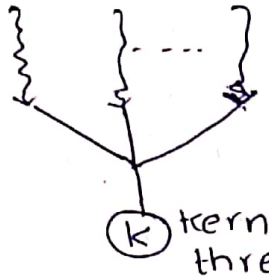
Kernel-level

Supported & managed by the OS.



If a thread makes a blocking system call to the kernel, whole process is blocked since the kernel doesn't know about threads of a process

(a) Many-to-one model:

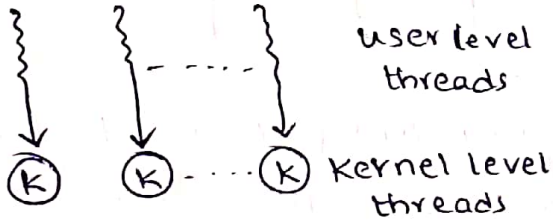


→ Thread management is done by a thread library in user space

→ Entire set of threads will be blocked if one of them makes a blocking system call.

→ not suitable for multiprocessors

(b) One-to-One Model:



→ provides more concurrency

→ when a thread blocks, others can run.

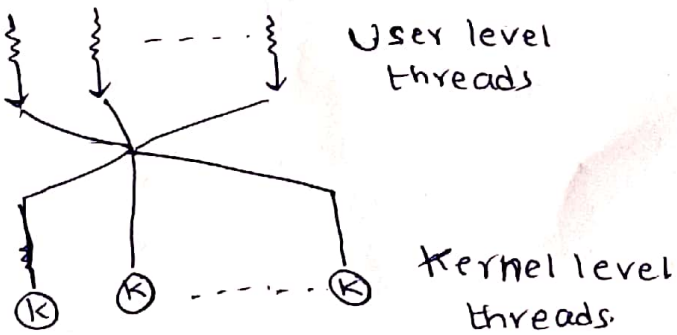
→ Additional overhead w.r.t. kernel.

[∵ kernel has to manage many kernel threads]

→ Linux / windows follows this.

→ can be extended to multi processors

(c) Many-to-many Model:



→ Solaris OS used this model