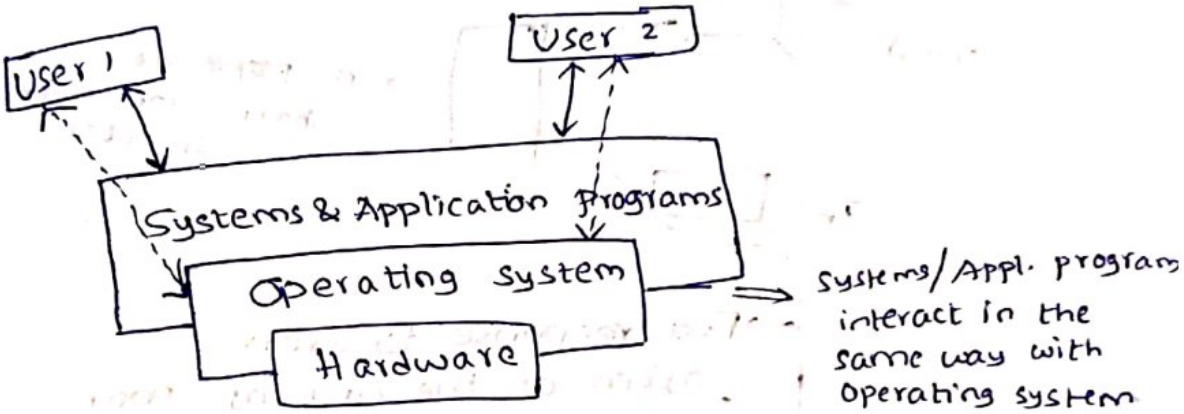
→ Web link :-
http://www.facweb.iitkgp.ac.in/~isg/os

→ 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

Scanned by CamScanner

* Evolution :-

→ Mini Computers and Mainframes (1970-80)

Ex: IBM 1130

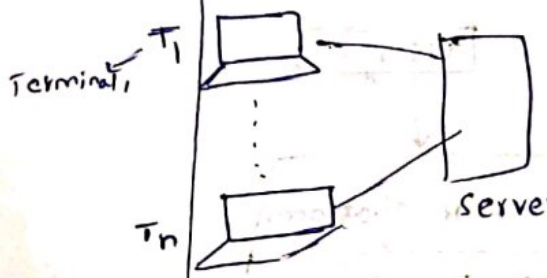
[Hallorith code → in Punched Cards]

→ 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 P_1 P_2 P_3

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.

Scanned by CamScanner

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

→ Flexible user interface, easy to use

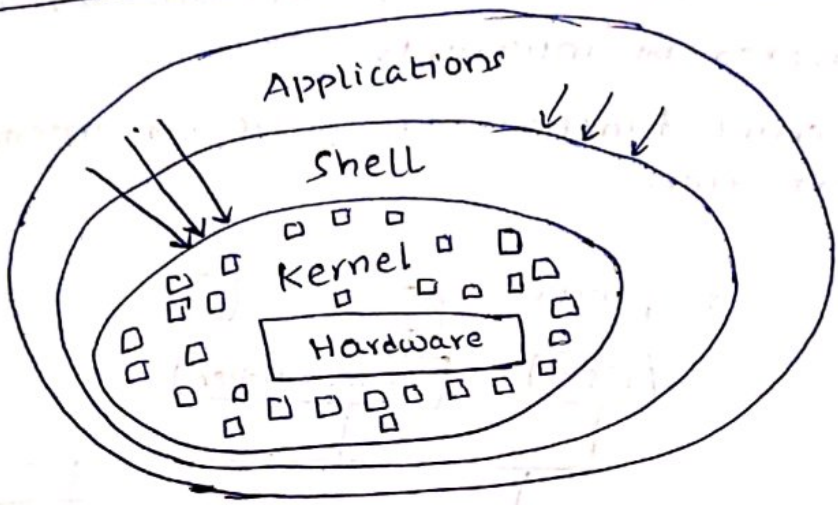
→ Single-user system (fully personal)

→ Mobile Systems

→ Constrained battery life

→ User experience

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

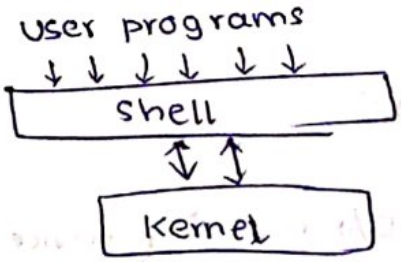


□ → function

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

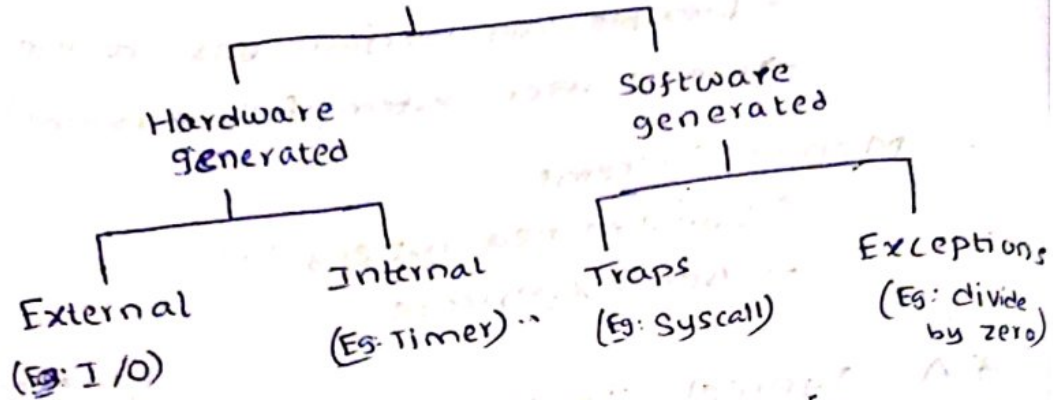
→ kernel runs at all times → INCORRECT Statement

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



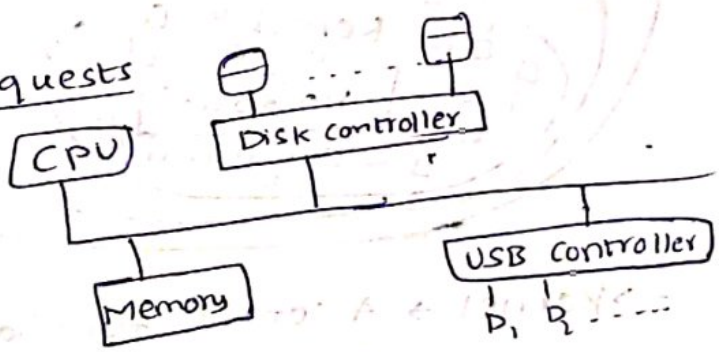
Scanned by CamScanner

* 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

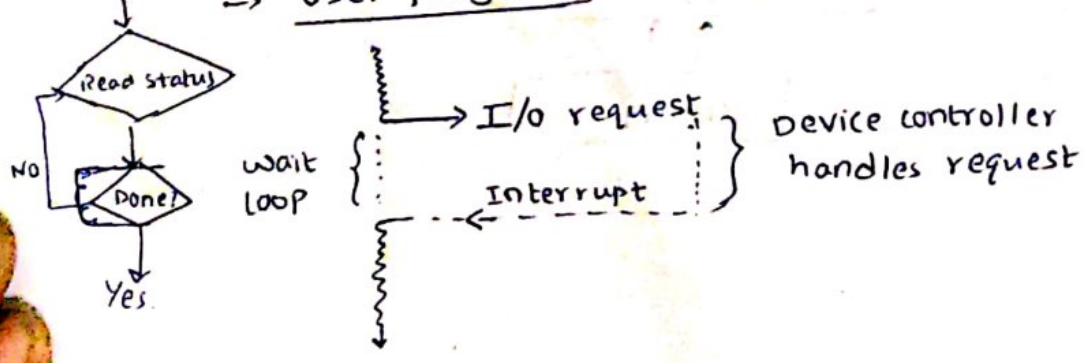


→ If there is no Disk controller, CPU has to spend so much time in disk access.

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

* Alternate I :-

→ User program

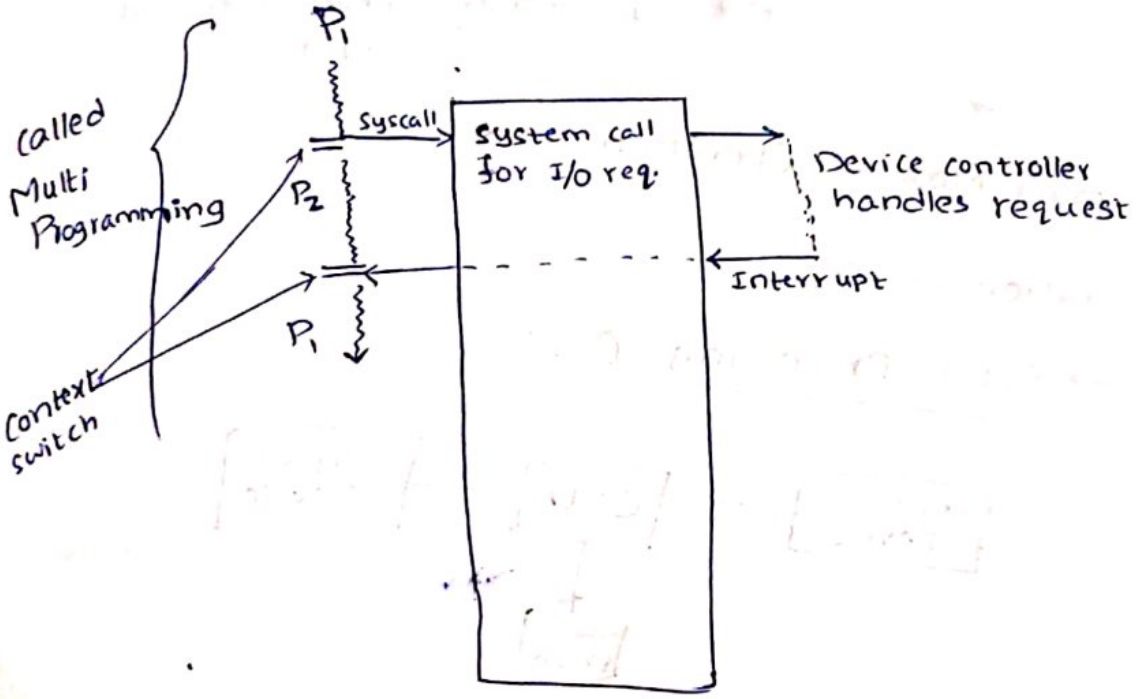


Scanned by CamScanner

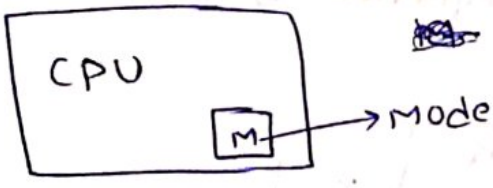
→ The above scheme is called ~~busy ...~~
"BUSY WAITING".

* Alternative 2 :-

→ Active programs P_1, P_2, P_3.

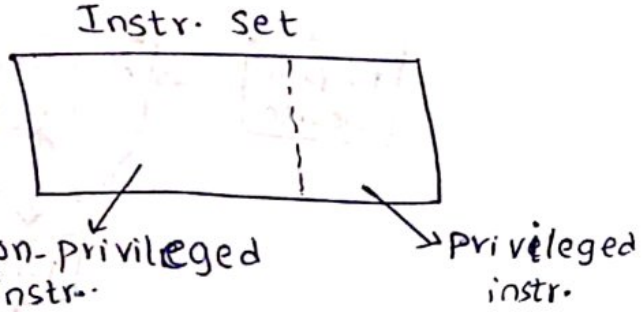


* Syscall



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.

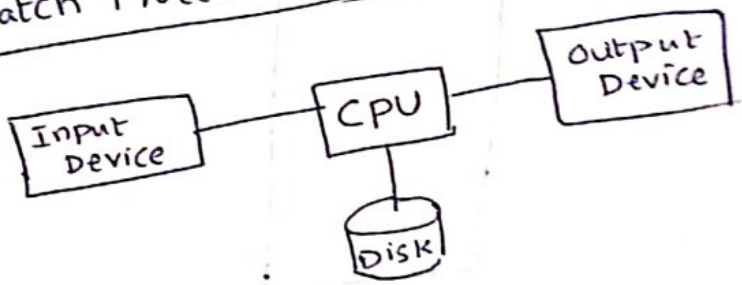
Scanned by CamScanner

→ Syscall ⇒ Set M=0 (Interrupt Handler)
Transfer control to IH (inside kernel)
↓
IH executes
↓
RETI (Return from Interrupt)
Sets M=1

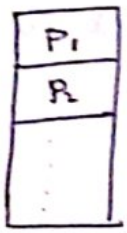
Dual-Mode of Operation.

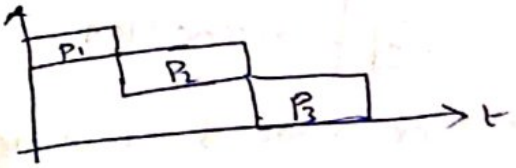
*Evolution of OS:-

→ Batch Processing OS:-

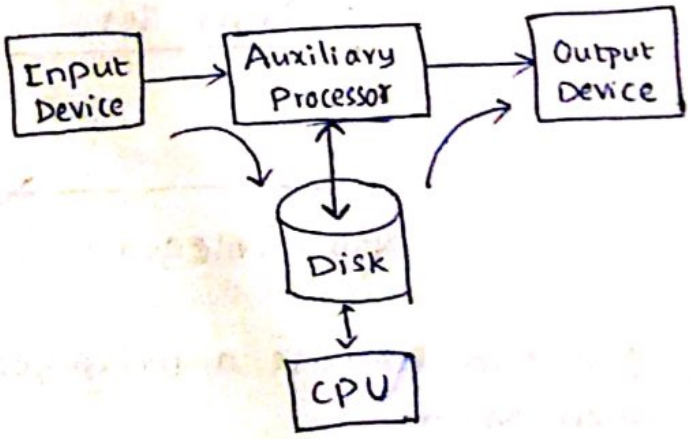


→ programs execute one at a time.
→ Batch: a sequence of programs





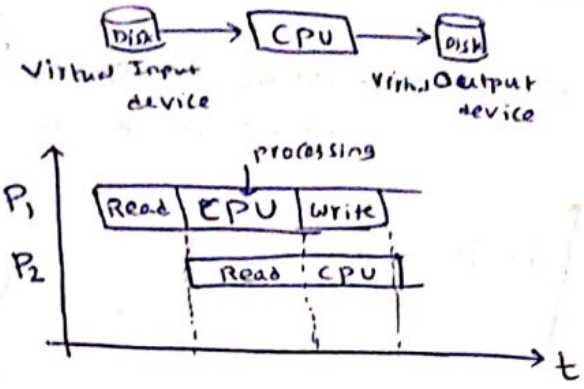
→ Simultaneous Peripheral Operations On Line (SPOOL)



Scanned by CamScanner

→ 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 a 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

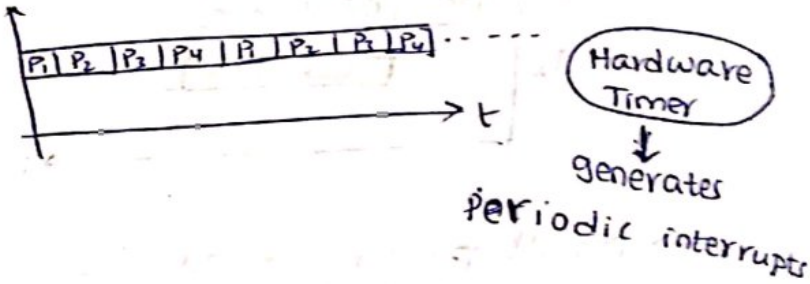
Scanned by CamScanner

→ 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 of time (5-10ms) to each prog.



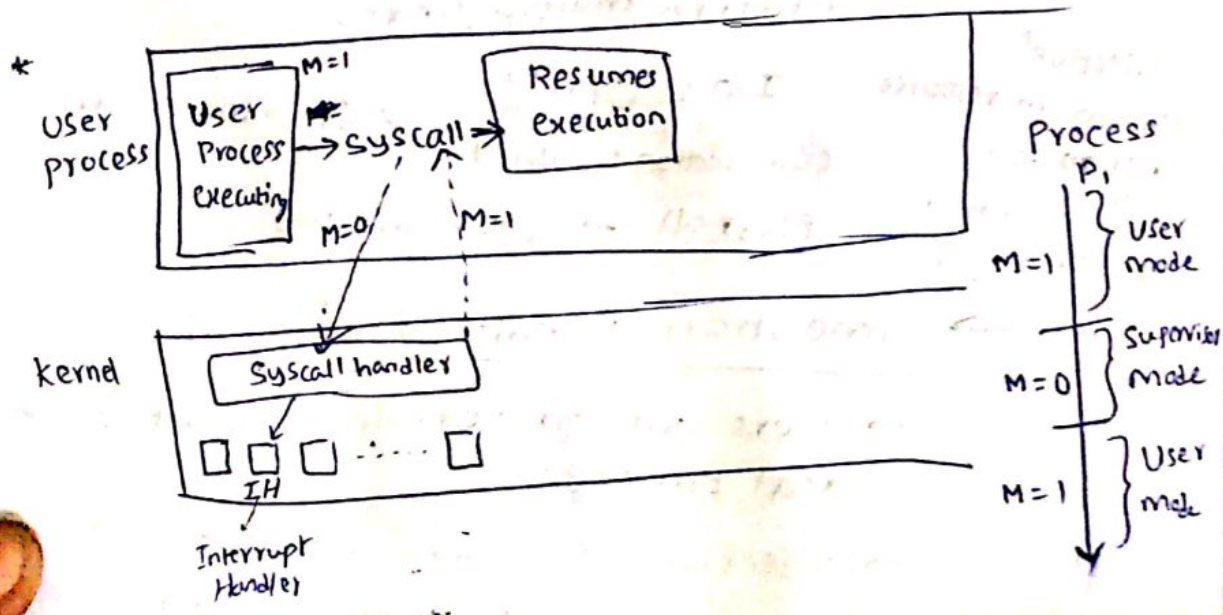
→ **Real-time OS:-**

→ 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

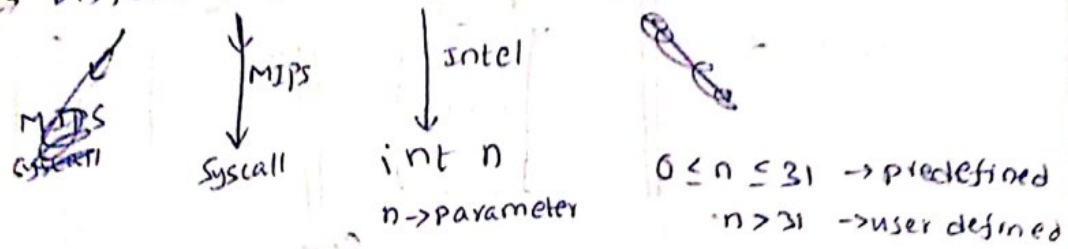


Scanned by CamScanner

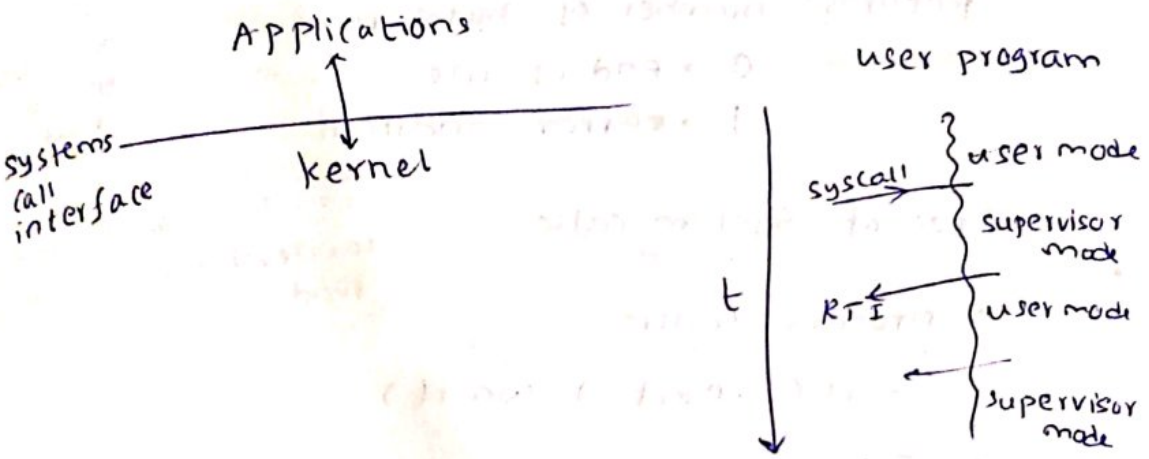
* System Calls

-> It is a machine instr.

-> different for different ISA.



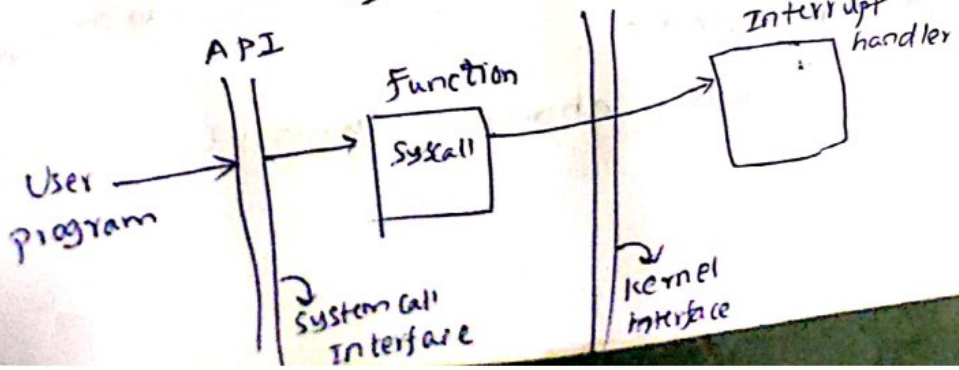
-> system call instr. is executed
- -> software interrupt
- -> Mode = supervisor
- -> Invoke interrupt handler
- -> Return from interrupt (Mode = user)



-> system call interface :-

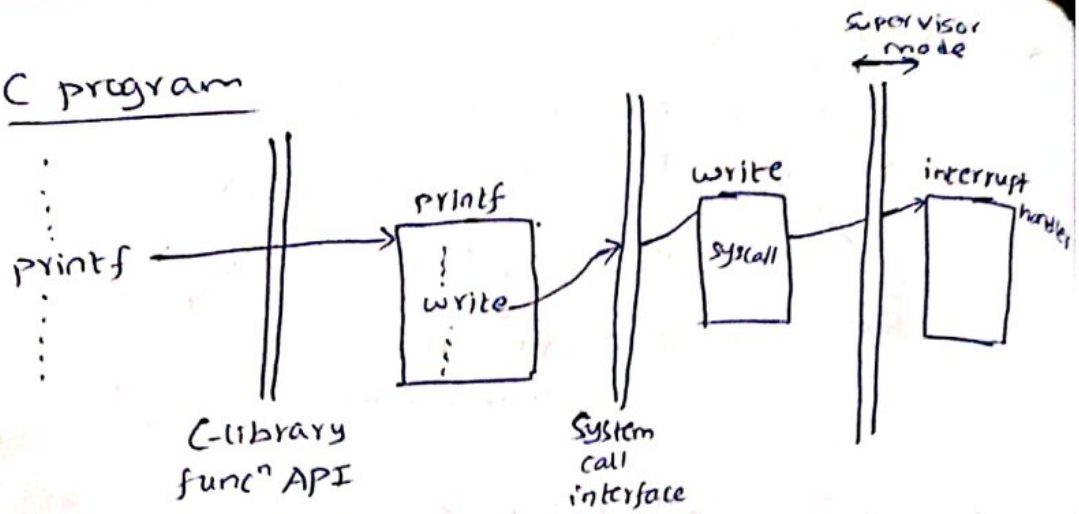
-> provided by the OS
- -> A set of low-level func^{n}s that can be called from a user program.

-> These low-level func^{n}s execute some system call.



Scanned by CamScanner

* 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:- (related to semaphores)
pipe(), semget(), semop(), semctl(),
shmget(), shmat(), shmdt(), shmctl(),
kill(), signal().

(pipe in terminal)

(related to shared memory)

(d) Protection:-
chmod(), chown(), umask()

Scanned by CamScanner

* 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.

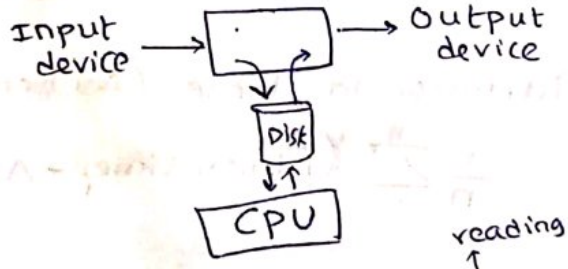
CPU idle time = 205 – 1000 × 5 ms

= 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

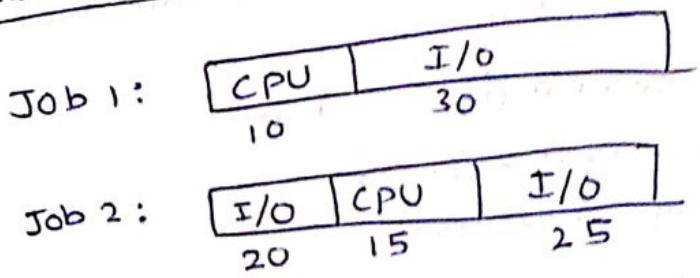
CPU idle time = 25 – 1000 × 5 = 20 s.

card reader reads one card every (100 + 10) ms ≈ 9 cards/second (100: reading from card; 10: writing to disk)

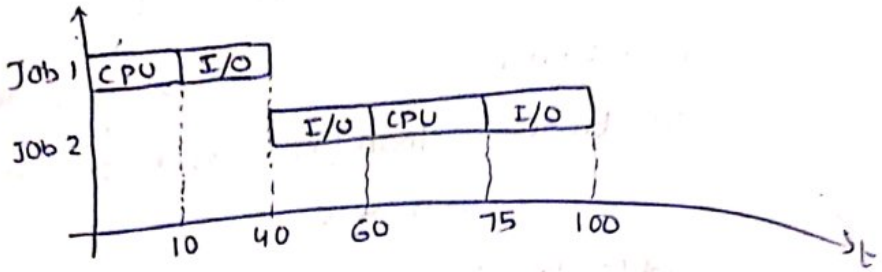
CPU utilisation = $\frac{5}{25} \times 100 = 20\%$

Scanned by CamScanner

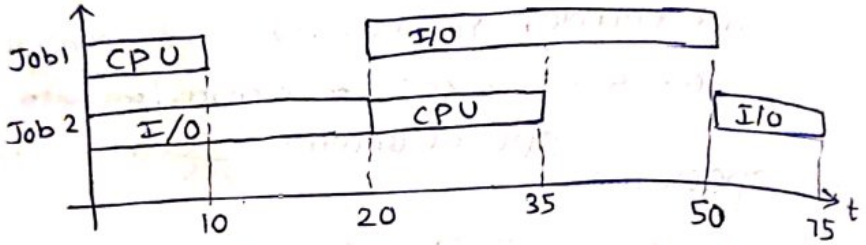
2. Related to multiprogramming :-



(a) Batch execution :-



(b) Multiprogramming :-



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

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

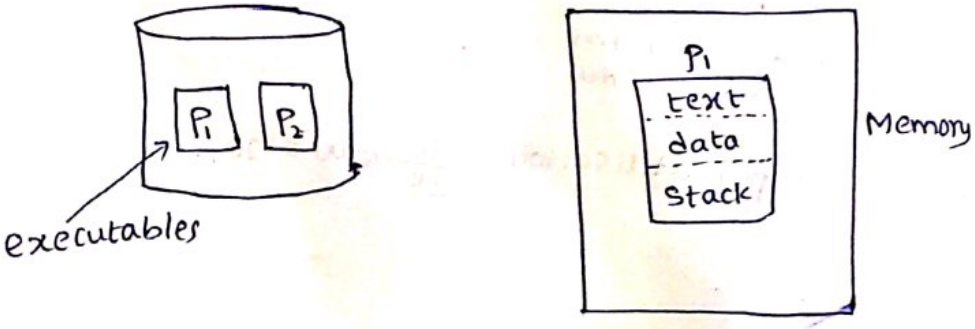
→ Average throughput

= No. of jobs completed per unit time

* Process :-

→ A program in execution

(terminologies of UNIX)



Scanned by CamScanner

→ A process consists of several regions

→ text → contains executable code

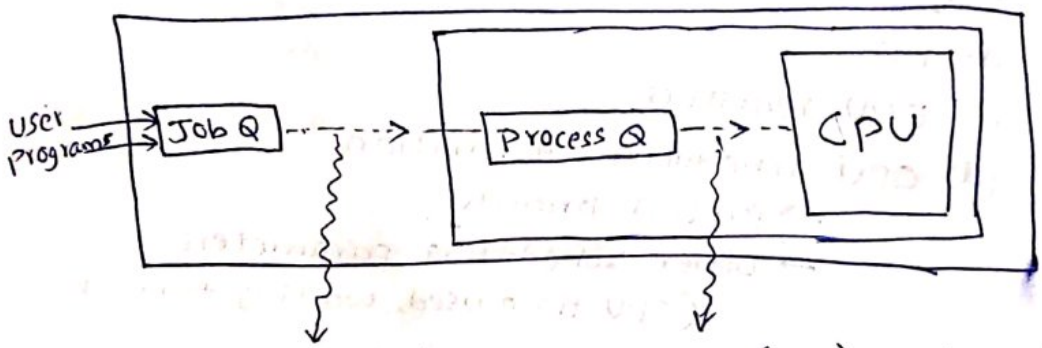
→ data → contains global data (static data)

→ 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

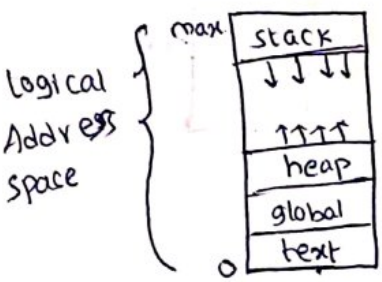
Process (CPU) scheduler / short-term scheduler
↓
decides which process should run now

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

→ Structure of a process :-

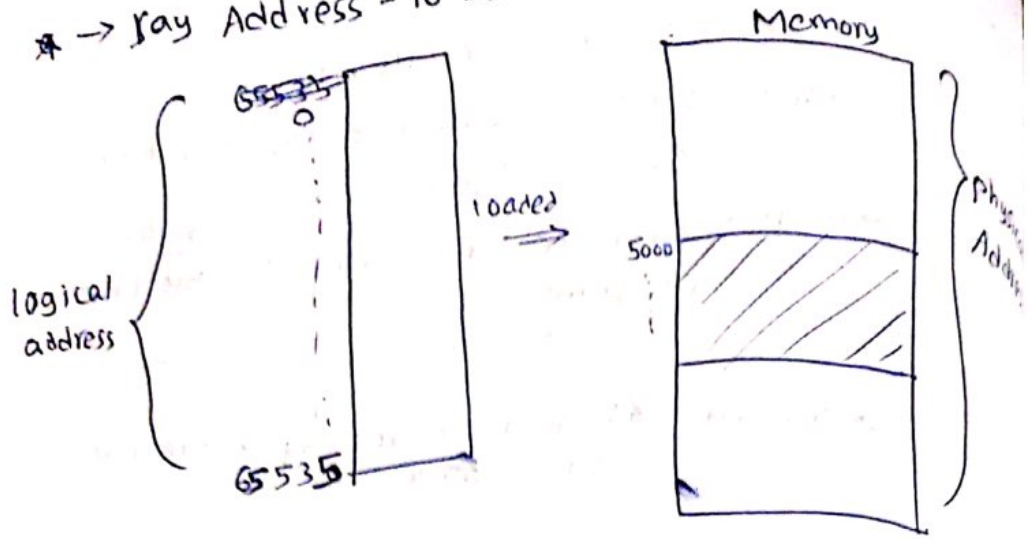
→ Each process is represented in the kernel using a data structure called Process Control Block (PCB).

↓
Contains relevant information about a process.



Scanned by CamScanner

* → Say Address - 16 bits



* Typical PCB entries:-

(a) process state

(b) PC

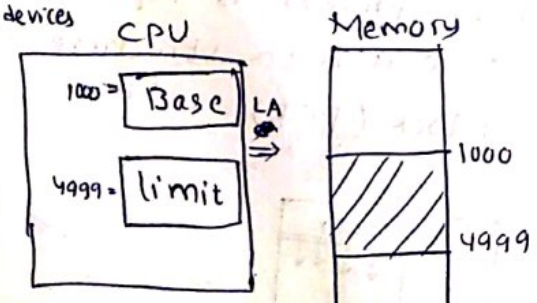
(c) CPU registers

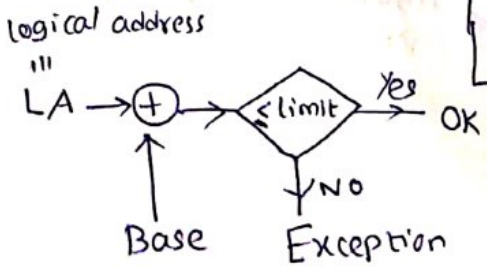
(d) CPU scheduling information
- → process priority
- → other scheduling parameters (CPU time used, waiting time etc.)

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

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

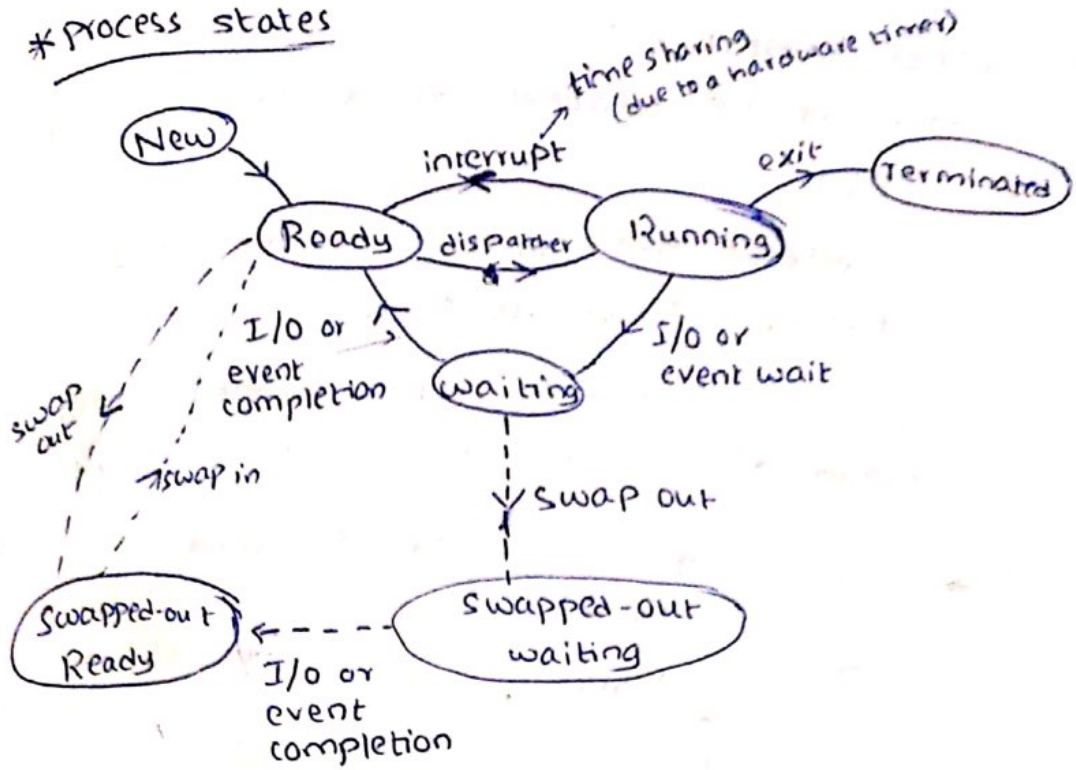
(g) I/O status info
- → list of I/O devices being used
- → list of open files





Scanned by CamScanner

*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.

Scanned by CamScanner

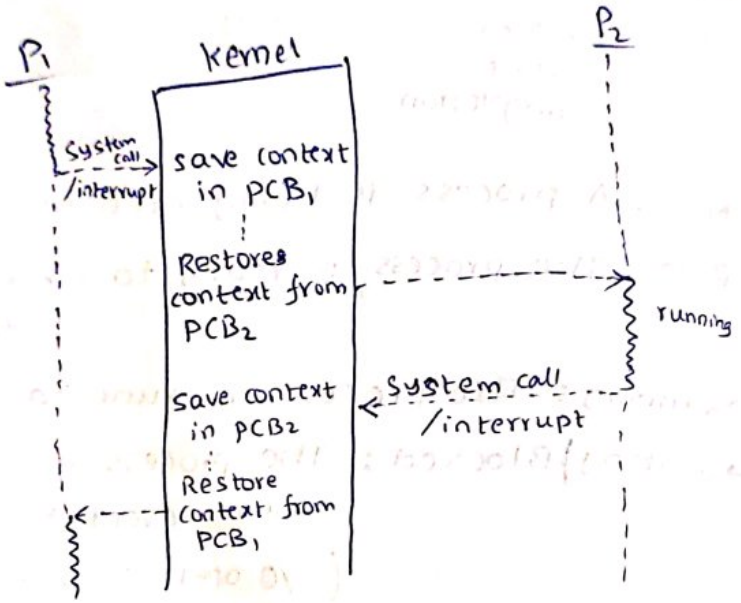
→ Context switch → overhead

→ 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)



P1 & P2 are concurrent processes

* Process scheduling :

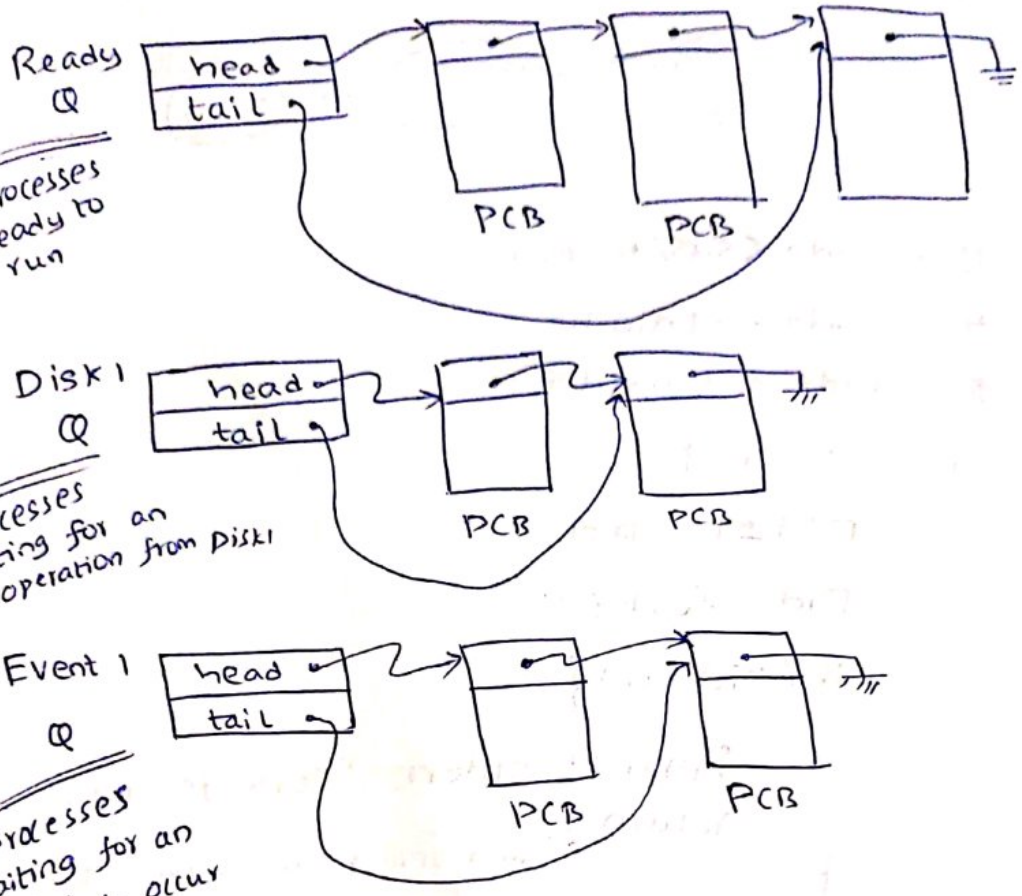
→ Ready state → queue of processes

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.

Scanned by CamScanner

* One of the many scheduling queues maintained by kernel.
  → linked list of PCBs



* Operations on Processes:-

→ Process creation:-

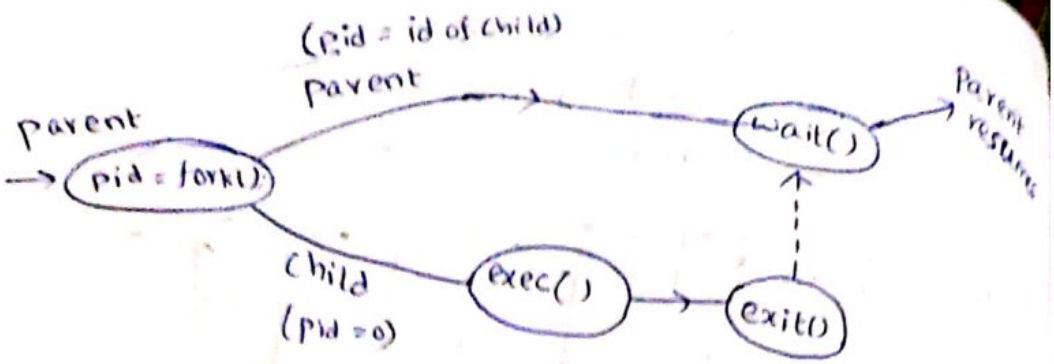
→ 'ps' command lists all the processes active.

→ 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.

Scanned by CamScanner



*
```
# 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;
      }
```
↳ non-zero return value ⇒ some error
```
    else if (pid == 0){
          execlp("/bin/ls", "ls", NULL);
      }
      else {
          wait(NULL);
          printf ("child complete");
      }
      return 0;
}
```


Scanned by CamScanner

→ 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

Scanned by CamScanner

* 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

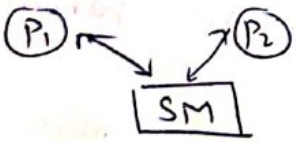
Process that are dependent. (share some data)

* **Co-operating processes :-**

→ Some mechanism using which the processes can exchange data.

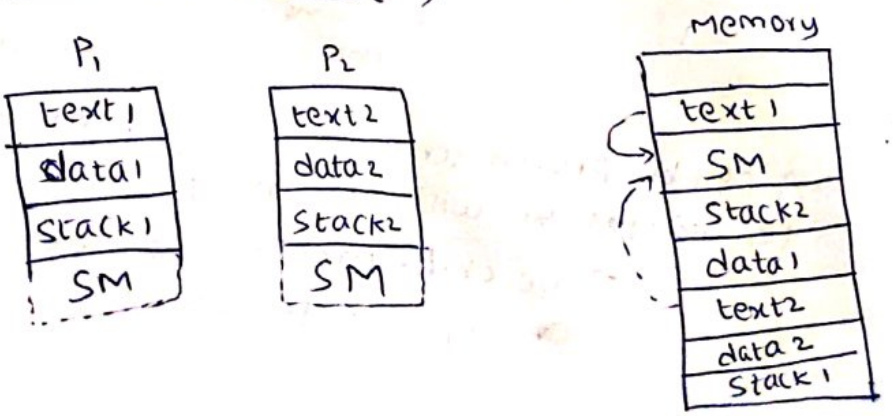
Inter Process Communication (IPC)

Shared Memory (SM)



Message Passing

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



Scanned by CamScanner

→ 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 ⇒ removes 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

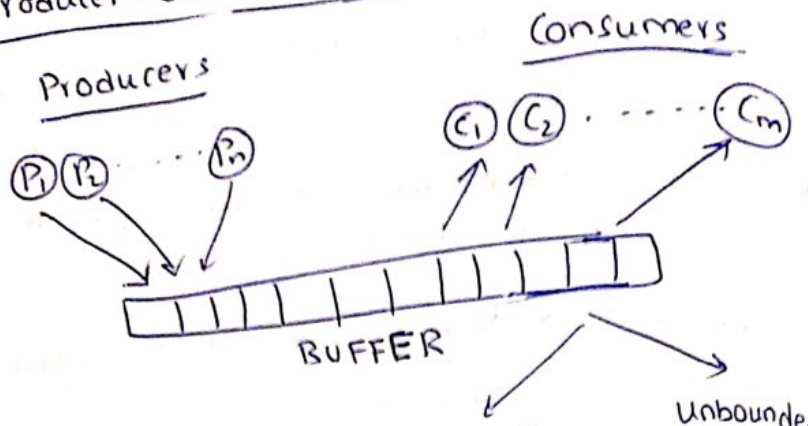
every user has a group id also

Scanned by CamScanner

* 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 :-

out ≡ pointer to the first data item in the buffer

in ≡ pointer to next free location in Buffer

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

→ 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;
}
```


Scanned by CamScanner

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_SIZ~~

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

in low level lang.

let's say S_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

Scanned by CamScanner

The above is called RACE condition.

↓

also happens when 1 is given to S-R Latch.

Example: Producer - Consumer problem

Producer

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

Consumer

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

→ Two approaches:

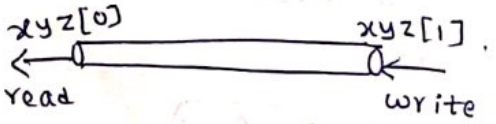
(a) unnamed pipes

(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.

Scanned by CamScanner

→ 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

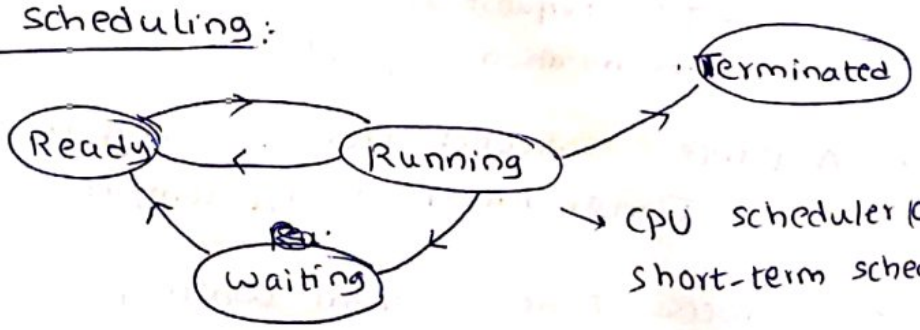
parent:
```
fd = open(myfifo, O_WRONLY);
write(fd, ....);
close(fd);
```

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 Commn across machines
- → Remote procedure call
- → Socket.

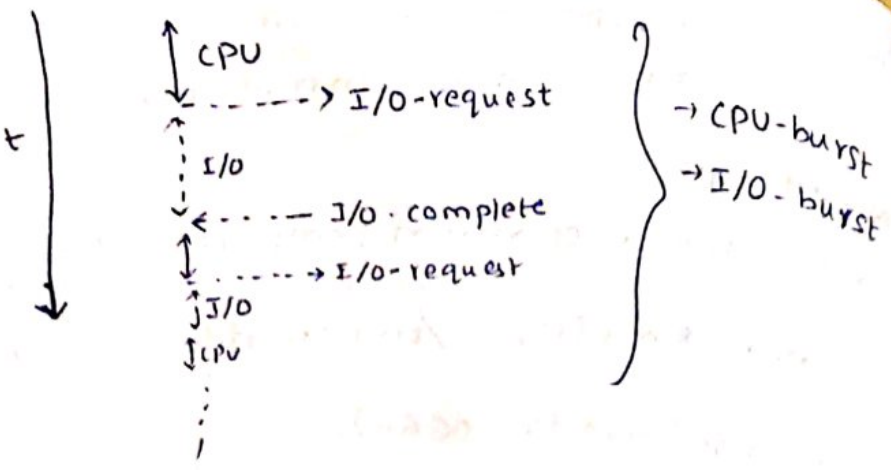
* CPU scheduling:

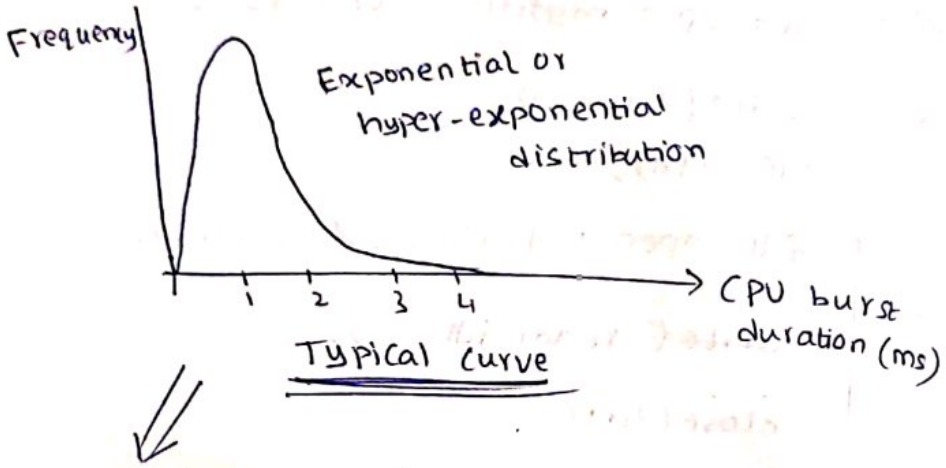


→ CPU scheduler (or) short-term scheduler

Typical process execution :-

Scanned by CamScanner





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.

Scanned by CamScanner

→ (a) and (d)

→ Non-Preemptive scheduling

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

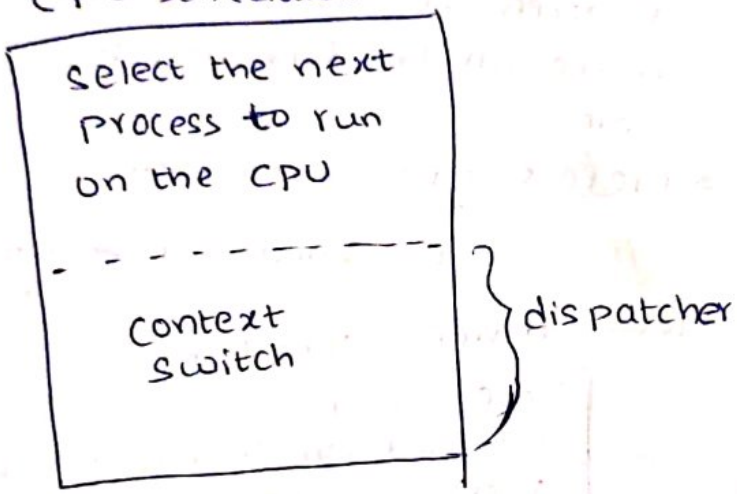
Here, the processes are coming out of CPU on their own.

→ (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.

Scanned by CamScanner

→(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.

⇓

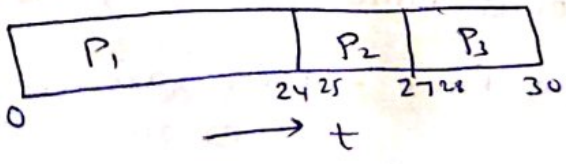
lower device utilization.

[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) $P_1 - P_2 - P_3$



Scanned by CamScanner

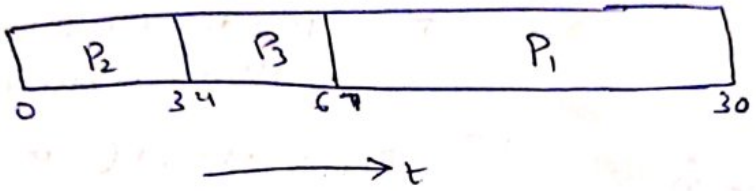
Avg. turnaround time = $\frac{(24-0)+(27-0)+(30-0)}{3}$

= 27 ms

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

= 17 ms

(b) $P_2 - P_3 - P_1$



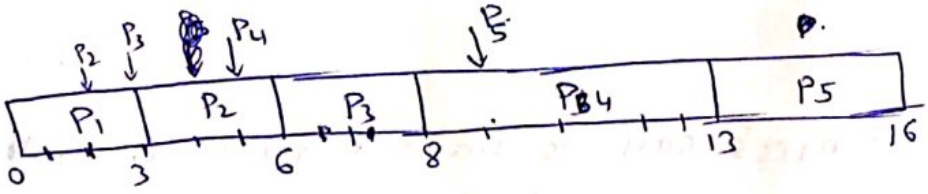
Avg. turnaround time = $\frac{(30-0)+(6-0)+(3-0)}{3}$

= 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

Scanned by CamScanner

(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 $0 \le \alpha \le 1$

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

τ_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
  ✓ Avg. Waiting Time

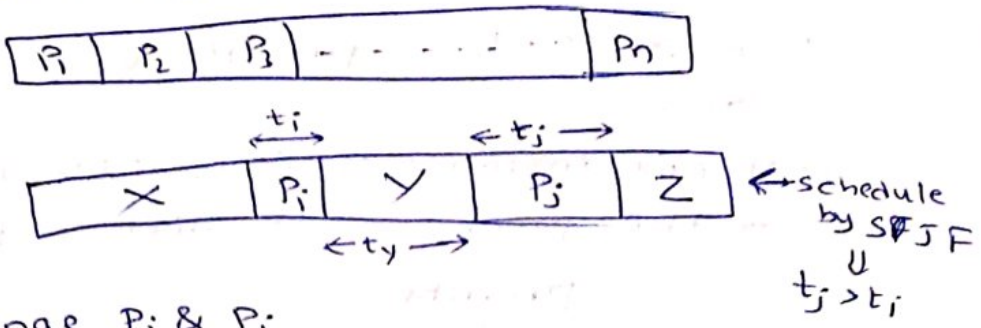
→ Let's say we have 'n' processes that all arrive at $t = 0$.
  CPU burst times are $t_1 \le t_2 \le \cdots \le t_n$.
  $P_1 \quad P_2 \quad \cdots \quad P_n$

✓ Proof that in this case, SJF minimizes AWT.:

Proof by contradiction.

Scanned by CamScanner

→ 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_i : waiting time increases by t_y.

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

→ preemptive version of SJN

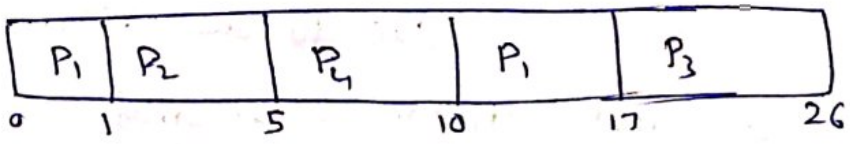
↳ A process with a shorter CPU burst can preempt a running process

→ 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



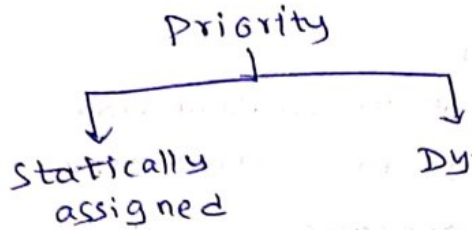
$$AWT = \frac{9+0+15+2}{4} = 6.5$$

Scanned by CamScanner

(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

Scanned by CamScanner

→ Response Ratio,

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

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

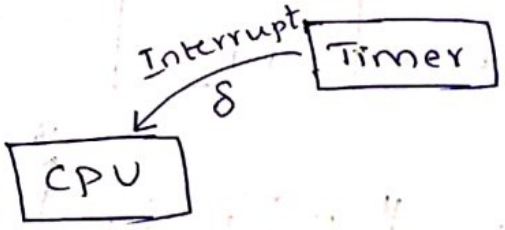
→ RR can indicate priority

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

(e) Round-robin scheduling :

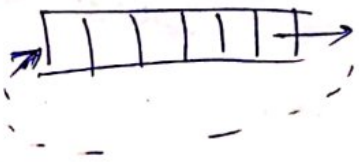
→ Designed primarily for interactive time-stationary systems.

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

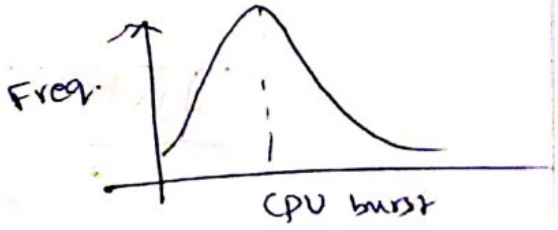


→ OS loads the timer
→ Decrements with a clock
→ Interrupts when I/O.

Ready list
↳ stored as circular queue.



→ The instruction to set the timer is priveleged instruction.



Scanned by CamScanner

CPU burst

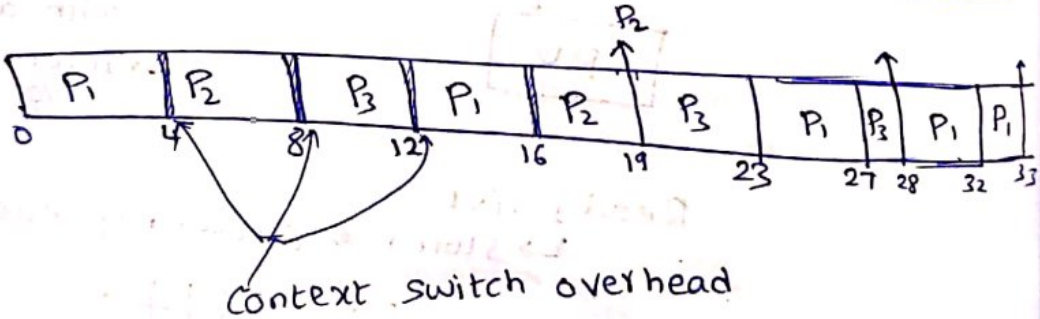
$< \delta$: process invokes a system call or an external interrupt arrives (say I/O)
⇓
process may go to waiting or ready state respectively (i.e process releases the CPU.)
⇓
Context switch

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

→ Example :-

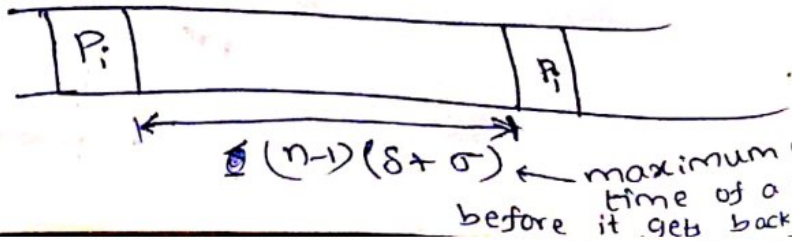
$\delta = 4$ ms

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.



Scanned by CamScanner

Very high δ ⇒ similar to FCFS.

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

$\delta >> \sigma$

δ typically 10 ms

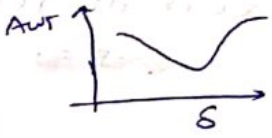
σ typically 10 μs.

Q.

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

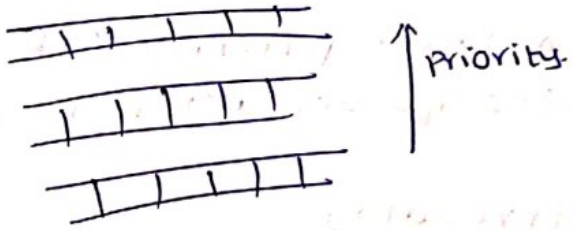
Compute AWT for $\delta = 1, 2, 3, 4, 5, 6, 7$.

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.

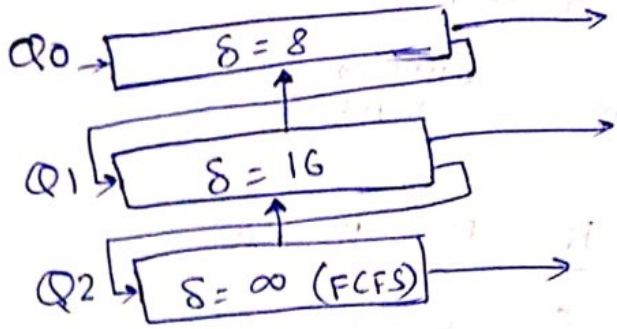
→ A process is assigned to one of the queues → priority
- → Foreground
- → Background

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

Scanned by CamScanner

(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.

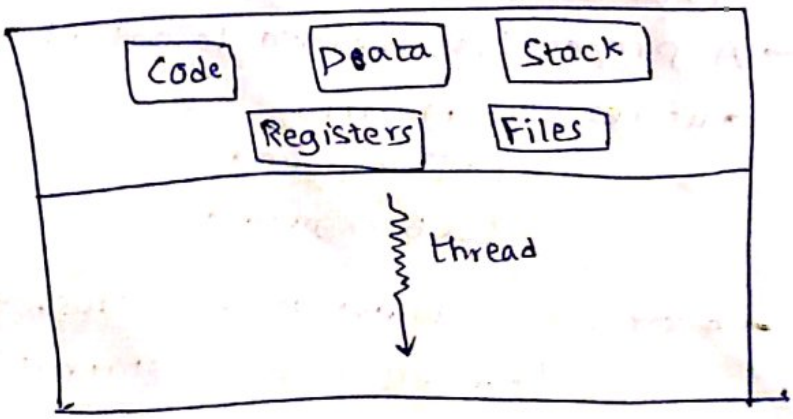
if cpu burst > 8 ms,

· 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



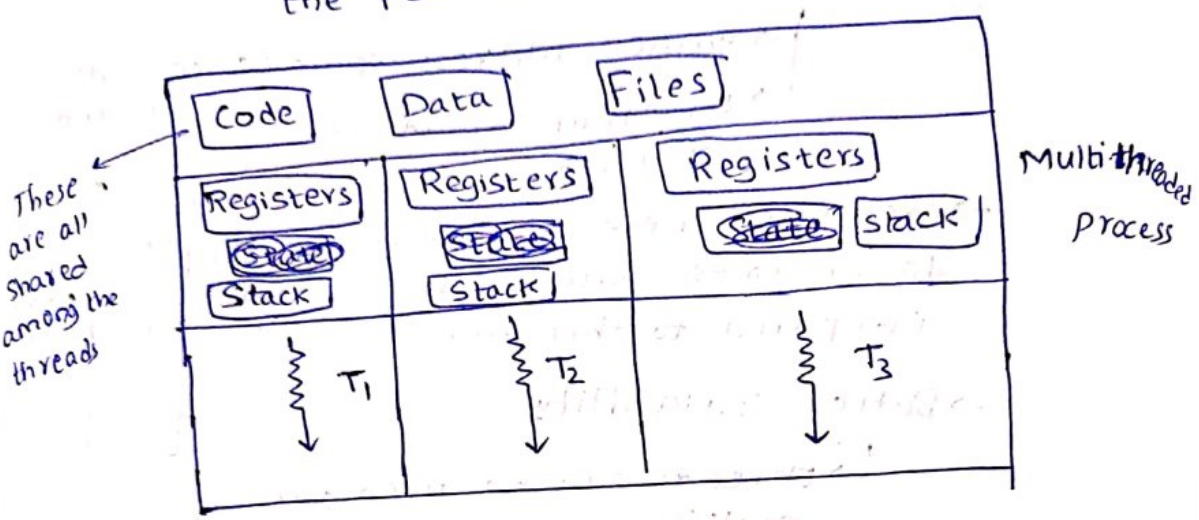
Scanned by CamScanner

→ 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.

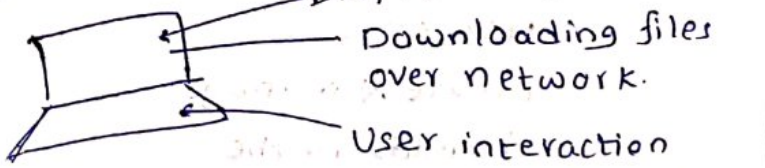


→ Why do we require multiple threads?

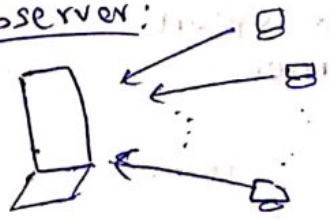
→ Context switch overhead is significantly reduced. [since only Registers & stack need to be saved, remaining all are same].

Ex:

→ Web client:-



→ Webserver:



Scanned by CamScanner

→ Multithreading benefits

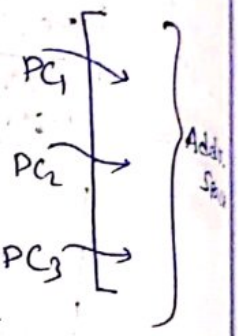
→ context switch among threads is faster

→ 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

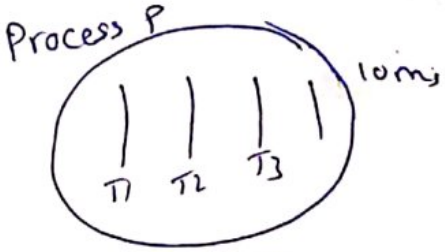
Use APIs to create & manage threads in the user space

⇓

Don't need support from kernel.

Kernel-level

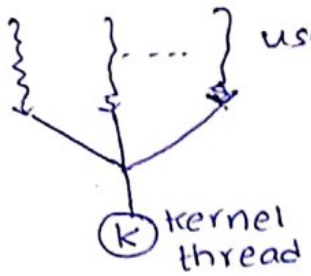
Supported & managed by the OS.



If a thread makes a blocking system call to the kernel, whole process is blocked since the kernel doesn't know about threads. It only sees as a process

Scanned by CamScanner

(a) Many-to-one model:

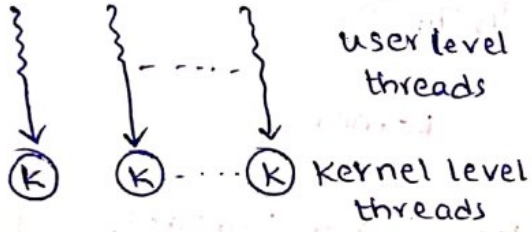


→ Thread management is done by a thread library in user space

→ 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.

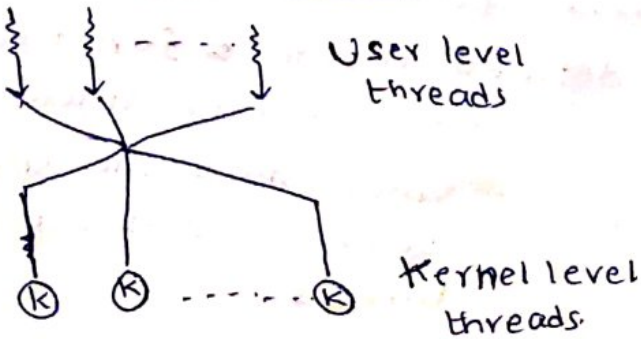
→ Additional overhead w.r.t. kernel.

[∵ kernel has to manage many kernel threads]

→ Linux / Windows follows this.

→ can be extended to multi processors

(c) Many-to-many Model:



Scanned by CamScanner

→ Solaris OS used this model.

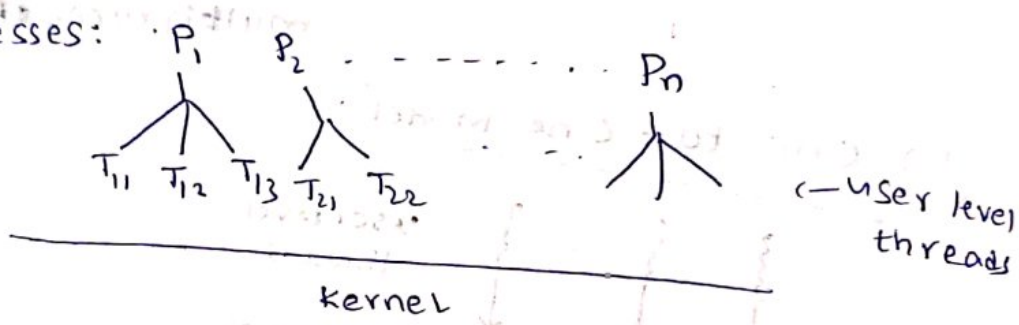
* Class Test 1

→ 7th February
7-8 PM
Rooms: CSE-107 & CSE-108

* Lab Quiz Test

→ 10th February
During Lab hours.

* Processes:



→ Irrespective of how many threads a process creates, other process's run time is not hampered
→ Allocating time to threads can be done using thread library.

* POSIX Thread Library (pthread Library)

→ Define the threads as ULT or KLT
(User Level Threads, Kernel Level Threads)
→ All global data are shared among threads.
→ Data defined inside functions
→ allocated on stack
→ Since stacks are separate, separate copies are created for threads.

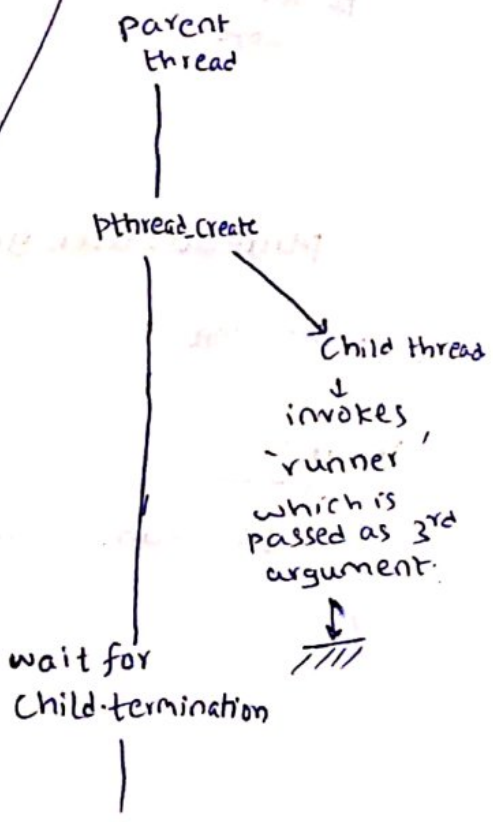
Scanned by CamScanner

```c
#include<stdio.h>
#include <pthread.h>
int sum;
void *runner (void * param);
int main(int argc, char argv[ ]).
{
    pthread_t tid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join ( tid, NULL);
    printf ("\n sum is %d", sum);
}
void * runner( void *param);
{
    int i, N = atoi(param);
    sum = 0;
    for (i=1; i <= N; i++)
        sum += i;
    pthread_exit(0);
}
```

retrieving the default attributes

creates a thread

wait for tid's termination.



Scanned by CamScanner

* Creating multiple threads

```
#define N_THR 10
Pthread_t mythreads[N_THR]
    .
    .
for(int i=0; i<N_THR; ++i)
    Pthread_create(&mythreads[i], &attr,
                   runner, NULL);
    .
    .
for (int i=0; i<N_THR; ++i)
    pthread_join ( mythreads[i], NULL);
```

* Thread scheduling using pthread

⇒

```
Pthread_attr_setscope(pthread_attr_t *attr, int scope);
```

- Pthread_attr_setscope → to set the scope
- int scope → can take values
  - User Level Thread → PTHREAD_SCOPE_PROCESS
  - (or)
  - Kernel Level Thread → PTHREA_SCOPE_SYSTEM

```
Pthread_attr_get_scope(pthread_attr_t *attr,
                       int *scope );
```

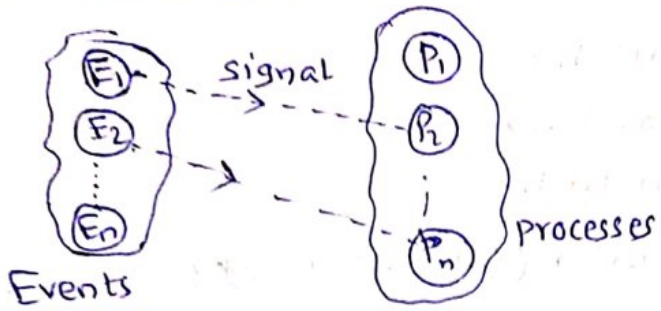
- Pthread_attr_get_scope → to get the scope

* Thread Cancelling

```
Pthread_cancel (pthread_t tid);
```


Scanned by CamScanner

*Signal Handling



→ A signal notifies a process that some event has occured.

→ An event occurs.
- → A signal corresponding to the event is generated
- → Signal is delivered to a process
- → Once delivered, the signal must be handled appropriately by the process.

→ Some of the POSIX signals
- → SIGABRT → Abort
- → SIGBUS → Bus error
- → SIGILL → Illegal instruction
- → SIGKILL → Kills a process
- → SIGQUIT → terminal quit
- → SIGSEGV → invalid memory reference (Segmentation fault)
- → SIGUSR1, → SIGUSR2 → User-defined
- → SIGINT → Ctrl + C

Scanned by CamScanner

* Signal handler in C

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int sig_no){
    if (sig_no == SIGINT)
        printf("\nReceived signal");
}
int main(){
    signal(SIGINT, sig_handler);
    while(1)
        sleep(1);
}
```

↑ install the signal handler

*Assign-1:-

Batch processing system:-

M : average mounting time (↑ to load the jobs in Hard disk)
T : average service time per job.
N : number of jobs
S : unit price of service time
W : unit price of waiting time per user.

Find optimum batch size (N) that minimizes total cost.

Sol: Total time = $M + NT$

Cost per unit time = $S + NW$

$$\frac{\text{Total cost}}{\text{user}}, C = \frac{(M+NT)(S+NW)}{N}.$$

$$C = \frac{MS + MNW + NTS + N^2WT}{N}$$

Scanned by CamScanner

$$C = \frac{MS}{N} + (WT)N + (MW+TS)$$

$$\frac{dC}{dN} = 0 \Rightarrow N = \sqrt{\frac{MS}{TW}}$$

* Sending Signal to a process or thread

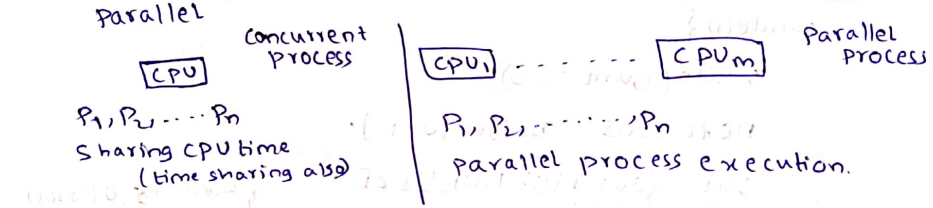
→ For process, kill (pid_t, pid, int Signal);

→ For thread,

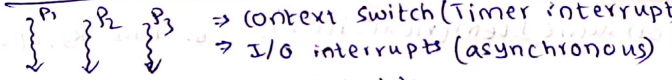
P_thread_kill (P_thread_t tid, int signal);

* Process Synchronization :

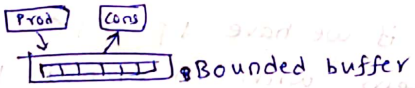
→ Multiple processes either execute concurrently or in parallel



* Concurrent processes:



Ex: Producer-consumer problem



→ Shared Memory

```
#define BUF_SIZE 50
typedef struct{
    ---
    ---
} item;
item buffer[BUF_SIZE];
int in=0, out=0;
int count=0;
```

→ shared Memory

↓ No. of items in buffer

Producer P

```
while(1){
    next_prod = ProduceItem();
    while (count == BUF_SIZE);

    buffer[in] = next_prod;
    in = (in+1) % BUF_SIZE;
    count++;
}
```

count++; →
1. LOAD R_1, 0(count)
2. ADD R_1, R_1, 1
3. STORE R_1, 0(count)

Consumer C

```
while(1){
    while(count == 0);
    next_cons = buffer[out];
    out = (out+1) % BUFSIZE;
    count--;
    consumeItem(next_item);
}
```

count--; →
4. LOAD R_5, 0(count)
5. SUB R_5, R_5, 1
6. STORE R_5, 0(count)

→ Even if we have 1P & 1C, we don't have problem with in or out but ~~count has a pro~~ the value of count is updated by both of them. ⇒ a problem.

→ suppose initially, count = 10
↳ Both P & C run one cycle each.

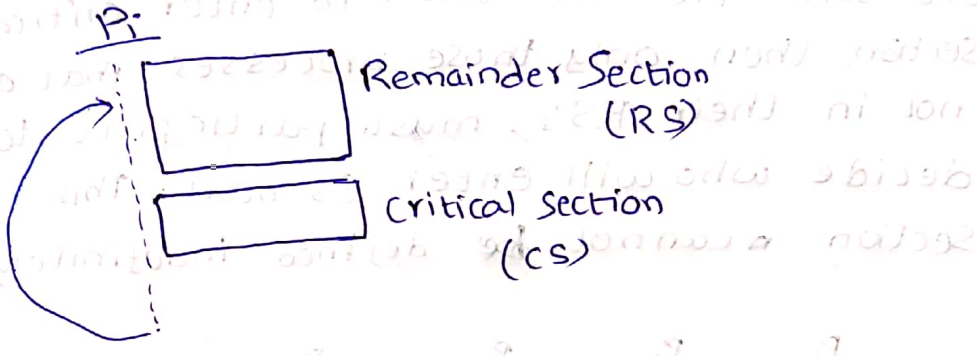
Scenarios	Final count	
1, 2, 3, 4, 5, 6	10	
4, 5, 6, 1, 2, 3	10	
1, 2, 4, 5, 6, 3	11	
4, 5, 1, 2, 3, 6	9	
4, 1, 2, 3, 5, 6	9	→ RACE condition
1, 4, 5, 6, 2, 3	11	

* Critical section Problem :-

→ There are 'n' concurrent processes
$P_1, P_2, \ldots, P_n$

↳ Each P_i has a segment of code where some shared data is updated.

⇓
simple numbers, file, etc--



→ When a process P_i is executing in its critical section, (CS), no other process P_j should be allowed to enter the CS.

⇓
Execution of CS code is mutually excluded in time

⇓
Also called Mutual Exclusion Problem.

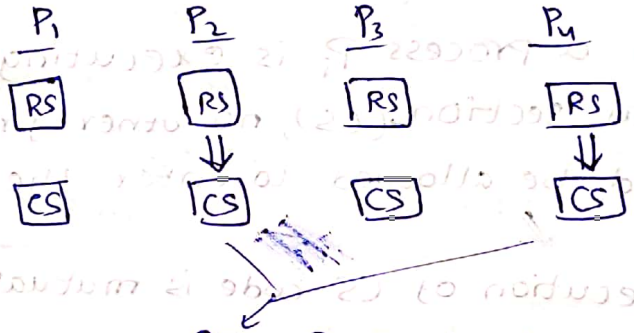
→ Structure of P_i

```
while(1) {
    RS
<Entry section>
    CS
<Exit section>
}
```

* Basic requirements of any correct solution to the critical section problem.

(a) Mutual Exclusion:-
No two processes must be inside their CS at the same time.

(b) Progress :- If no process is executing in CS, and some processes wish to enter critical section, then only those processes that are not in their RS's, must participate to decide who will enter CS next. This section cannot be defined indefinitely.



Only P_2, P_4 decide who among them will go first.
P_1, P_3 has no say.

(c) Bounded waiting:- There exists a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its CS & the request is granted.

→ Let's say P_1, P_2, P_3 are visiting CS frequently. P_4 also wants to enter CS.

⇓

It should not be the case that P_4 will be always denied in favour of P_1, P_2, P_3.

*Software Solution :- [Peterson's Algorithm]

→ Restrict to two concurrent processes only.

```
int turn = 0;
boolean flag[2];
```
→ Shared Memory.

P_0

```
while(1){
    flag[0] = true;
    turn = 1;
    while(flag[1] && turn == 1);
    CS
    flag[0] = false;
    RS
}
```

Entry section: `flag[0] = true; turn = 1; while(flag[1] && turn == 1);`

Exit section: `flag[0] = false;`

P_1

```
while(1) {
    flag[1] = true;
    turn = 0;
    while(flag[0] && turn == 0);
    CS
    flag[1] = false;
    RS
}
```

→ flag[0] → willingness of P_0 to enter CS

→ flag[1] → willingness of P_1 to enter CS

→ turn → specifies who will enter next

* Modern computer Architectures

→ Typical execution sequence

P_1 : turn = 0 → flag[1] = false

P_0 : turn = 1

P_0 : flag[0] = true

P_0 : flag[1] && turn == 1

⇒ P_0 enters CS

P_1 : flag[1] = true

P_1 : flag[0] && turn == 0 → WRONG

⇒ P_1 enters CS

This error occurs for those architectures which can trigger out-of-order execution.

* Hardware Solution for Synchronization:-

lock = false — shared

false → unlocked , true → locked

P_i

```
while(1) {
    acquire(&lock);
    CS
    release(&lock);
    RS
}
```

acquire(&lock); → check if the lock is available, if not, it waits, else it sets lock = true;

Must be atomic ⇓

They must be either executed completely or not executed at all.

(a) Disabling & Enabling Interrupts:-

Pi

```
while (1) {
    DI
    [CS]
    EI
    [RS]
}
```

⇒ we are disabling interrupts before the CS. So in the sense, we are making CS execute like it's atomic.

↓

But not so good.

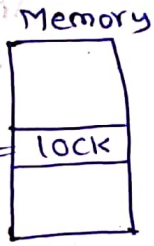
(b) Test-and-set Instruction:-

TAS R_1, 0(1000)

|||

$R_1 \leftarrow M[1000]$

$M[1000] \leftarrow 1$



As a pseudocode,

```
boolean test_and_set(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

[lock = FALSE] shared

Pi

```
while (1) {
    while (test_and_set(&lock));
    [CS]
    lock = FALSE;
    [RS]
}
```

→ can be used for any number of processes.

(c) Compare and swap instruction;

Ex: CAS $R_1, R_2, 0(1000)$

```
if (R1 == M[1000])
    M[1000] = R2
```
→ Atomic.

~~→ Pseudo code → int CAS~~

~~→ Process Pi~~

~~while(1) {~~

→ pseudo code

M[1000] = *value, R1 = expected, R2 = new

```
int CAS(int *value, int expected, int new)
{
    int temp = *value;
    if (*value == expected)
        *value = new;
    return temp;
}
```

→ Process Pi

```
while (1) {
    while(CAS(&lock, 0, 1) != 0);
    CS
    lock=0;
    RS
}
```

Initially, lock=0.

Called spin lock

S.M.

lock	0

*Spin lock

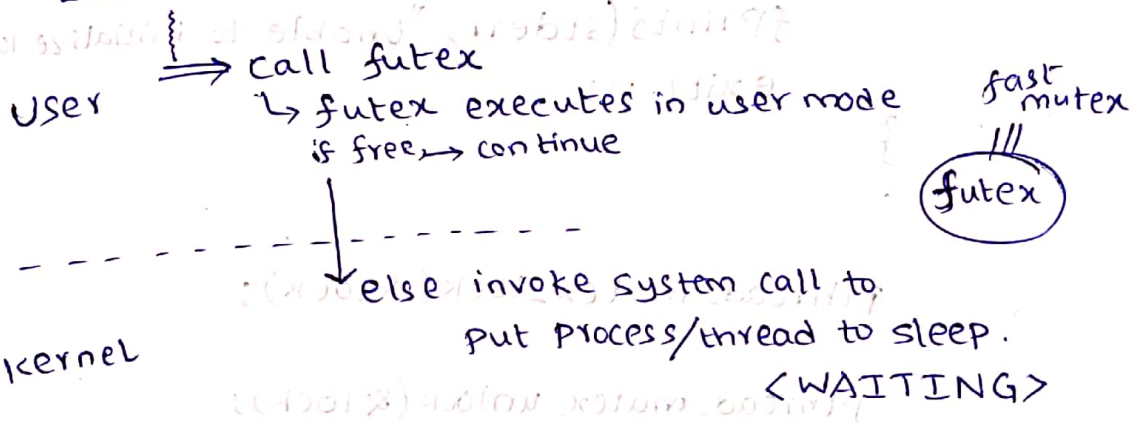
→ The process that is waiting in the entry section is actually running in a loop ⇒ consuming CPU cycles.

* Mutex locks :-

→ simple mechanism for locking & unlocking.

- You require one of the special hardware instructions
- You may or may not wait on spin lock if the CS is occupied.

→ Typical mutex implementation that avoids ~~avoids~~ spinlock



→ Spinlocks

→ Draw back :- → It wastes CPU ~~cycl~~ time.

→ Advantages :

(a) It avoids context switch during waiting. → To put the process to waiting state.

(b) May be good if the CS code is fairly small.

* acquire() and release() :-

Pi

```
while (1) {
    acquire(&lock);        → Spinlock or context swith may occur.
    [CS]
    release(&lock);
    [RS]
}
```

→ acquire(int *lock)
{
while(*lock == 1);
}

release(int *lock)
{
*lock = 0
}

when using spinlock.

→ Mutex lock for pthread library:

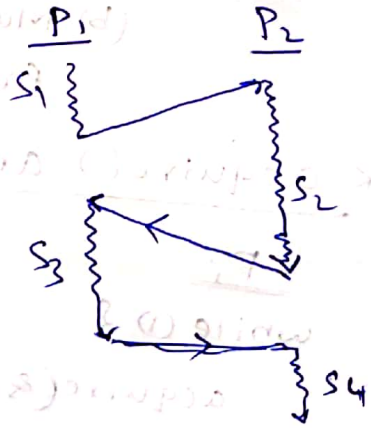
```
pthread_mutex_t lock;

if(pthread_mutex_init(&lock, NULL) != 0){
    fprintf(stderr, "Unable to initialize lock");
    exit(-1);
}
.
.
pthread_mutex_lock(&lock);
.
pthread_mutex_unlock(&lock);
.
.
pthread_mutex_destroy(&lock);
```

* Process synchronization:

(a) Mutual exclusion

(b) Process precedences



(c) Resource counting

count ++ → atomic instruction