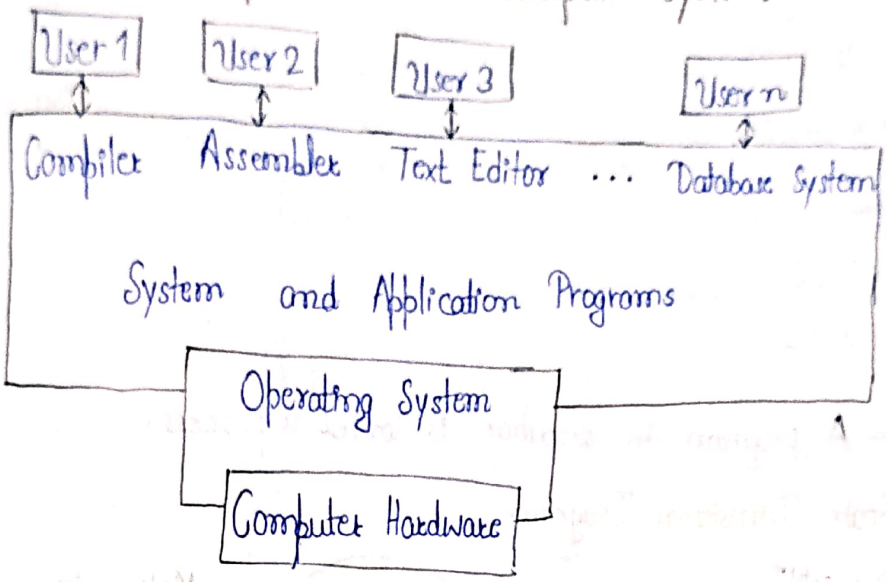
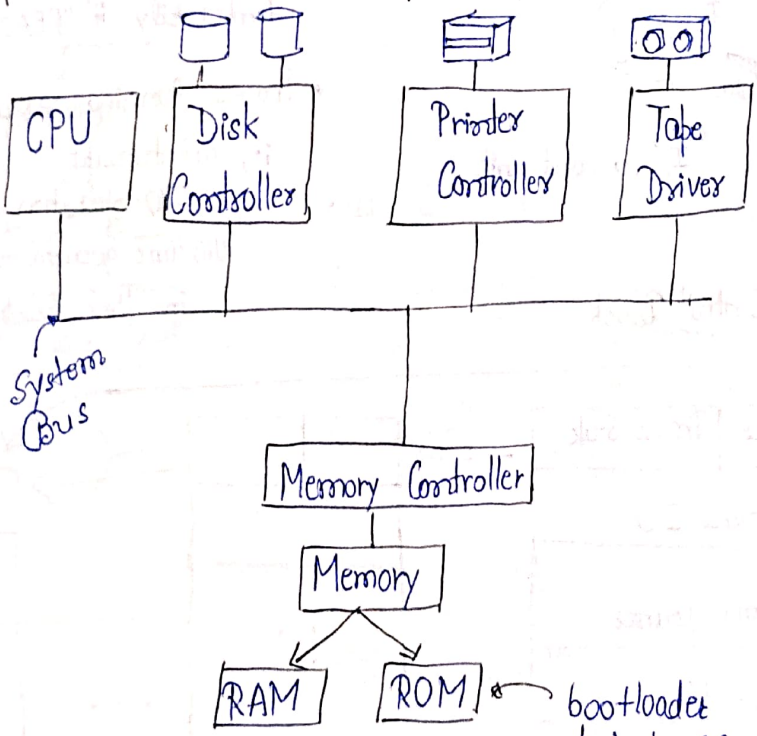
Operating System

* Abstract View of Components of a Computer System

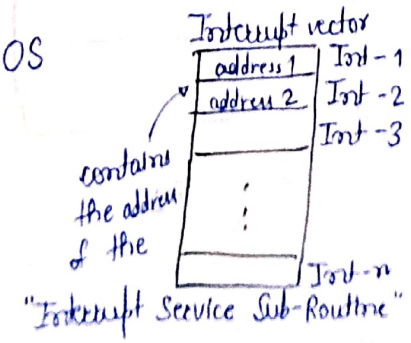


* Components of Modern Computer System



* boot loader
  - ↳ loads OS into RAM (core part of OS → kernel)
  - ↳ initializes different components of system

* Interrupt-Driven OS



OS → Batch OS → Multi-programming OS → Time-Sharing OS → Real Time OS

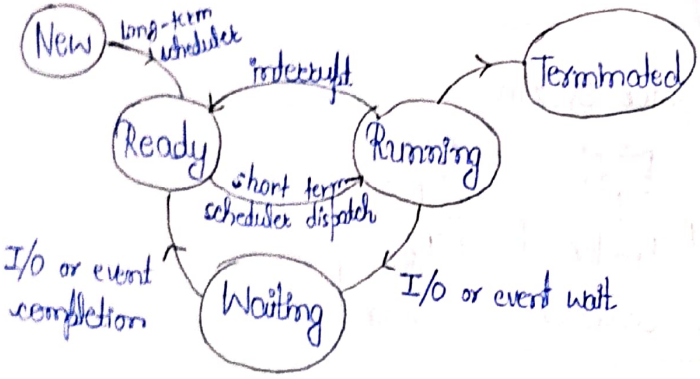
* Context Switching – control switches from one program to another

Types of OS:

- Batch OS
- Multi-programming OS
- Time-Sharing OS
- Real-Time OS
- Distributed OS

* Process – A program in execution is called a process.

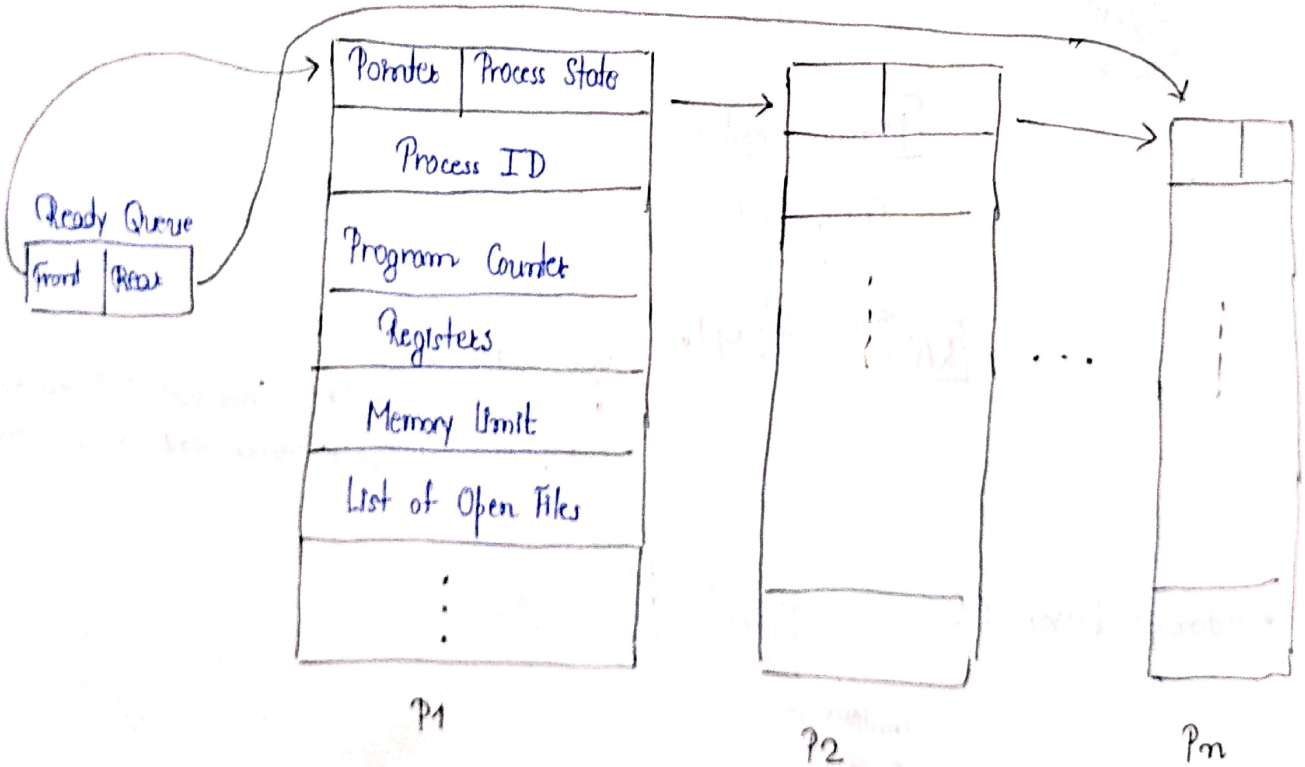
* Process State Transition Diagram



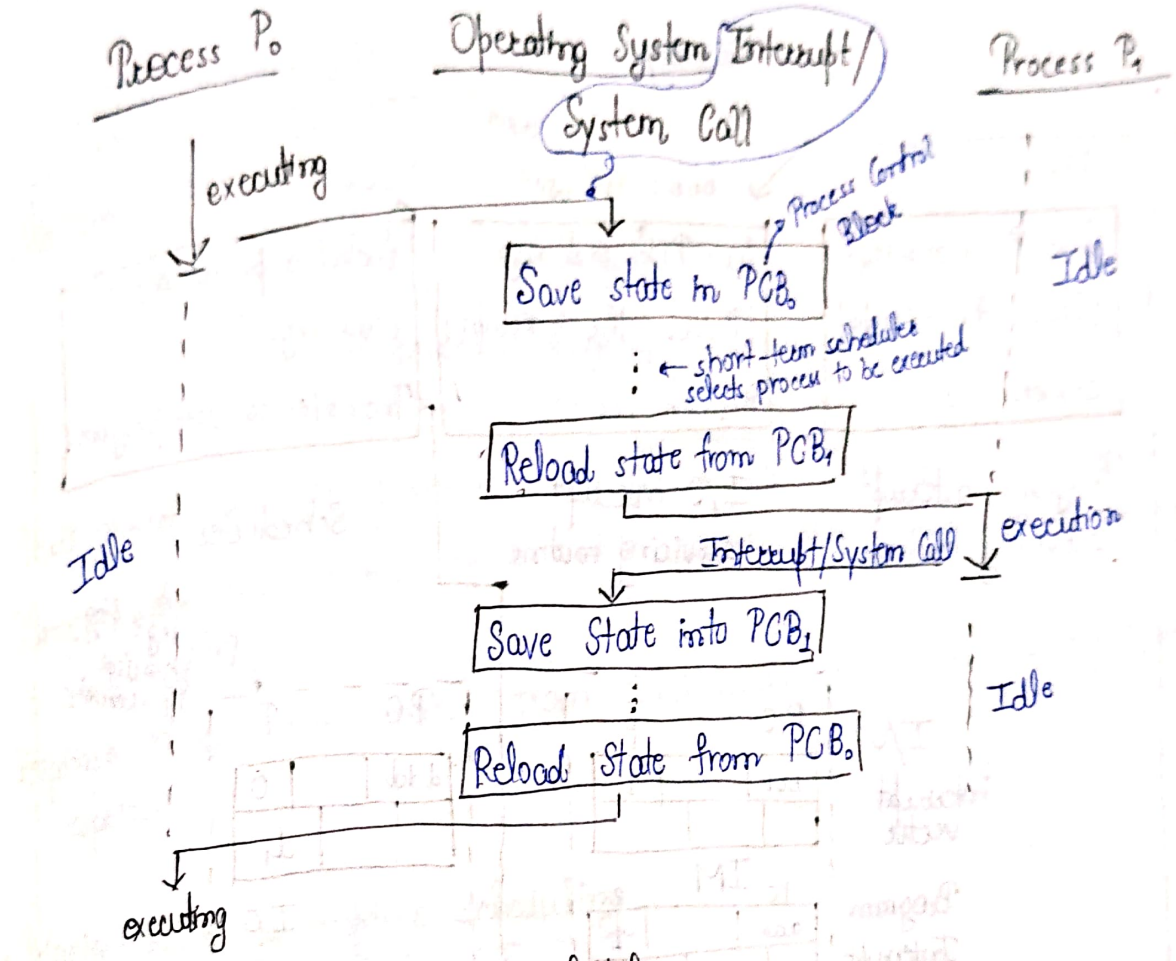
- Medium-term scheduler: swaps out some of the processes temporarily if "READY" is exhausted

* Process: Running → Ready
if, (i) interrupt
(ii) I/O operation
(iii) time quanta over
in Time-Sharing OS

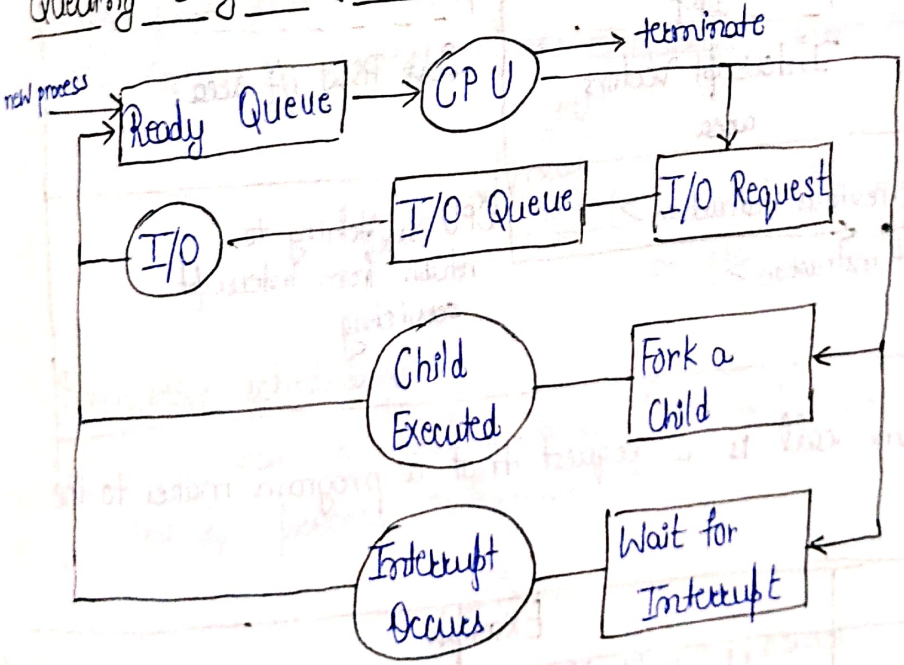
* Process Control Block



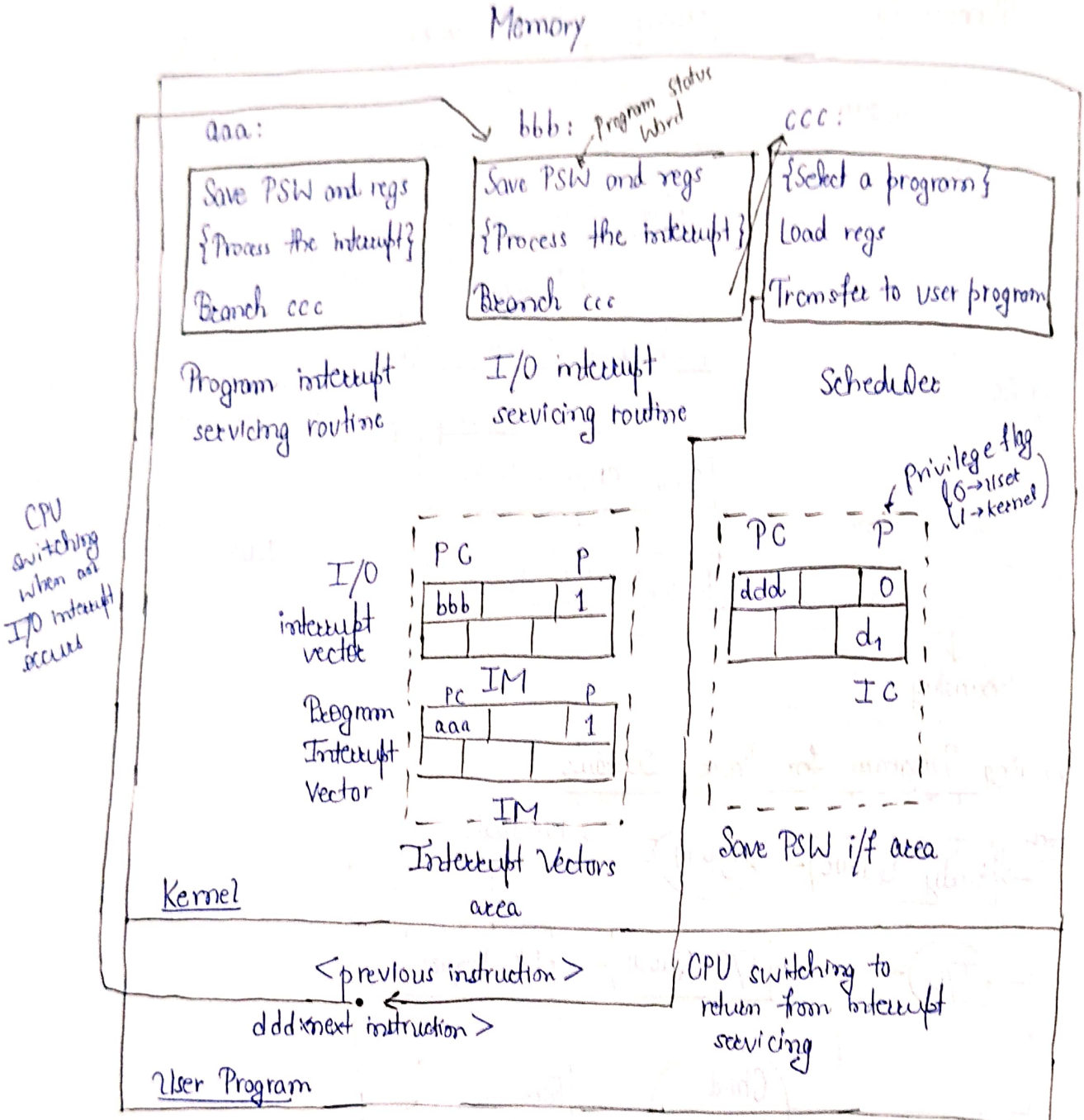
Context Switching



Queuing Diagram for Process Schedule



Interrupt Processing



System Call: A system call is a request that a program makes to the operating system.

Type	Example
Resource	allocate/deallocate resource, check resource available
Program/Process	Execute/terminate program
File	open/close file, read/write file
Information	get time information, O.S. information
Communication	send/receive message through network

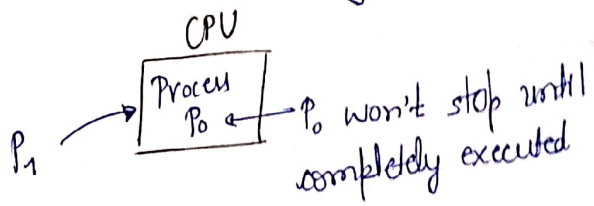
* Multi-Programming
  . CPU-bound program
  . I/O-bound program

* Medium Term Scheduler
  . Controls degree of multi-programming
    no. of processes running simultaneously

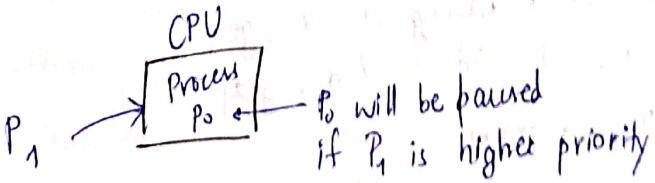
. Throughput: no. of processes completed by CPU in unit time
. Turned-around time: total time taken from submission of a process to its completion
. Response time: time taken from submission of a process until the first response.

* Priority-based pre-emptive scheduling

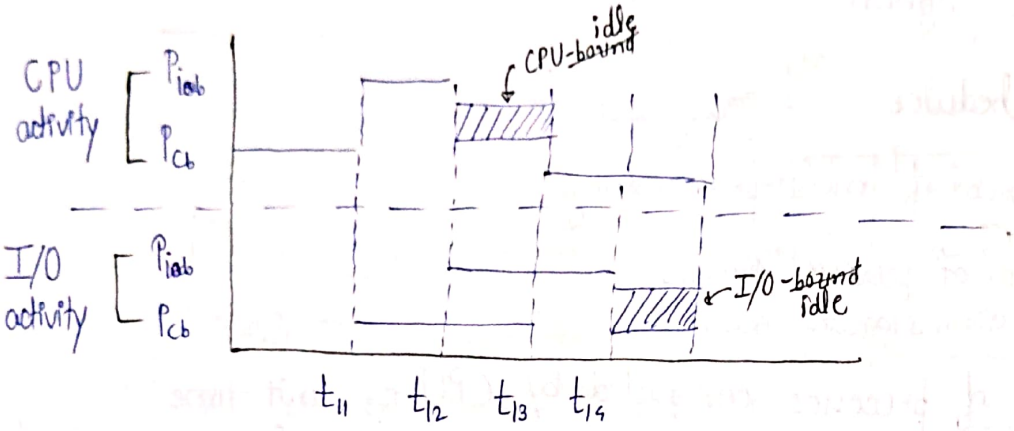
. Non-preemptive scheduling
  – Once a process starts executing, it runs to completion



. Preemptive scheduling
  – CPU can be taken away from a process mid-execution if a higher priority process interrupts it



Timing Chart when CPU-based bound program has higher priority



Here P_{cb} has higher priority than P_{iob}

P_{cb} ↑ CPU-bound

P_{iob} ↑ I/O-bound

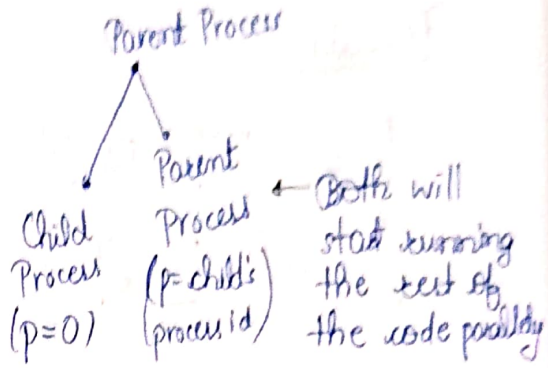
Operation on Process

→ Creation of child process using fork() system call

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/wait.h>
int main(){
    pid_t p;
    printf("Before fork \n");
    p= fork();  ← system call that creates a new process by duplicating the calling process
    if (p==0){  //child process
        printf("I am child having id %d", getpid());
        printf("My parent's id is %d", getppid());
    }else{          //parent process
        wait(null);  ← parent process will wait until child process has finished
        printf("My child's id is %d \n", p);
        printf("I am parent having id %d \n", getpid());
    }
    printf("Common\n");
}   // End of main function
```

Output:

```
Before fork
I am child having id 20
My parent's id is 15
Common
My child's id is 20
I am parent having id 15
Common
```

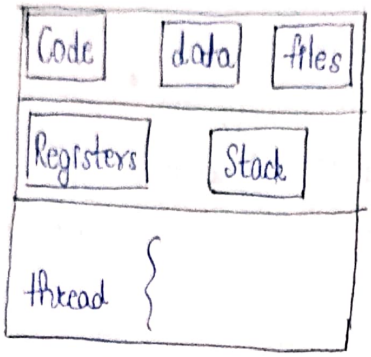


- If a child process is intended to perform some subtask on a system with multiple CPUs, then execution will be sped up as compared to using a function call to perform the subtask
- Child's process id is generally greater than the parent's process id (because it is created later)

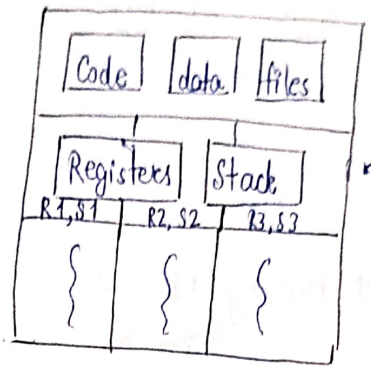
→ Inter Process Communication using Shared Memory

```
#define buffer_size 10
typedef struct{
    - - -
    - - -
} item;
item buffer[buffer_size];
int in=0;
int out=0;
while(1){   ← Producer Process
    /* Produce an item in next Produced*/
    while(((in+1)%buffer_size)==out){
        }   // do nothing
    buffer[in]=nextProduced;
    in=(in+1)%buffer_size;
}
while(1){   ← Consumer Process
    while(in==out){
        } // do nothing
    nextConsumed=buffer[out];
    out=(out+1)%buffer_size;
/* Consume the item in next consumed */
}
```

Threading



Single-thread



Multi-thread

- Thread ~ "light-weight process"
- A thread is the smallest unit of execution within a process

Benefits of Threading:

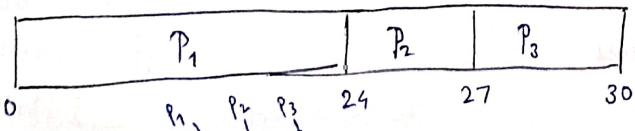
- Responsiveness
- Resource Sharing
- Economic
- Utilization of multi-processor

CPU Scheduling Techniques

- First come, first served
- Shortest Job First
- Priority
- Round-Robin

(for round-robin) Time Slice = 4	Process	(for priority scheduling) Priority	Burst Time ← total time a process needs the CPU for execution without any interruption
	P_1	2	24
	P_2	4	3
	P_3	1	3

FCFS:



Gantt Chart

Avg. Waiting Time: $\frac{0+24+27}{3} = 17$ (total time a process spends waiting in ready queue before it gets CPU for execution)

Avg. Turned-around Time: $\frac{24+27+30}{3} = 27$

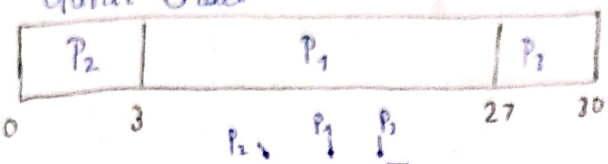
(total time taken from the moment a process arrives in the system to the moment it completes execution)

SJF: Gantt Chart

Avg. Waiting Time = $\frac{0+3+6}{3} = 3$

Avg. Turned-around Time = $\frac{3+6+30}{3} = 13$

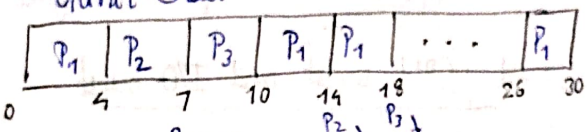
Priority: Gantt Chart



Avg. Waiting Time = $\frac{0+3+27}{3} = 10$

Avg. Turned-around Time = $\frac{3+27+30}{3} = 20$

Round-Robin: Gantt Chart



Avg. Waiting Time = $\frac{(0+6)+4+7}{3} = \frac{17}{3} \approx 5.67$

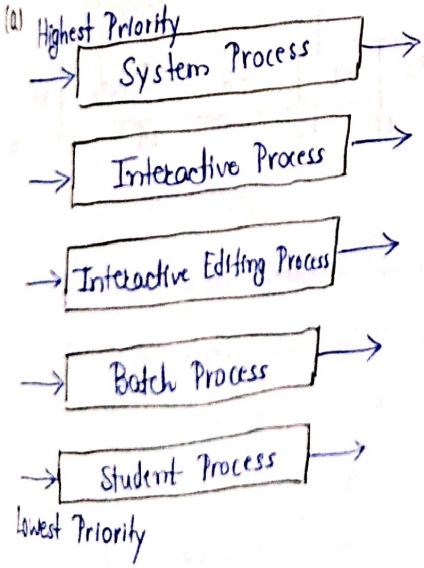
Avg. Turned-around Time = $\frac{30+7+10}{3} = \frac{47}{3} \approx 15.67$

CPU Scheduling Algorithm

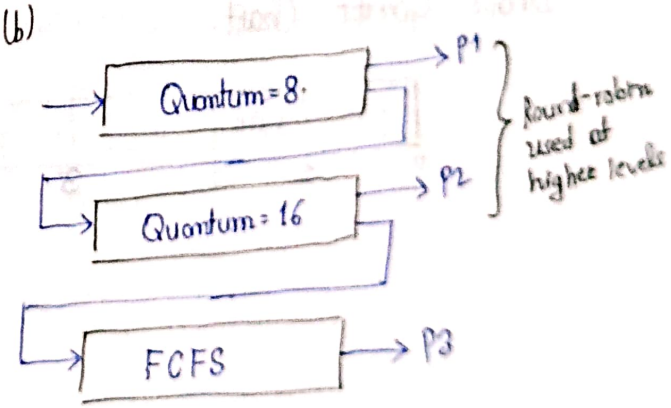
(a) Multi-Level Queue

(b) Multi-Level Feedback Queue

(a)



(b)



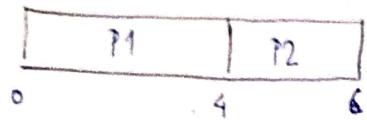
P1 6

P2 12 4

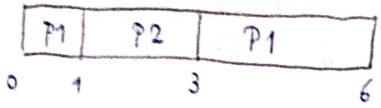
P3 30 22 6

	Arrival Time	Priority	Burst Time
P1	0	2	4
P2	1	4	2

Non-preemptive :



Preemptive :



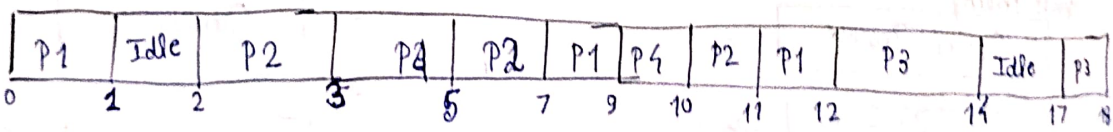
* Starvation: Lower priority processes do not get access to the CPU as higher priority processes keep coming.
* Aging : After waiting for some time, the process moves up to a higher priority queue.

CPU Scheduling Problem with mixed CPU burst and I/O burst

Process	Arrival Time	Priority	CPU	I/O	CPU
P1	0	2	1	5	3
P2	2	3	3	3	1
P3	3	1	2	3	1
P4	3	4	2	4	1

Model: Pre-emptive Scheduling-Priority

Draw Gantt Chart.



Process Synchronization

- producer – consumer problem:

```
Item buffer[Buffer_Size];      int counter=0;
int in=0, out=0;
```

Producer:

```
while (1){                          counter == Buffer_Size
    while ((in+1) % Buffer_Size) == out)
        ;
    buffer[in] = nextProduced;
    in = (in+1) % Buffer_Size;
    counter++;
}
```

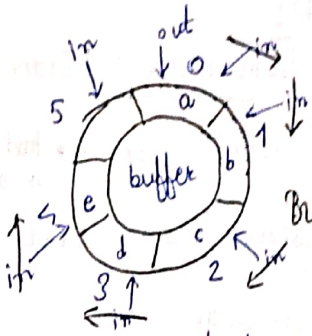
Consumer:

```
while(1){                           counter==0
    while (in == out)
        ;
    nextConsumed = buffer[out];
    out = (out+1) % Buffer_Size;
    counter--;
}
```

critical section

// Entry Section

// Exit or Remainder Section



Buffer_Size = 6;

- Only (Buffer_Size − 1) elements can be stored
- If we do (in == out) for producer as well, we won't be able to distinguish b/w empty queue and full queue
- Using another variable "counter" to keep track of the no. of elements in the queue, we can utilise the 1 wasted slot

Producer:

```
reg1 = counter
reg1 = reg1 + 1
counter = reg1
```

Consumer:

```
reg2 = counter
reg2 = reg2 − 1
counter = reg2
```

- If both run sequentially, counter remains unchanged. (no issues)
- If both run concurrently (one possibility):

T_0 : Producer	reg_1 = counter;	(5)	reg_1
T_1 : "	$reg_1 = reg_1 + 1$;	(6)	reg_1
T_2 : Consumer	reg_2 = counter;	(5)	reg_2
T_3 : "	$reg_2 = reg_2 - 1$;	(4)	reg_2
T_4 : Producer	counter = reg_1;	(6)	counter
T_5 : Consumer	counter = reg_2;	(4)	counter

counter: 5 → 6 → 4

(we run into issues)

- This problem is called "Race Condition" (happens when multiple processes run concurrently sharing data)

- Part of code where shared resource is utilised, it is called "critical section".
- The idea is to not allow another process to enter its critical section, while one process is executing its critical section

Semaphore

s = 1

wait(s) → waits if critical section is running

signal(s) → increments 's' if critical section isn't running

Semaphore

```
int s=1;
wait(s){
    while (s<=0)
        ; //No operation
    s--;
}
signal(s){
    s++;
}
```

(wait and signal: atomic)

* Mutual-exclusion implementation with sephamore

```
do{
    wait(mutex); // Entry Section
    critical section; → Part of code where shared data is accessed
    signal(mutex); // Exit Section
    remainder section;
}while(1);
// Code of Process: Pi
```

* Let $\{P_1, P_2, \ldots, P_k\}$ processes want to access a shared data or resource, for the shared data there is a semaphore variable "mutex" which is initialized to 1.
* wait(s) and signal(s) should be "atomic" functions, i.e the function when called should be executed in its entirety or not at all, no in-between interruption.
* Disadvantage: if s<=0, the while loop will be constantly checking the condition, taking CPU resources (BUSY WAITING)

Solution:

```
typedef struct {
    int value;
    struct process * L;
}Semaphore;
void wait(Semaphore S){
    S.value--;
    if (S.value <0){
        add this process to S.L;
        block();
    }
}
```

(L: list of waiting processes)

```
void signal(Semaphore S){
    S.value++;
    if (S.value <=0){
        remove process P from S.L;
        Wakeup(P);
    }
}
```

Synchronization Hardware

```
boolean Test_And_Set (boolean target){
    boolean rv = target;
    target = true;  // modifies the original
    return rv;      // value of target not
                    // a copy
                    (use address to modify
                     actual target)
}
```

atomic instruction (NOT a function, CPU instruction)

```
do{
    while (Test_And_Set (lock));  // Entry Section
    critical section;
    lock = false;  // Exit Section
    remainder section;
} while (1);
```

// Mutual-exclusion Implementation with Test_And_Set

- Another technique to overcome race-around condition — "Synchronization Hardware"
- Solution to Critical Section Problem satisfies three requirements:

i) Mutual Exclusion — only one process should access critical section at a time

ii) Progress — the processes not executing remainder section decide which process enters critical section next (decision made by processes who're not in remainder section)

P4 →
P1 → Entry Section
P2 → Critical Section
Exit Section
P3 → Remainder Section

iii) ~~Bounded~~ Hardware Waiting

↳ the processes should wait finitely

- If there are n processes, and 1 fails to get the CPU, it has to wait at most for (n-1) processes to execute.
- Synchronization Techniques have no control over which process enters the CPU next, so a process may end up waiting infinitely.

```
do{
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
        key = TestAndSet(lock);
  waiting[i] = false;
    // Critical section
  j = (i+1)%n;
  while ((j != i) && !waiting[j])
        j = (j+1)%n;
  if (j == i)
      lock = false;
  else
      waiting[j] = false;
    // Remainder section
} while(1);
```

Entry Section

Exit Section

Common data structure used by $P_0, P_1, \ldots, P_{n-1}$

- boolean waiting[n];
- boolean lock;

// Code for a process P_i