

1.

Operating Systems : Internals & Design  
Principles by William Stallings

2.

Operating System Concepts - by Galvin

OS -

Transistors - CMOS

↳ are connected to logic gates.

Gates are used for logical operations

VLSI

↳ Organization → Memory units, I/O units

↓  
OS

↓

Application

H/W Abstraction

printf ("Hello World"), "HelloWorld" register  
RAM ← Processor  
fetch

Color,  
coordination + Graphics  
depth Card → Monitor

Scheduling

## App1 / App2

OS keeps all applications isolated in their respective environments.

- I/O Access
- File "
- Program Execution (libraries, utilities)
- Error Detection

15/11/25

## Evolution of OS

1. Serial Processing
2. Simple Batch Processing
3. Multi-programmed batch processing
4. Time-Sharing

### Serial Processing (1940s - 1950s)

- Program directly interacted with the processor.
- Display lights, toggles, switches

H/W → display lights  
→ switches  
board ↘  
Scheduling

- ① Scheduling → 3 jobs  
↳ arrival time  
more priority \* Job 1 → 12 o'clock →  
Job 2 → " → 12.01 sec

Since job 2 has higher priority.  
It will switch to executing job 2.

User 1 → 40 units  
(12 - 12:40)

User 2 → waiting

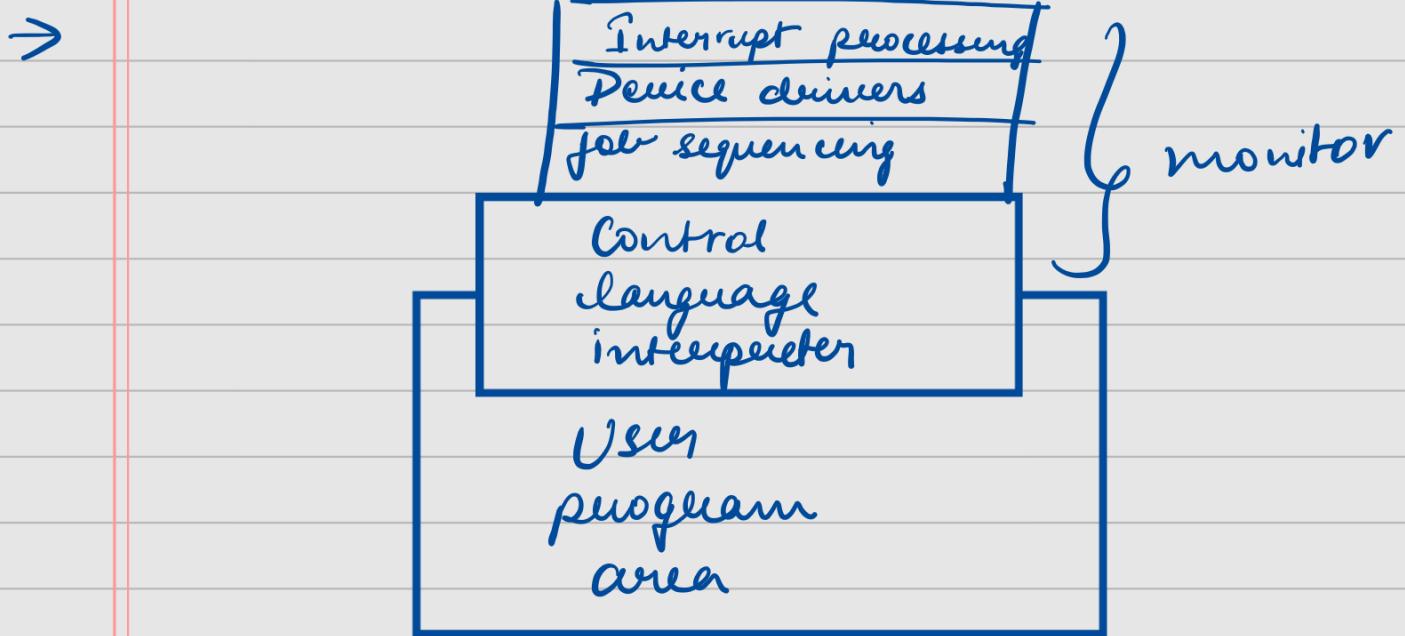
All processes of execution of program  
→ loading, compiling, linking

- ② Setup time

## Simple Batch System (mid 1950s)

- monitor → a piece of software
- user no longer had direct access to processor.

→ tape (input device) → deploy the program on the tape



→ I/O processing is time-consuming.

Job +

15 ms ← Read value 'n' → I/O  
2 ms ← Compute factorial (n) ↗ CPU  
15 ms ← Print factorial (n) → I/O

$\frac{2}{30}$

### Modes of Operation

- User Mode - access to some parts of memory
- Kernel Mode - access to entire memory.

### (3) Multi-processing

5ms      2ms      5ms

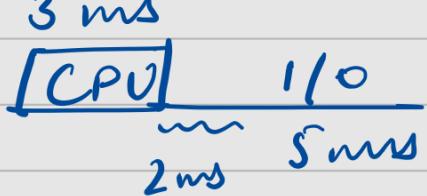
Job +

I/O      CPU      I/O

12 ms

CPU is idle

Job 2



$$3 + 5 = 8 \text{ ms}$$

$$5 + 5 = 10 \text{ ms}$$

Efficient & Minimize response time.

(4)

Time-Sharing



'n' processes  
↓  
 $1/n$

Time Slicing

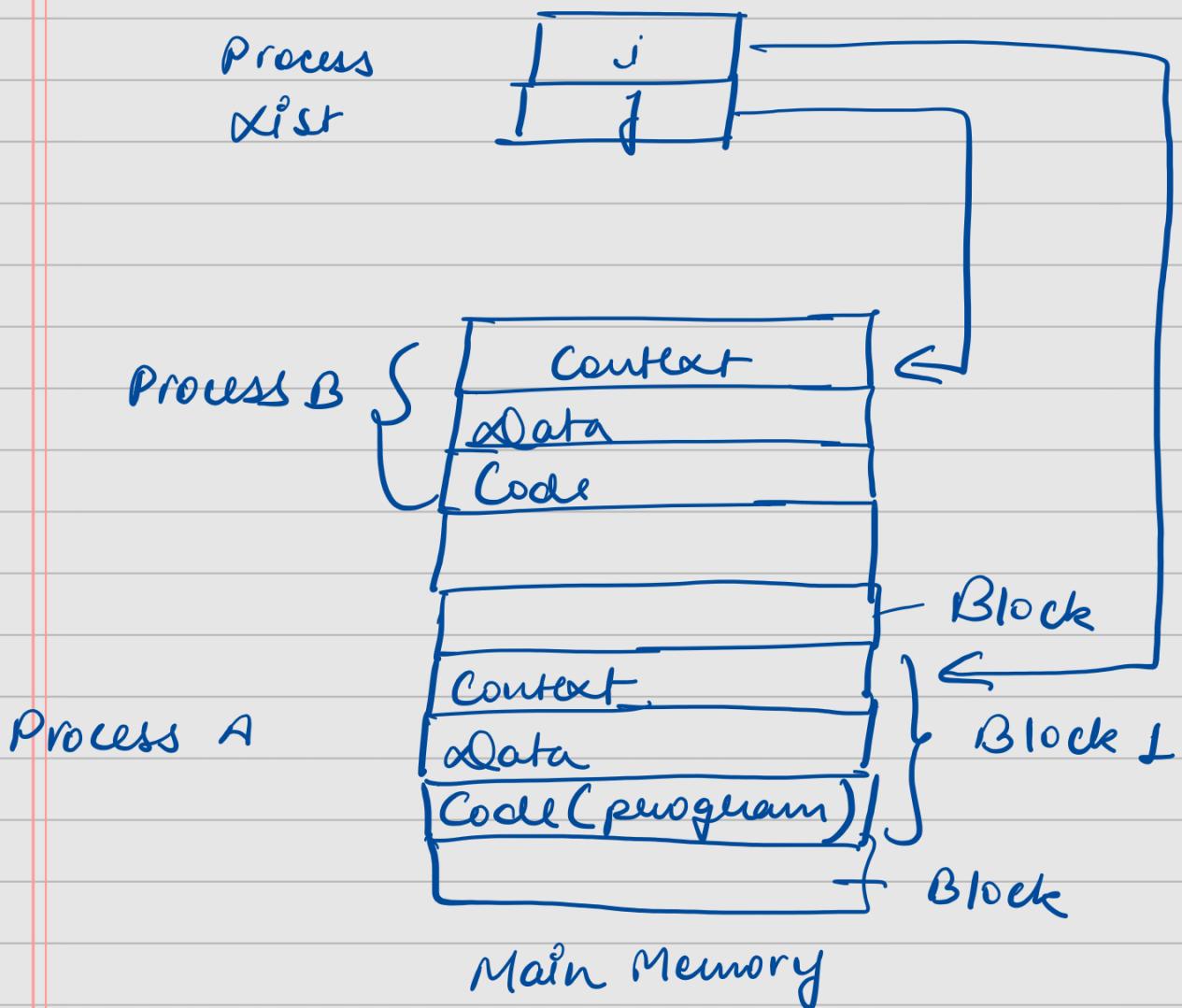
- \* Dividing the time of utilization of CPU
- \* Allow users to specify priority of a job.
- \* Additional storage required for both.

21/1/25

Processes

A      B  
 ↓            ↓  
 block        block of memory  
 of memory

A { (prog, data, )  
 content }

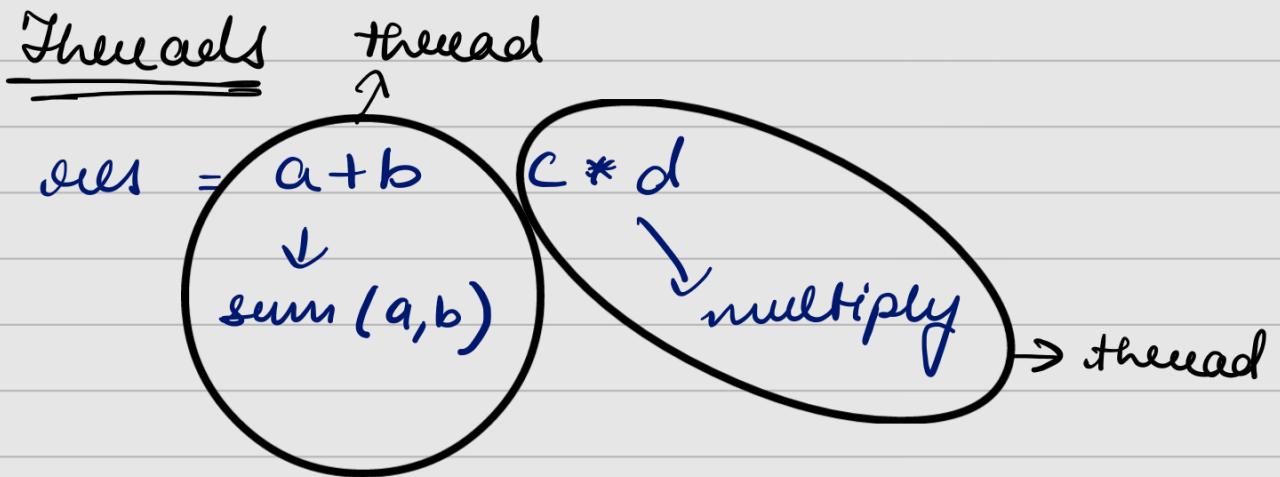


**Process**  
 Starting address of the process it is currently executing  
 L  
**Process Index** → Pointer to the process list for the processes its currently running  
**Base Register**  
**Limit Register**  
 ] stores the size (20 bytes)

**Context Switching** - Whenever there are 2 process executing and if one of them stalled, the info of the state at that time is stored. This phenomenon of switching b/w task will preserving the info of the state of the process.

- Memory is allocated to each process.

States of process  $\Rightarrow$  Ready, Running, Wait / Stop



- Processes are not co dependent.
- They can be executed parallelly.
- Improves efficiency.

## Memory Management

- Process Isolation
- Automatic allocation & Management
- Support for modular programming
- Access control
- Long - Term storage

## Information Security

CIA  $\longrightarrow$  Authenticity  
 $\swarrow$   $\searrow$   
 Confidentiality

Integrity



Preserving the data

Sender → Reciever

## Scheduling & Resource Management

1 - Fairness

2 - Differential Responsiveness

3 - Efficiency

① All processes must get their resources at some point of time.

② Giving priority such that, where there are multiple jobs, one job has higher priority than other. The one with higher priority will get more preference to resource access.

③ Maximize throughput, minimize response time

22/1/25

## Process Control Block (PCB)

- |                       |                    |
|-----------------------|--------------------|
| 1. Identifier         | 6. Context Data    |
| 2. State (New, Ready) | 7. I/O status info |
| 3. Priority           |                    |

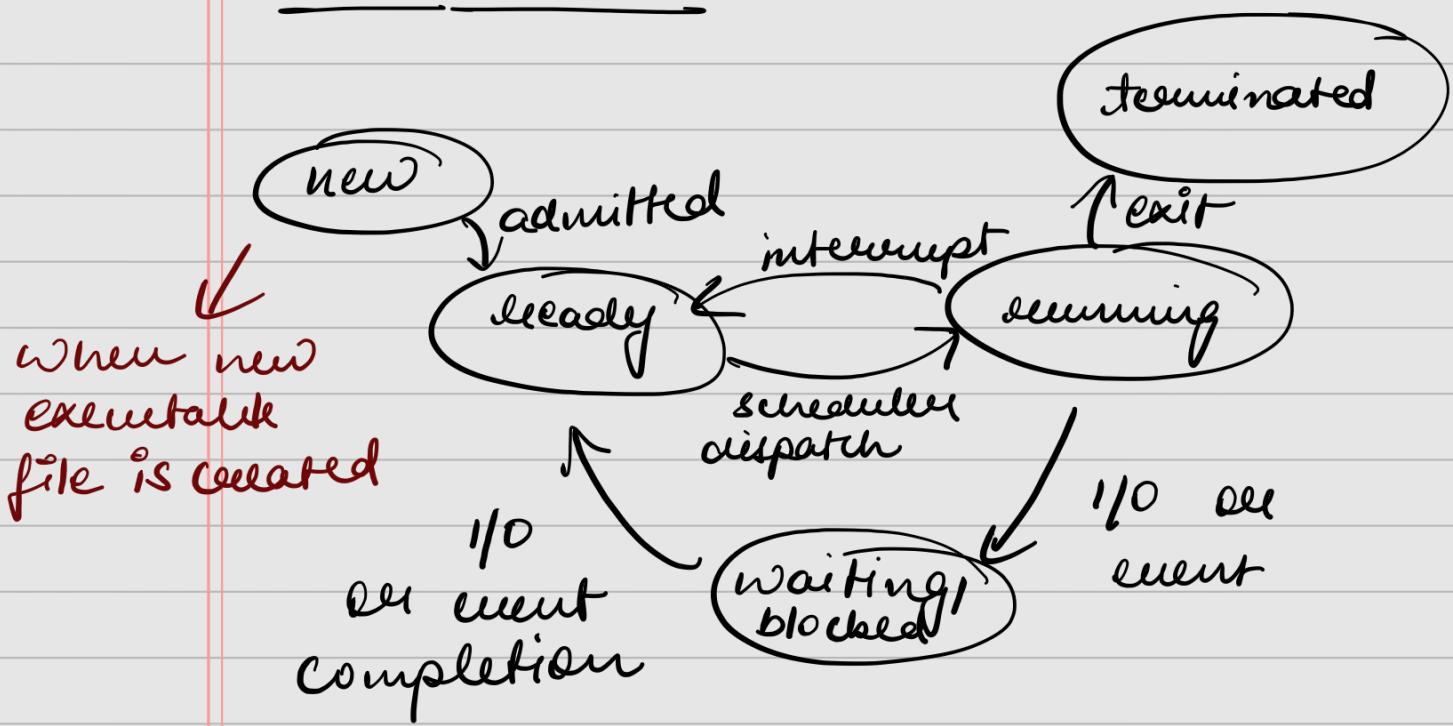
3. Priority
4. Program Counter
5. Memory Pointers

\* PCB are created by the OS for the processes.

### Need for PCB

When OS is handling multiple processes, Using PCB, OS decides on which process to handle.

### 5-State Model



- Only when a program is in the ready state, can it be brought to running state.

program

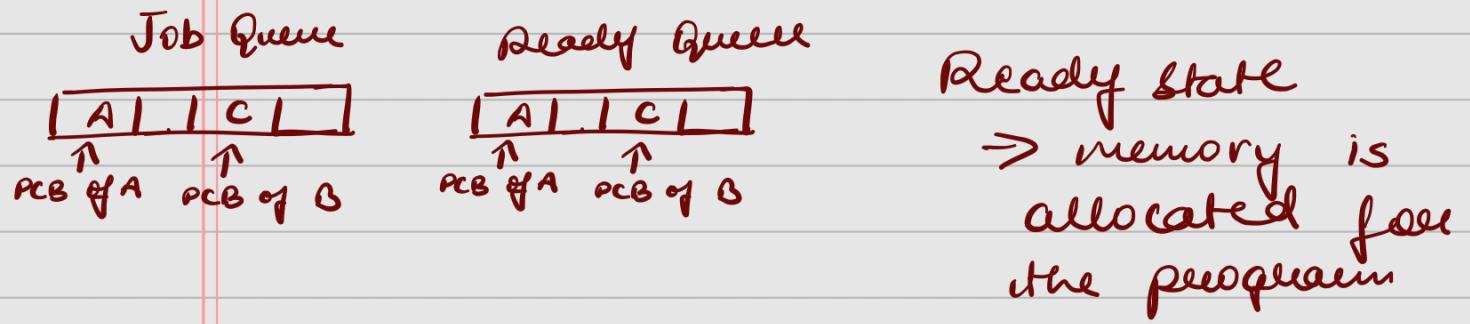
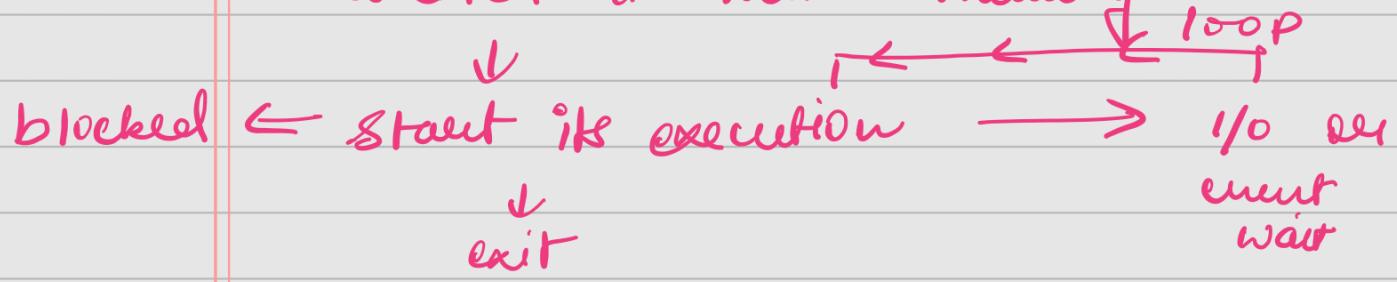


executable file



allocated to main memory

Info inside PCB  
are stored in  
a linked list.

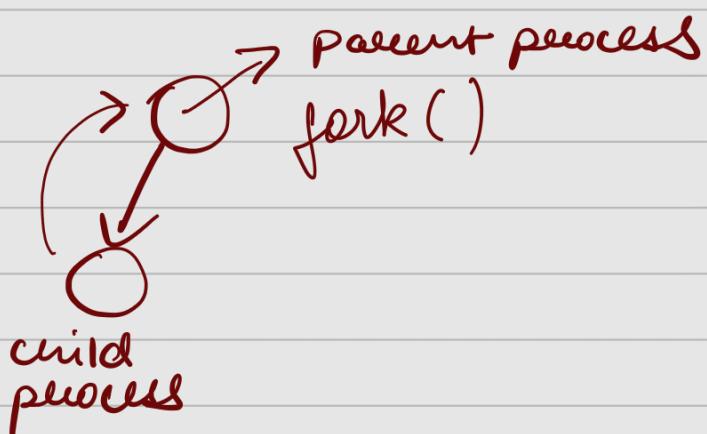


- Job Queue - whenever a job is in new state. The OS will look into the PCB of the processes and then decides to put it in running state using dispatcher
- Ready Queue - ....
- Dispatcher removes the process from the job queue to running state. In running state, it will either go to exit or waiting state.
- Every time a job is moved out from the job queue, the contents of the PCB get modified.
- Every I/O device will have a separate queue for itself, I/O queue.
- On ready state  $\Rightarrow$  job queue

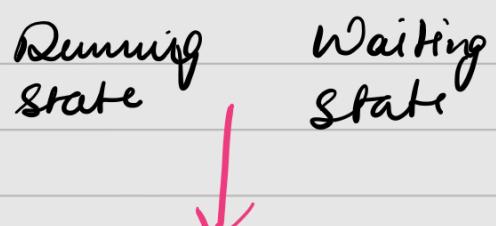
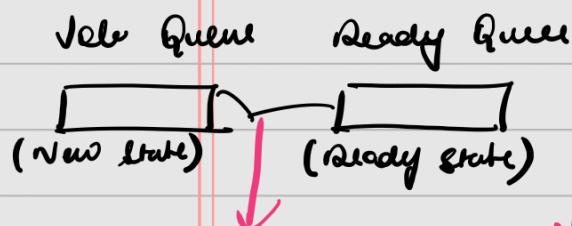
On running state  $\rightarrow$  I/O queue

When a process is running :-

- Call for an I/O operation
- Create a child process
- forcefully removed due to errors



## Schedulers

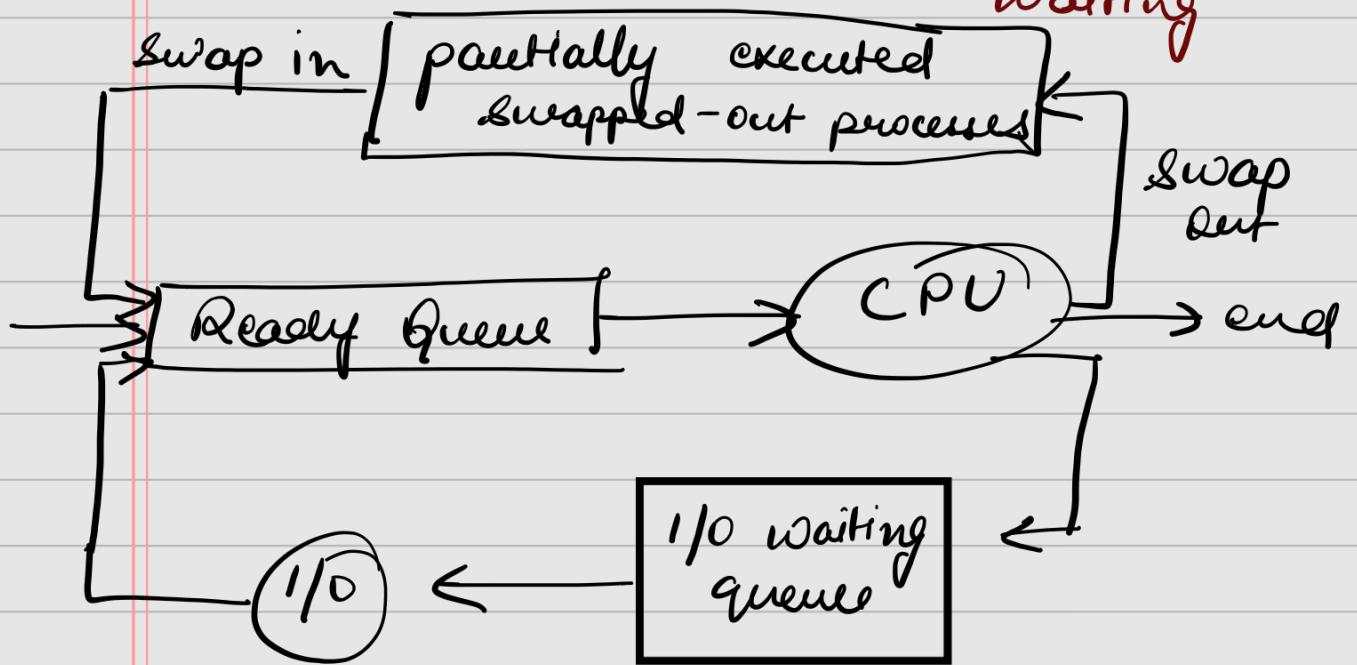


- Take processes from job queue and load them into memory
- Decides the degree of multiprogramming
- i/o activities are more consuming than CPU.

Objective of OS is to have a good balance b/w CPU & I/O bound.

Why?  
If there is too much memory is occupied by the memory.

M.T.S swaps processes b/w running & waiting



## Kernel (Core Component of OS)

- { - User Mode (calculator)
- Kernel Mode (communicate with devices, first move to kernel to give instructions)
  - ↓ additional overhead
  - Frequently shift b/w modes

Switching b/w modes is called context switching. (It saves the current state)

\* States are stored in PCB.

System Call → a programmatic approach of requesting a service from the Kernel.

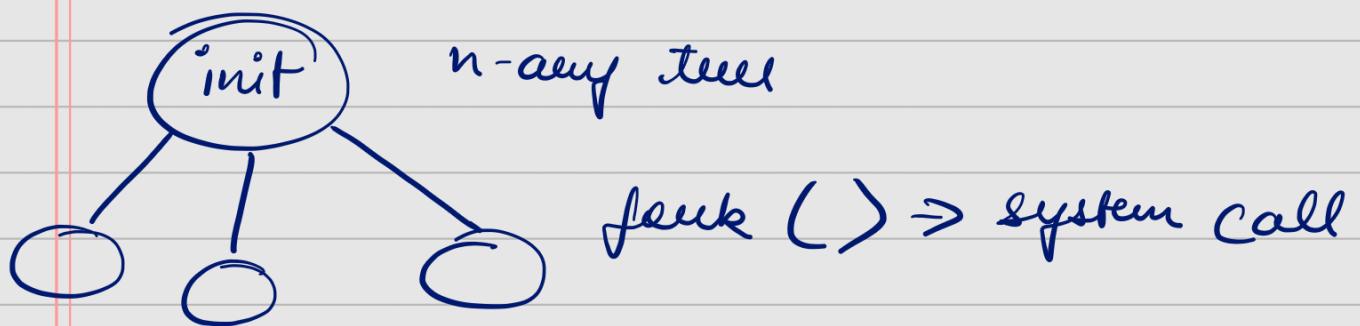


- \* If both processes are in same node, then it's not context switching

~~27/1/25~~

### Process Creation

pid - process id  
init process - pid = 1



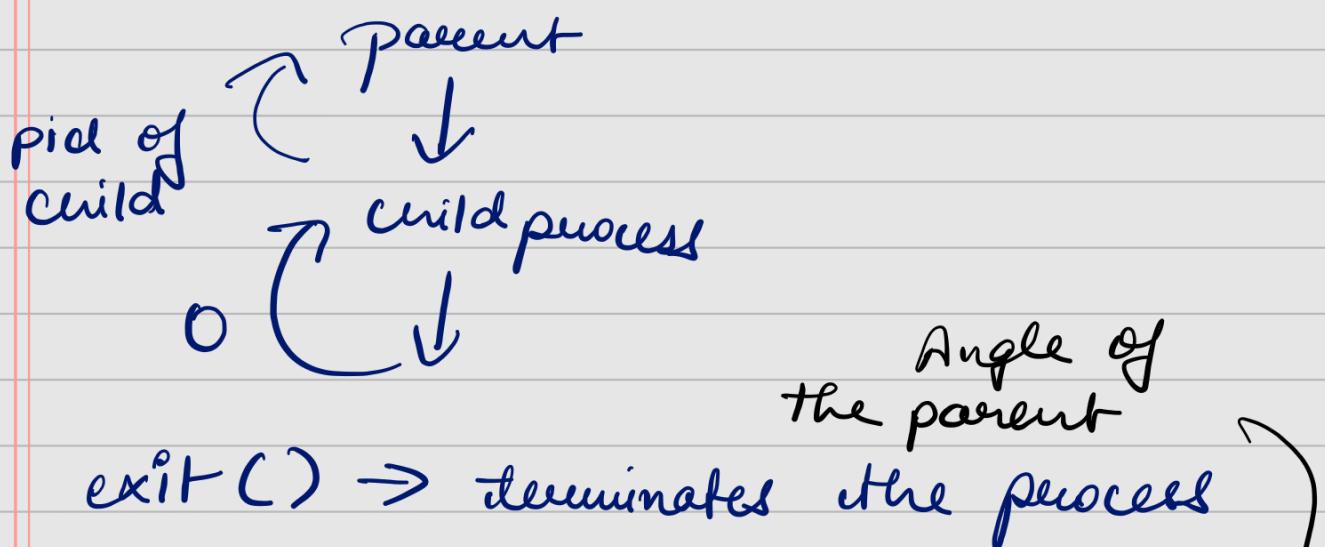
ps  $\rightarrow$  all the processes in the system

ps-cl  $\rightarrow$  all the running processes

### Process Creation Steps

1. Assigning the pid
2. Allocating memory space for the process.
3. Initialising the PCB  $\rightarrow$  transition process

4. [ ] has gone through  
Set appropriate linkages. [ ] after its creation
5. Create other data structures, if needed.



Cascading Termination  $\Rightarrow$

When parent process terminates, the child processes must terminate regardless of their state.

Wait()  $\rightarrow$  The parent waits until the child process finish execution.

$\rightarrow$  Returns the pid of the child that has finished its execution

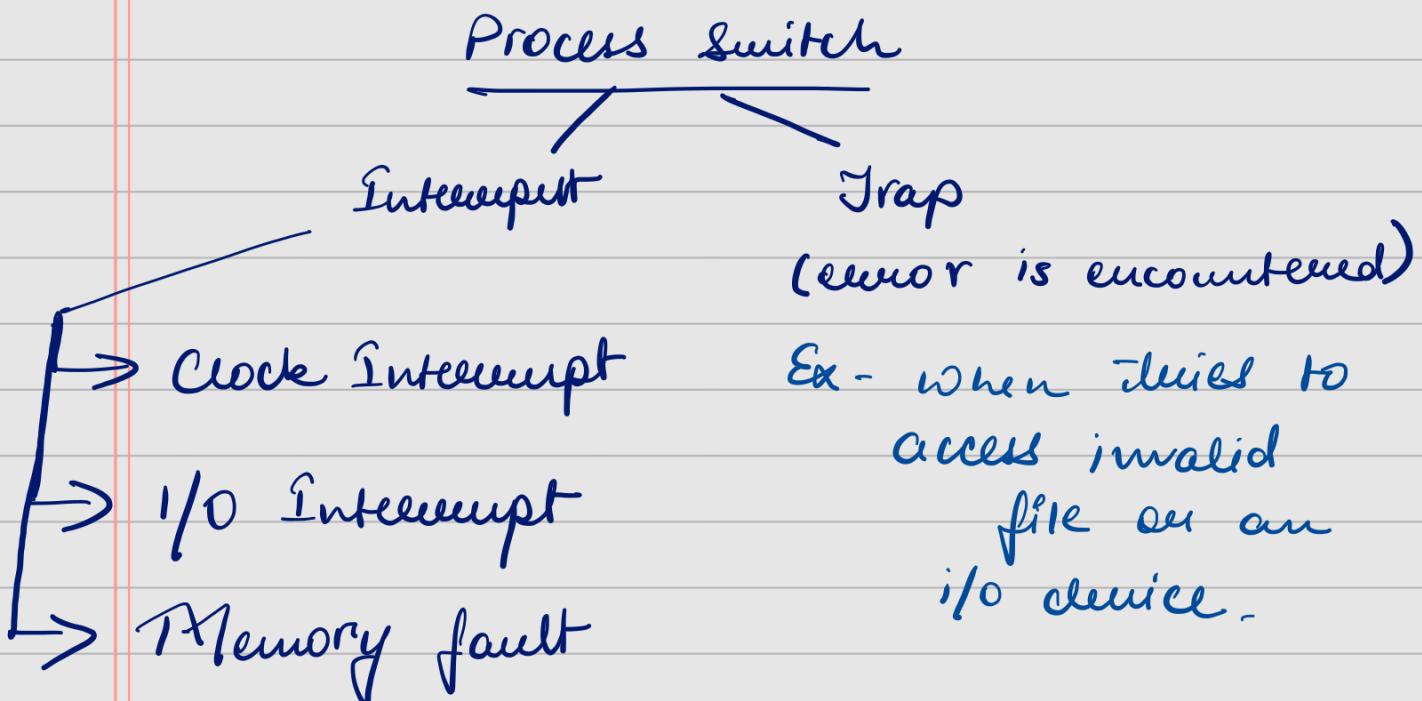
Child wants to terminate first  $\Rightarrow$

- resources given to the child are deallocated by the OS. However the terminating entry still remains in the PCB, but

doesn't call  
a wait ()  
↓  
Orphan  
process

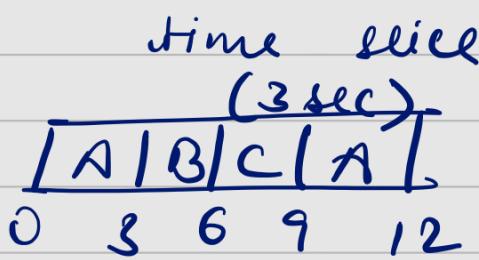
→ Zombie process → process has terminated  
however contents still remain in the  
PCB.  
↳ parent is alive.

Conditions under which process switch



- When multiple processes are running but doing significant task, OS needs to provide equal access to resources based on a time slice.

Time slice - At every particular interval, the access to the resources is shifted to other process, halting the previous process.



Process is partially loaded in main memory and call rest of part from secondary memory to main memory, so it has to wait

28/1/25

Threads - a unit of CPU utilization.



Process A

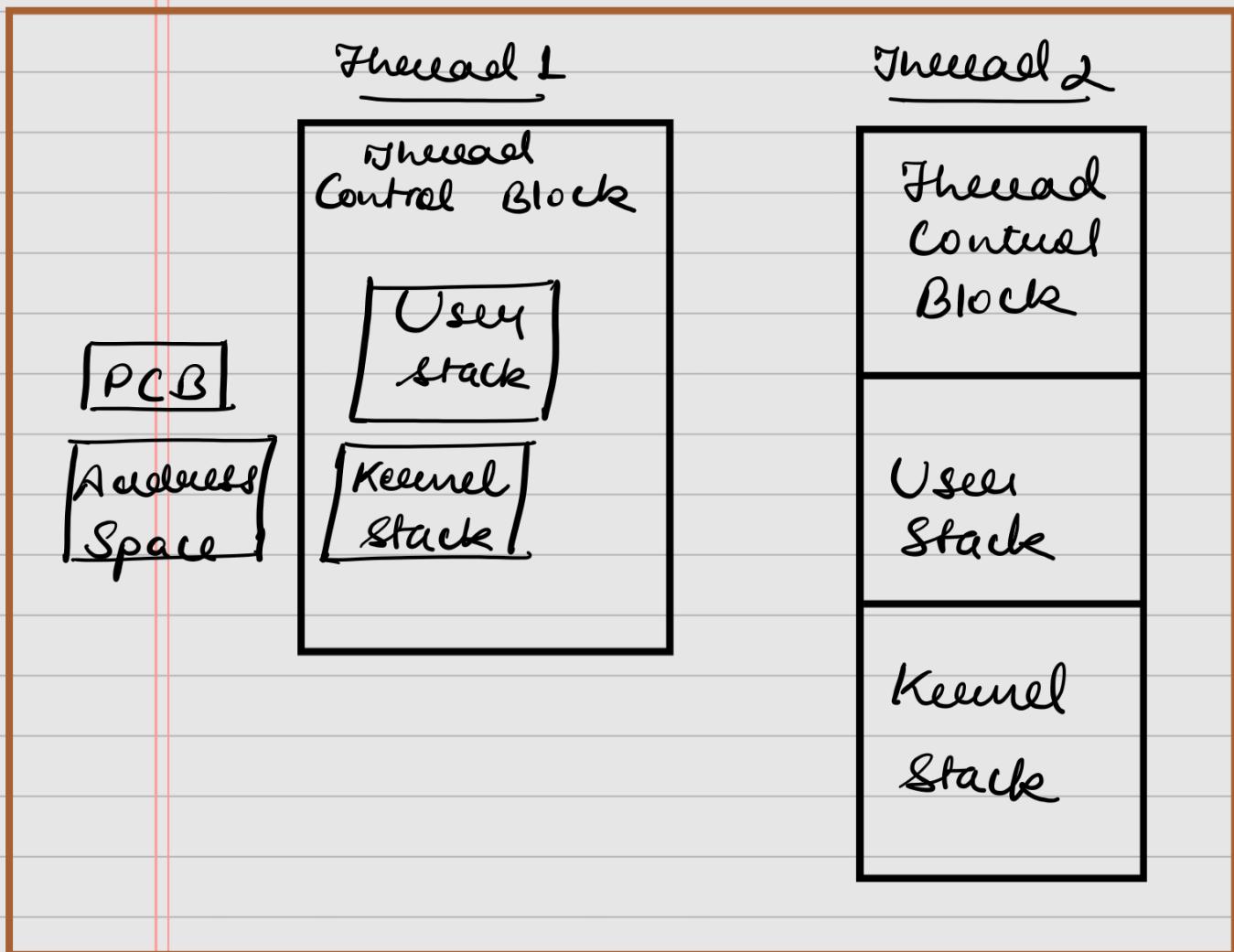
- A process can have multiple threads.

$$x = a + b \rightarrow \text{Thread 1}$$

$$y = a - b \rightarrow \text{Thread 2}$$

$$\text{Output} = x * y$$

Multithreading  $\rightarrow$  ability of the OS to support multiple concurrent execution paths within a single process.



### Benefits

1. Responsiveness
2. Resource sharing
3. Economy
4. Scalability

## Relationship b/w Threads & process

Thread

Process

1 : 1

M : 1 → multiple threads are defined & executed within a single process.

1 : M → One thread may migrate process from one process to another

M : N → multiple threads are defined

## Multithreading Model

User Threads → support is provided at the user level.

Kernel » → support is provided at Kernel level.

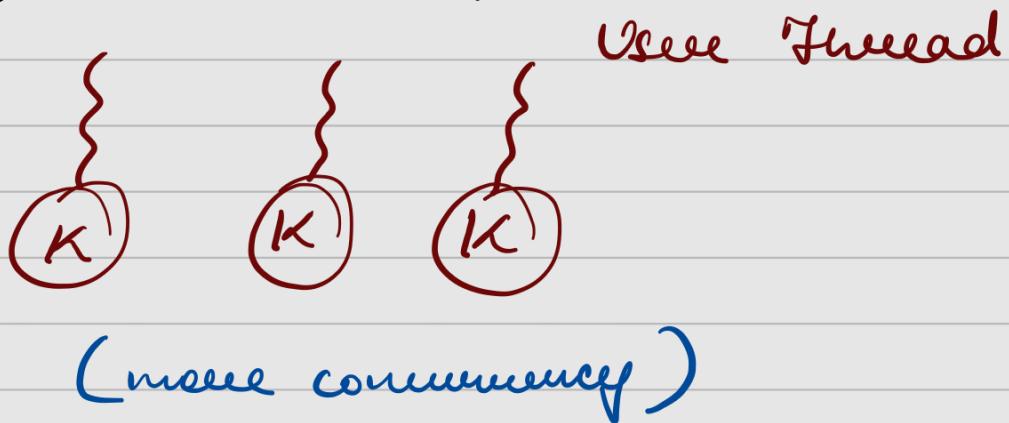


L.) One - Many model

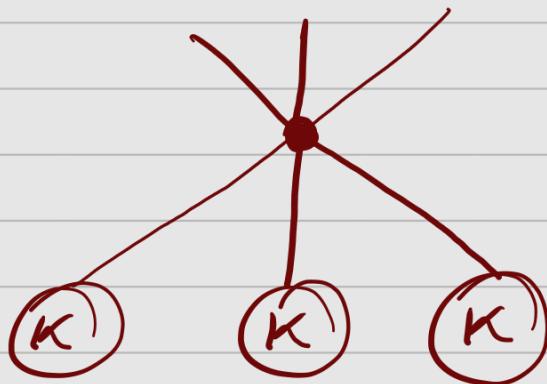


'Kernel-level  
thread

## 2) One-One Model



## 3) Many-to-Many



Every user level thread can access every kernel level thread.

- True concurrency is much possible.

29/1/25 Scheduling

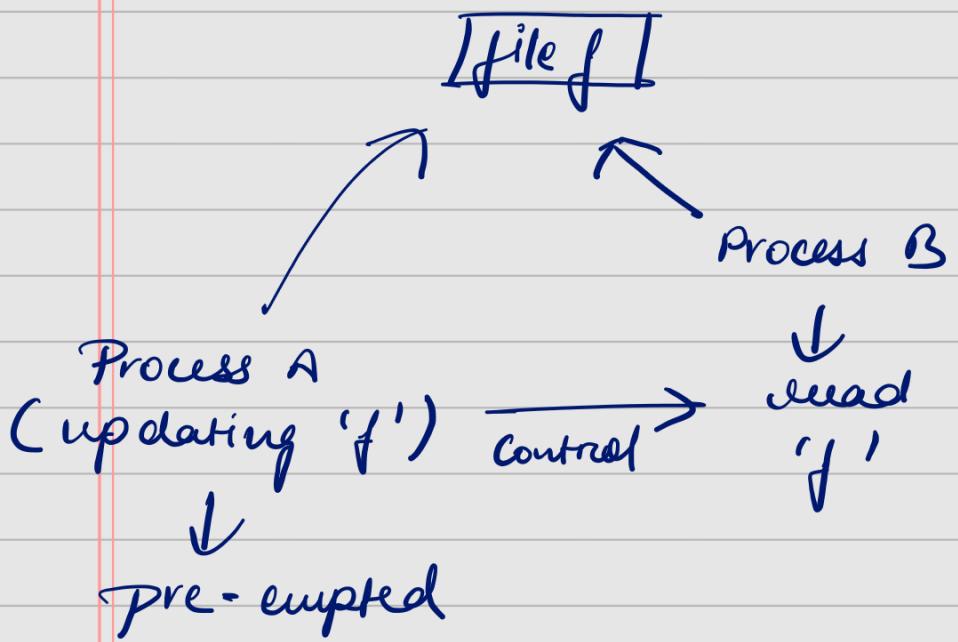
- Preemptive - Interrupts occur (process moves from running to waiting)

- Non-preemptive / co-operative

(non-preemptive / Co-operative  
(voluntarily releasing it)

→ process terminates  
" goes to i/o

### Race Condition



Mutual Exclusion ← Race Condition

### Scheduling Criteria

1. CPU utilization
2. Throughput  $\rightarrow$  no of processes completed per unit time
3. Turnaround Time  $\rightarrow$  Time from Submission to completion of a process.

4. Waiting Time  $\rightarrow$  Sum of all the times that a process is waiting
5. Response Time  $\rightarrow$  minimized

### First-Come First Serve (FCFS)

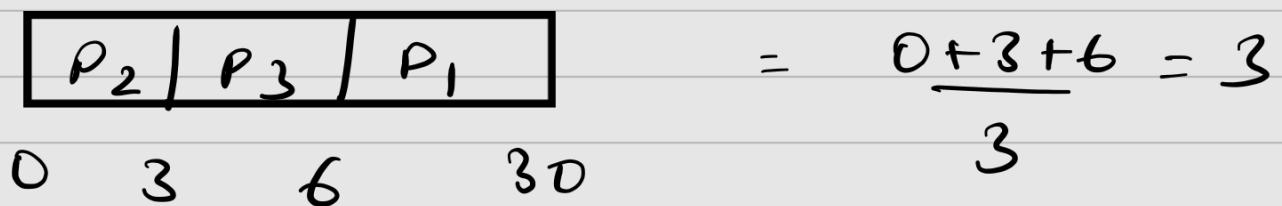
FIFO Queue

<u>Process</u>	<u>Burst Time (ms)</u>	<u>Graph Chart</u>
P <sub>1</sub>	24	
P <sub>2</sub>	3	
P <sub>3</sub>	3	
.	.	0
.	.	24 27 30

↑ { P<sub>2</sub>  
P<sub>3</sub>  
P<sub>1</sub>

$$\text{Avg. Waiting Time} = \frac{0+24+27}{3} = \frac{51}{3}$$

P<sub>2</sub>, P<sub>3</sub>, P<sub>1</sub>



Shortest-Job first (SJF)

Avg. Waiting Time

Process

Burst Time

P<sub>1</sub>

6

P<sub>2</sub>

8

4

P<sub>3</sub>

7

P<sub>4</sub>

3

⇒ 7 ms

P <sub>4</sub>		P <sub>1</sub>		P <sub>3</sub>		P <sub>2</sub>
----------------	--	----------------	--	----------------	--	----------------

0      3      9      16      24

Process

Arrival Time

Burst Time

P<sub>1</sub>

8 (7)

P<sub>2</sub>

4 (3)

P<sub>3</sub>

9

P<sub>4</sub>

5

preempted



P <sub>1</sub>		P <sub>2</sub>		P <sub>4</sub>		P <sub>1</sub>		P <sub>3</sub>
0	,	5	-	10	-	12	-	26

$$\text{Average waiting time} = (10-1) + (1-2)$$

$$+ (17-2) + (5-3)$$

4

5/2/25

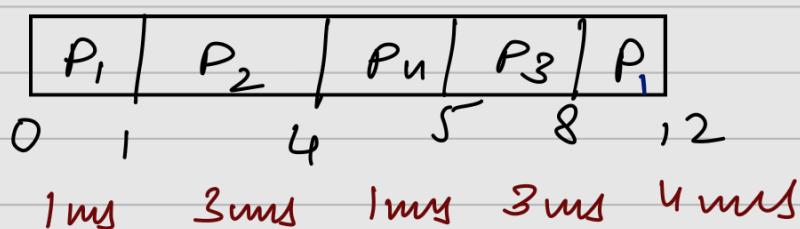
SJF

↳ Shortest Remaining Time first (SRTF)  
 - (pre-emption exists)

- SRTF (avg Turnaround Time)

<u>Process</u>	<u>S.T</u>	<u>B.T</u>
P <sub>1</sub>	0 +	5
P <sub>2</sub>	- 1	4
P <sub>3</sub>	2	2
P <sub>4</sub>	4	1

P <sub>1</sub>	0 +	5
P <sub>2</sub>	- 1	4
P <sub>3</sub>	2	2
P <sub>4</sub>	4	1



$$\text{Avg T.T} = \underline{(12-0)} + \underline{(4-1)} + \underline{(8-2)} + \underline{(5-4)}$$

$$\Rightarrow (12-0) + (4-1) + (8-2) + (5-4)$$

4

$$\Rightarrow \frac{12+3+6+1}{4}$$

$$\Rightarrow \frac{22}{4} = \frac{11}{2}$$

## Priority Scheduling

0 - highest

n - lowest

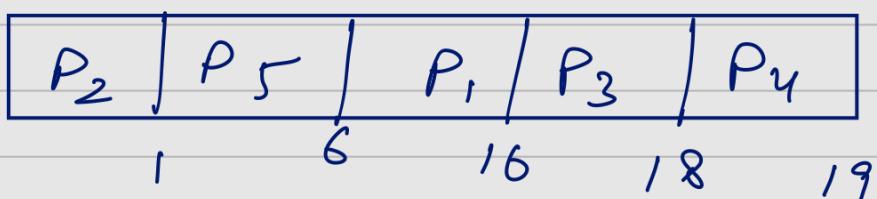
- FCFS  
feel & c  
breaking

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
----------------	-------------------	-----------------

$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

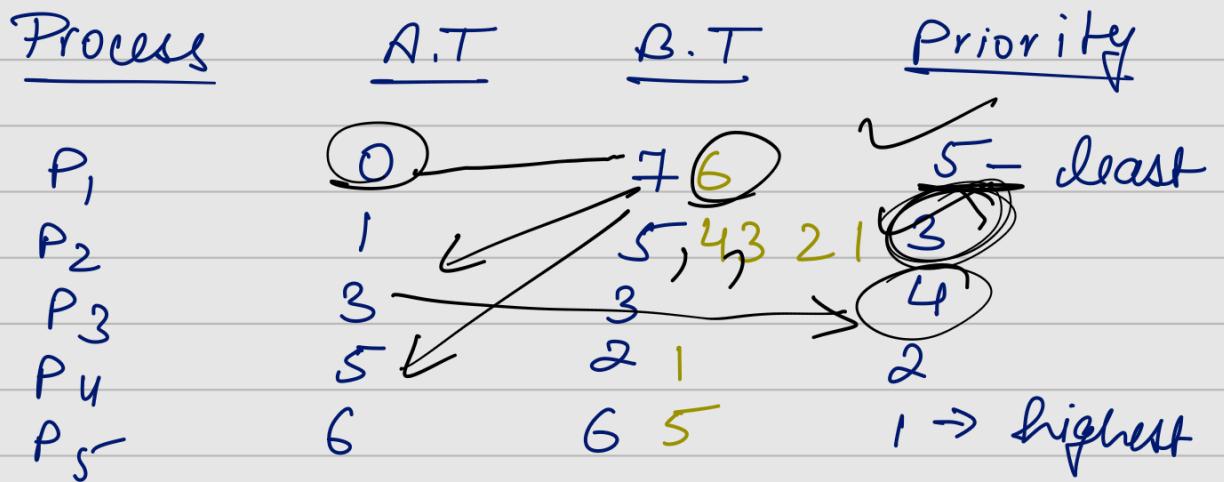
(1)  $\rightarrow$  h.p

(5)  $\rightarrow$  l.p



$$\Rightarrow \frac{1+6+16+18}{5}$$

$$\Rightarrow 8.2$$



P <sub>1</sub>	P <sub>2</sub>	P <sub>2</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>4</sub>	P <sub>2</sub>	P <sub>1</sub>	P <sub>1</sub>
0	1	3	5	6	12	13	14	17

$$\begin{aligned}
 \text{Waiting Time} &= (17 - 1 - 0) + (13 - 4 - 1) \\
 &\quad + (14 - 3) + (12 - 1 - 5) \\
 &\quad + (6 - 6) \\
 &= 16 + 8 + 11 + 6 \\
 &= 41
 \end{aligned}$$

$$\Rightarrow 8.2$$

10 | 2 | 25 Priority Scheduling

- Indefinite blocking or Starvation

- ↓ Ageing → to reduce effect of Starvation.

Every 10 min the process with lower priority will be increased by  $5/10$  min so that it has to wait for long.

# Round Robin (Time Sharing)

Everybody will get access to CPU based on a time slot.

→ Using Time Splice  
2 sec



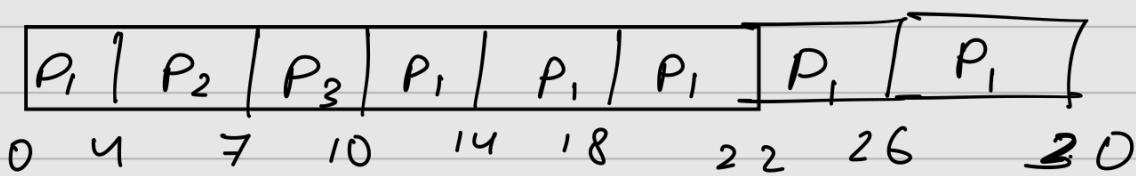
## Circular Queue following (FIFO)

After 2 sec

# Process

$P_1$	24
$P_2$	3, 5
$P_3$	3

Time  
Slice  
= 4ms

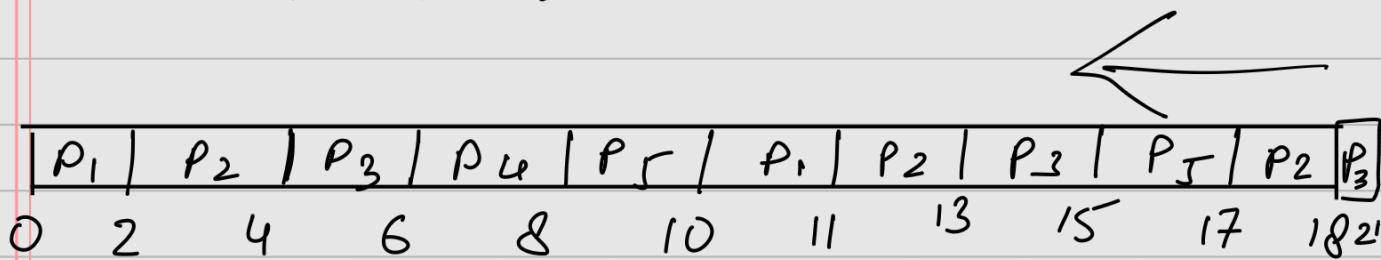


$$\text{Waiting Time} = \frac{(10-4)}{3} + 4 + 7$$

$$\Rightarrow 5.6 \text{ ms}$$

<u>Q.</u>	<u>Process</u>	<u>B.T</u>	<u>T.S</u>
	P <sub>1</sub>	3x10	2 ms
	P <sub>2</sub>	8x10	2 ms
	P <sub>3</sub>	7x3	2 ms
	P <sub>4</sub>	20	2 ms
	P <sub>5</sub>	4x10	2 ms

$$\text{Time slice} = 2 \text{ ms}$$



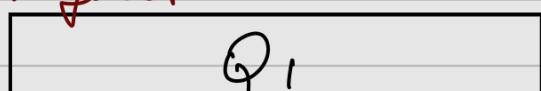
$$W.T = \underbrace{(20-6) + (17-4) + (15-2) + (6-0) + (10-2)}_{5} \text{ ms}$$

$$\Rightarrow 10.8$$

### Multi-Queue Scheduling

Foreground Process

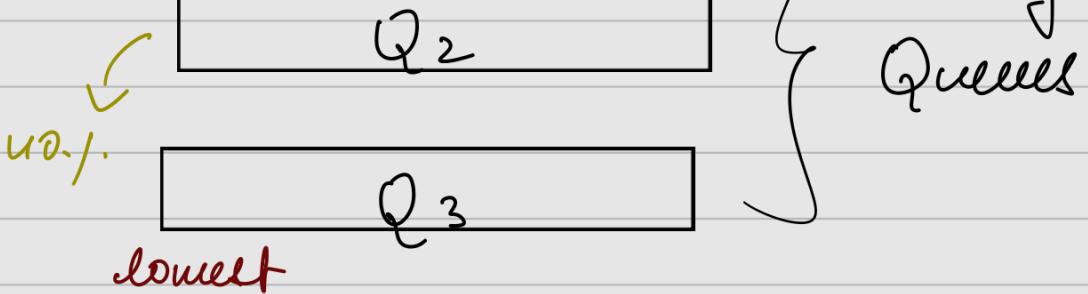
50.1. highest



)

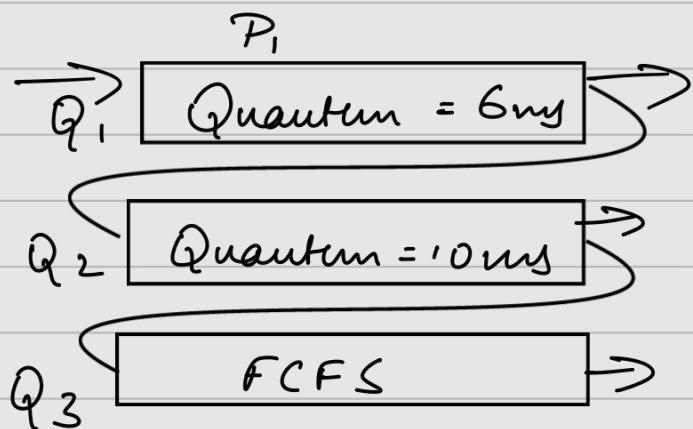
Ready

Background process



- Each of the queues may have different scheduling procedures based on DS.
- The CPU will only go from higher to lower priority queue, when the current queue is completely empty.

### Multi Level Feedback Queue Scheduling



- First process is allocated to Q<sub>1</sub>. If the process < quantum. It will finish in one go.

Otherwise rest of the process will be put in Q<sub>2</sub>. It will finish there.

Otherwise the rest of the process

will be shifted to Q<sub>3</sub>.

11/02/25

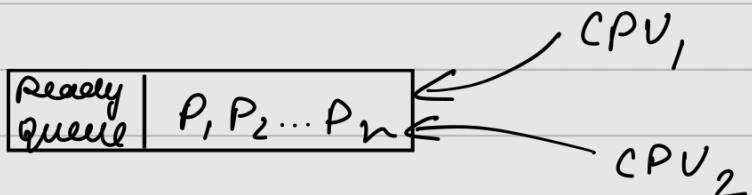
## Multiple Processor Scheduling

All processes

- 1) Load balancing - when multiple processors are idle, a process is divided among several processors.

### Approaches

- Asymmetric multiprocessor
  - All scheduling decisions, I/O decisions, system activities are handled by master server (one processor)
- Symmetric multiprocessing (SMP)
  - Each processor is self scheduling.  
" " has its own scheduler.



Mutual Exclusion - where multiple

processors try to access the same process.

## Processor Affinity

  $Q_1 \rightarrow \text{CPU}_1$

  $Q_2 \rightarrow \text{CPU}_2$

Phenomenon in which processes don't want to shift from one CPU to other.

Reason - It will create unnecessary overhead since it will require memory to load the process again.

- **Soft affinity** - Processes may or may not be shifted. They are not given a choice as to where they need to be shifted.
- **Hard affinity** - Processes have to shift, they are given a choice for main processor and subset processors.

Load Balancing  $\rightarrow$  trying to distribute workload among all

$3 \text{ CPU} \rightarrow 2$  are working  $\frac{\text{available}}{\text{resources}}$ .  
1 is idle, however

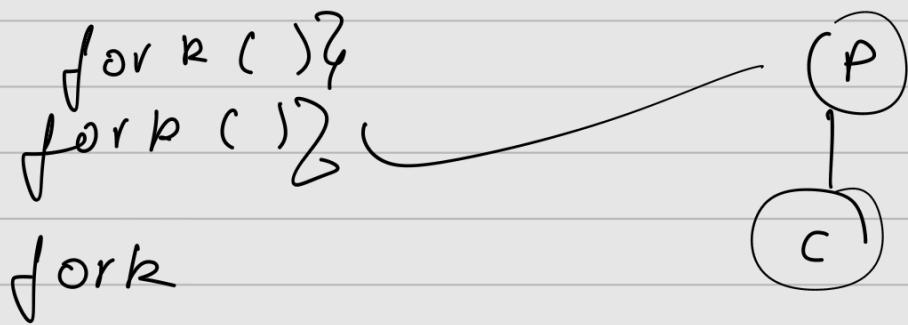
| processes are waiting to be scheduled.

Push Migration  $\rightarrow$  specific program

It pushes processes from overburdened processors to idle / less - burdened processors.

Pull Migration  $\rightarrow$

Idle processors pull processes from over-burdened processors.

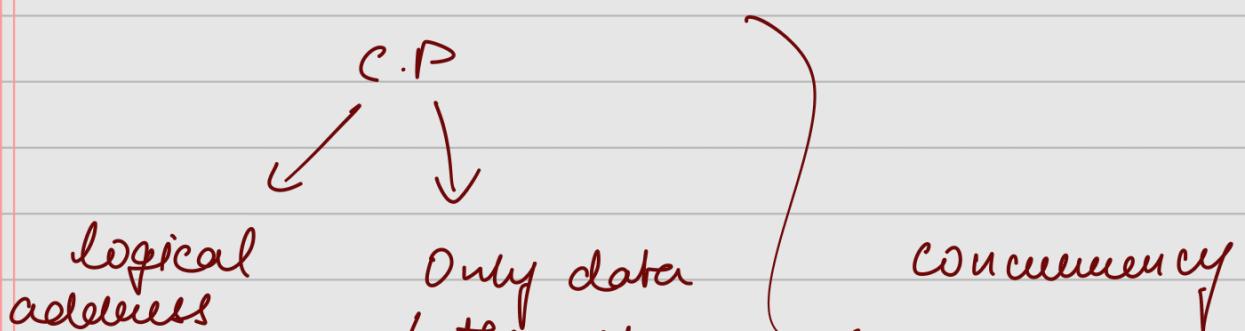


### Module 3

17/02/25

### Cooperating Processes

The multiple processes who are trying to share data amongst each other.



space  
(both code  
& data)

(through  
files or  
messages)

## Problems of concurrency :-

- Data inconsistency - When different processes try to access the same data.

### Properties:-

- Communication amongst processes.
- Sharing of resources
- Competing for resources
- Synchronization of activities b/w processes.
- Allocation of processor time to processes.

### Example

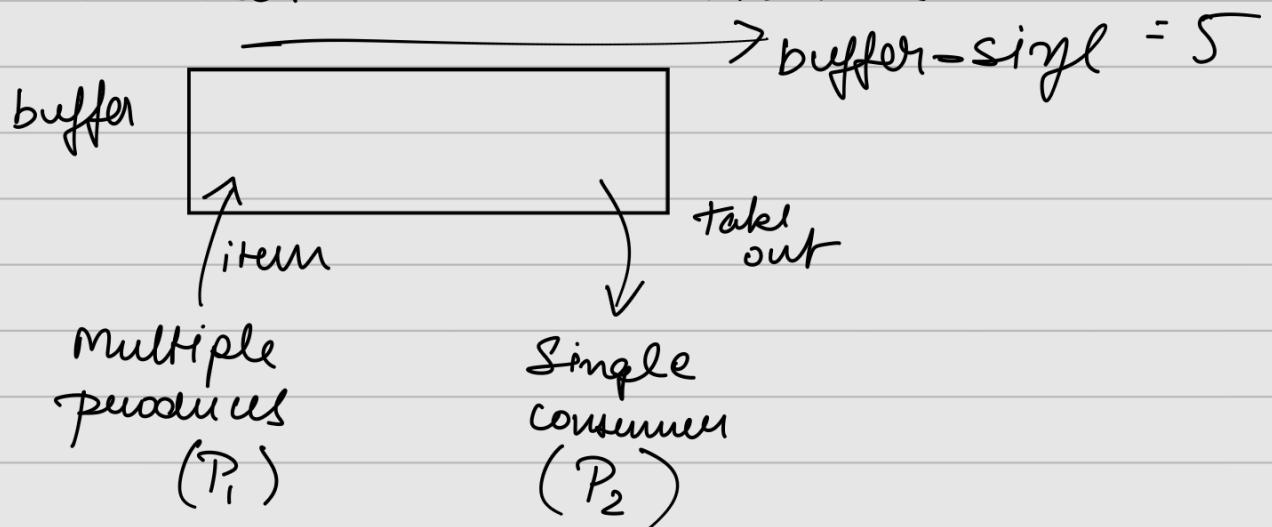
#### Global Variables

$$a = 1, b = 2$$

sequence  
of  
execution

Process i	Process j
$a = a + b$	$b = a + b$

## Producers - Consumers Problem



### Problem

Underflow - No item is being produced by producer but consumer is trying to get the item

↳ data synchronization in the soln

When multiple concurrent process who are trying to access a shared resource, there should be some order.

Outcome will depend on the order/sequence in which the access to the shared resource happens is known as Race Condition.

### Producer

while (true)

### Consumer

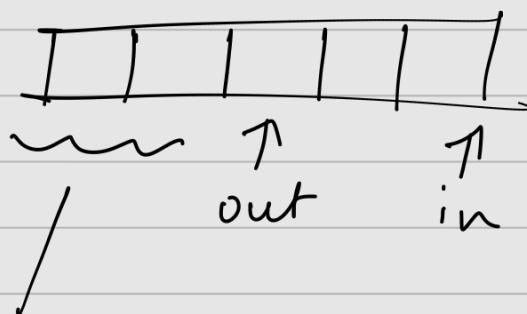
while (true) { , }

```

while (counter == 0) {
    while (counter == BUFFER_SIZE);
    buffer[in] = produced;
    in = (in + 1) % Buffer_size;
    counter++;
}
    consumed = buffer[out];
    out = (out + 1) % Buffer_size;
    counter--;
}

```

counter - keep track of no of elements in buffer



vacant space so use %.

~~18 | 2 | 25~~

Critical section -

Shared resource / variable



critical section  $\leftarrow$  buffer[in] = produced  
 consumed = buffer[out]



critical section  $\leftarrow$  in++,  
 sumt(buffer[n])  $\leftarrow$  remainder set in

Define:- A section of code within a process that requires access to a shared resource & it must not be executed while another process is running in its corresponding critical section.

do {

entry section

critical section

exit section

remainder section

}

while (true)

)  
structure  
of a  
program  
which  
tries to  
access a  
shared  
resource

E.S  $\Rightarrow$  Each process must request permission to enter its critical section and the section of code that is implemented is called entry section.

A solution to the CS problem must satisfy the following:-

- Mutual Exclusion - When two processes are trying to access a shared resource, only one must access the critical section of the resource at a time.

- **Progress** - If no process is executing in its critical section and wish to enter their critical section, then only those processes which are not executing their remainder section can participate in deciding which will enter its critical section next, this selection can't be postponed indefinitely.
- **Bounded wait** - when  $P_2$  is looping in its E.S, it should not loop forever.

Sol<sup>n</sup>: -

After  $P_1$  finishes execution, it will update a variable at exit section which will satisfy the entry condition for  $P_2$ .

*Right the story  
here !*

### Peterson's Solution

#### Process i

exit.s ←

```

do S
flag [i] = true;
team = j;
while (flag [i] && team == j);
critical section
flag [i] = false;
remainder section
}
while (true)
    
```

#### Process j

```

do {
flag [j] = true;
team = i;
while (flag [i] && team == i);
critical section
flag [j] = false;
remainder section;
}
while (true);
    
```

- Both process will start executing simultaneously.
- Both will set  $\text{flag}[i] = \text{true}$  &  $\text{flag}[j] = \text{true}$
- Either  $\text{turn} = j$  or  $\text{turn} = i$ , both differing by a millisecond.
- If  $\text{turn} = j$ , then process  $j$  will go to critical section.

Definition

Differentiate b/w

& focus on scheduling [Avg waiting time  
& on fork Avg turnaround time]

5(a) & 5(b) from module 3

Drawbacks of Peterson soln:-

- Only two processes
- Only one shared resource
- Spin wait - the process which is looping, is stuck doing nothing. It cannot even do work which doesn't require critical section.

19/2/25 - Locking - protecting the CS through locks

- Acquiring the lock
- Releasing " "

/

### Mutex lock

→ variable  $\Rightarrow$  available

```
acquire () {  
    while (!available) // busy waiting  
        available = false;  
}
```

```
release () {  
    available = true;  
}
```

```
do {  
    acquire lock ;  
    critical section  
    release lock ;  
    remainder section ;  
} while (true);
```

Process  $P_i$   $\rightarrow P_1, P_2$

```

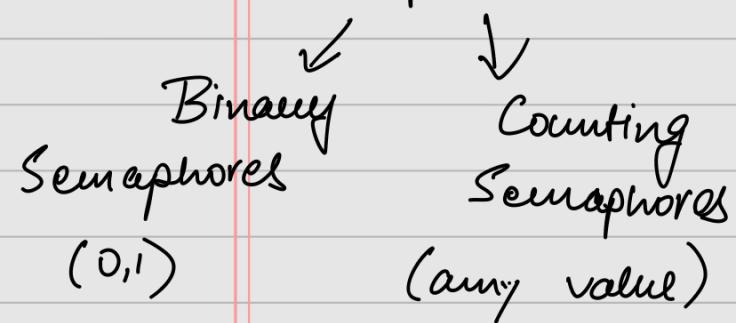
do {
    acquire ();
    Execute CS;
    release ();
    remainder section
}
while (true)

```

## Drawbacks

- One Shared resource
- Spin Wait

## Semaphores



A semaphore is an integer value used for signalling amongst processes

Two operations:-  $\text{wait}() \rightarrow P$   
 $\text{signal}() \rightarrow V$

Semaphore ( $s$ )

$\text{wait}(s)$  { resource

$\text{while } (s <= 10);$   
 $s--;$

$\text{signal}(s)$  }

$s++;$

y

3