1. Operating Systems : Internals & Design Principles by William Stallings

2. Operating System Concepts – by Galvin

OS -

Transistors - CMOS

↳ are connected to form gates.
Gates are used for logical operations

VLSI

↳ Organization → Memory units, I/O units

↓

OS

↓

Application

H/W Abstraction

Resource Management

printf ("Hello World"), "Hello World" Registers

RAM ← Processor

fetch

Color, coordination, depth + Graphics Card → Monitor

Scheduling

App1 | App2

OS keeps all applications isolated in their respective environments.

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

15/1/25

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

Job 1 → 12 o'clock →
Job 2 → " → 12.01 sec

more priority *

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

User 1 → 40 mins
(12 - 12:40)

User 2 → waiting

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

② Setup time

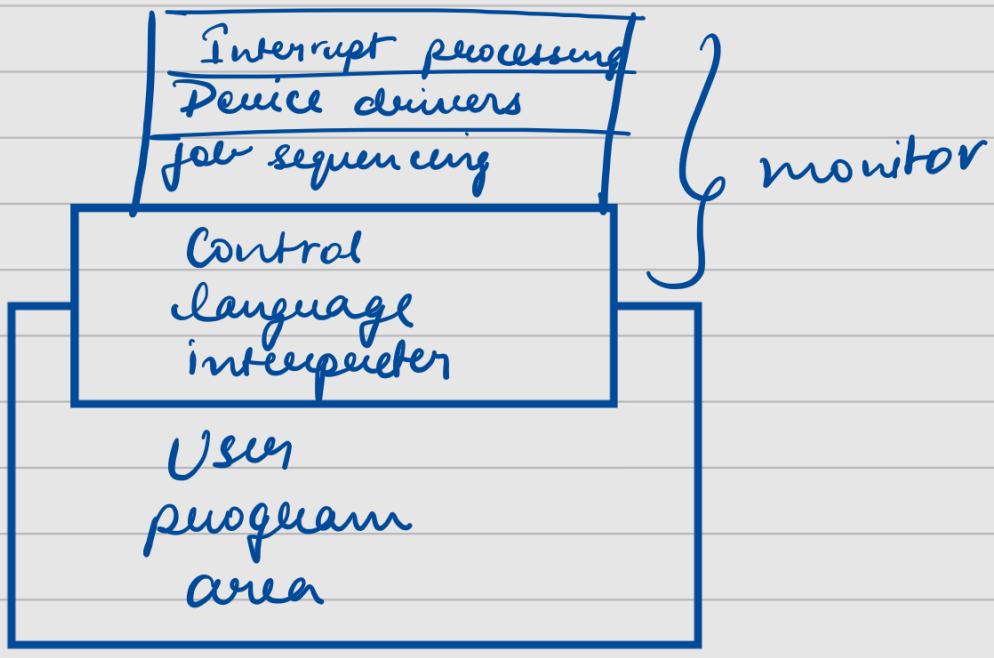
Simple Batch System (mid 1950s)

→ 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 1

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

2/30

Modes of Operation

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

(3) Multi-processing

Job 1: I/O (5 ms) | CPU (2 ms) | I/O (5 ms) — 12 ms

CPU is idle

3 ms

Job 2 [CPU] I/O

2 ms 5 ms

3 + 5 = 8 ms

5 + 5 = 10 ms

Efficient & Minimize response time.

(4) Time-Sharing

'n' jobs

↓

1/n

Time Slicing

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

21/2/25

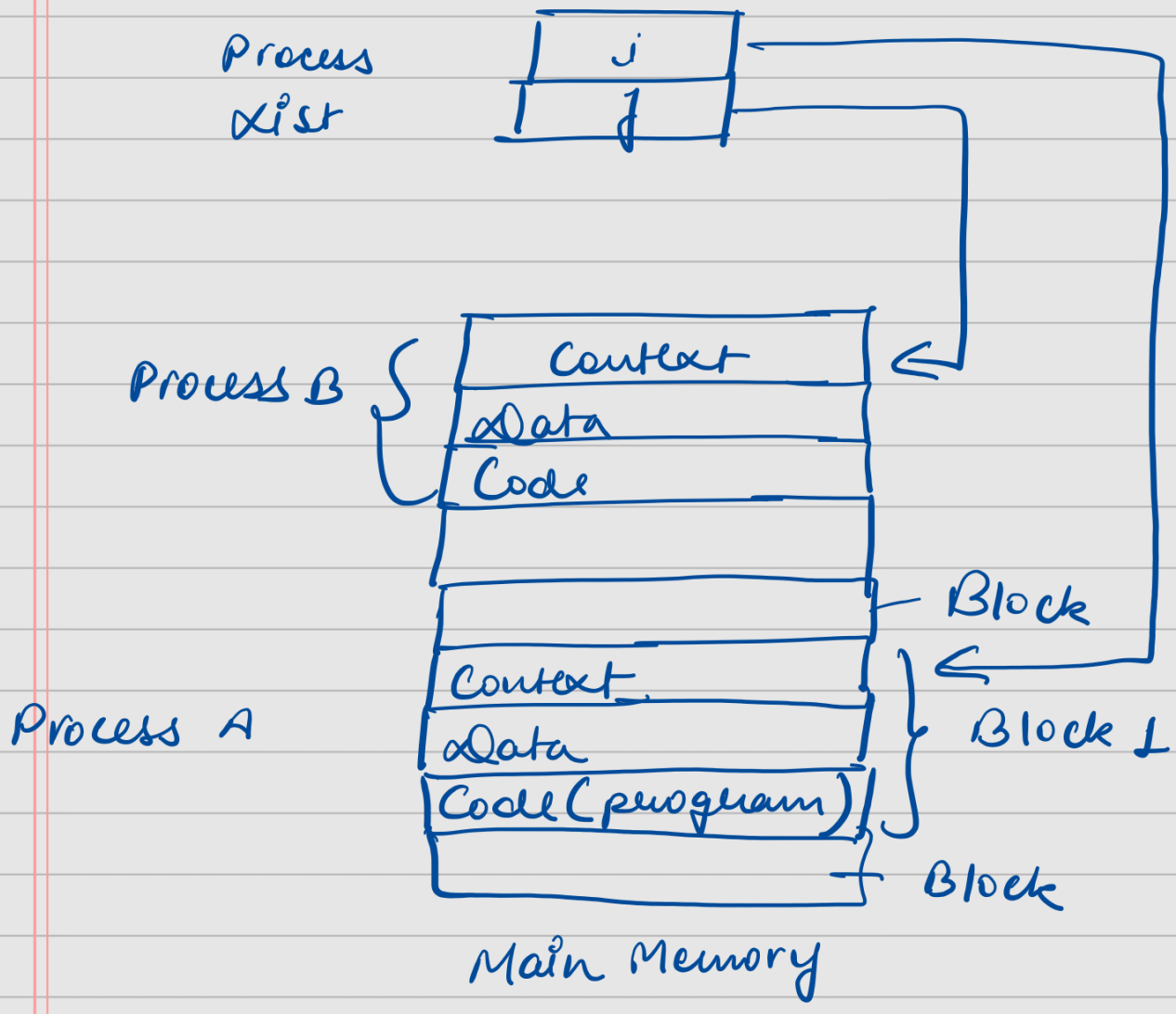
Processes

A B

↙ ↘

block of memory block of memory

A { (prog, data, content) }



Processor

Starting address of the process it is currently executing

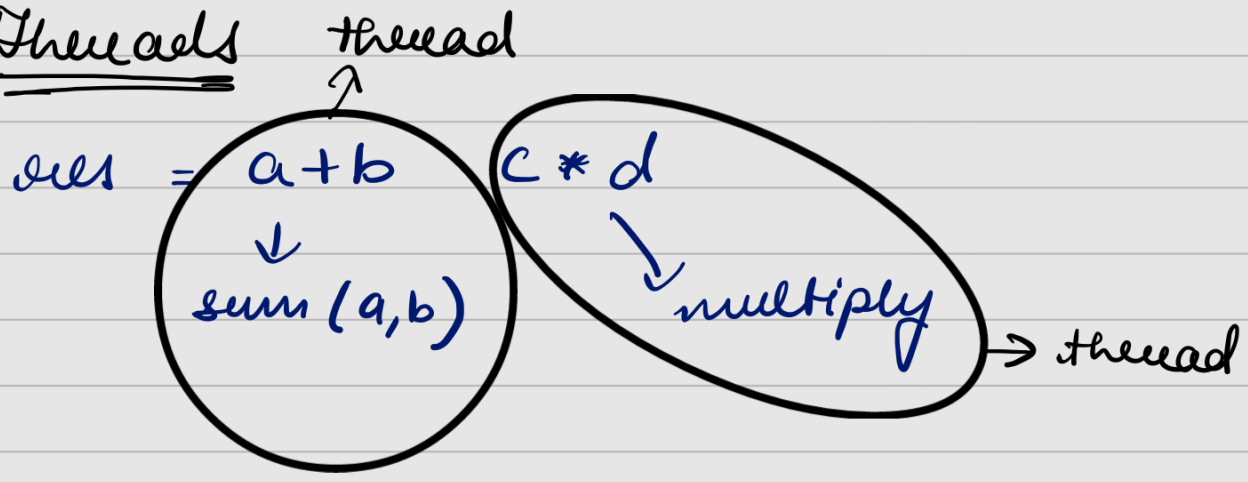
- Process Index → Pointer to the process list for the processes its currently running
- Base Register
- Limit Register → stores the size (20 bytes)

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 while preserving the info of the state of the process.

- Memory is allocated to each process.

States of a process => Ready, Running, Wait / Stop

Threads



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

Memory Management

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

Information Security

CIA ⟶ Authenticity

↙ 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
2. State (New, Ready)
6. Content Data
7. I/O status info

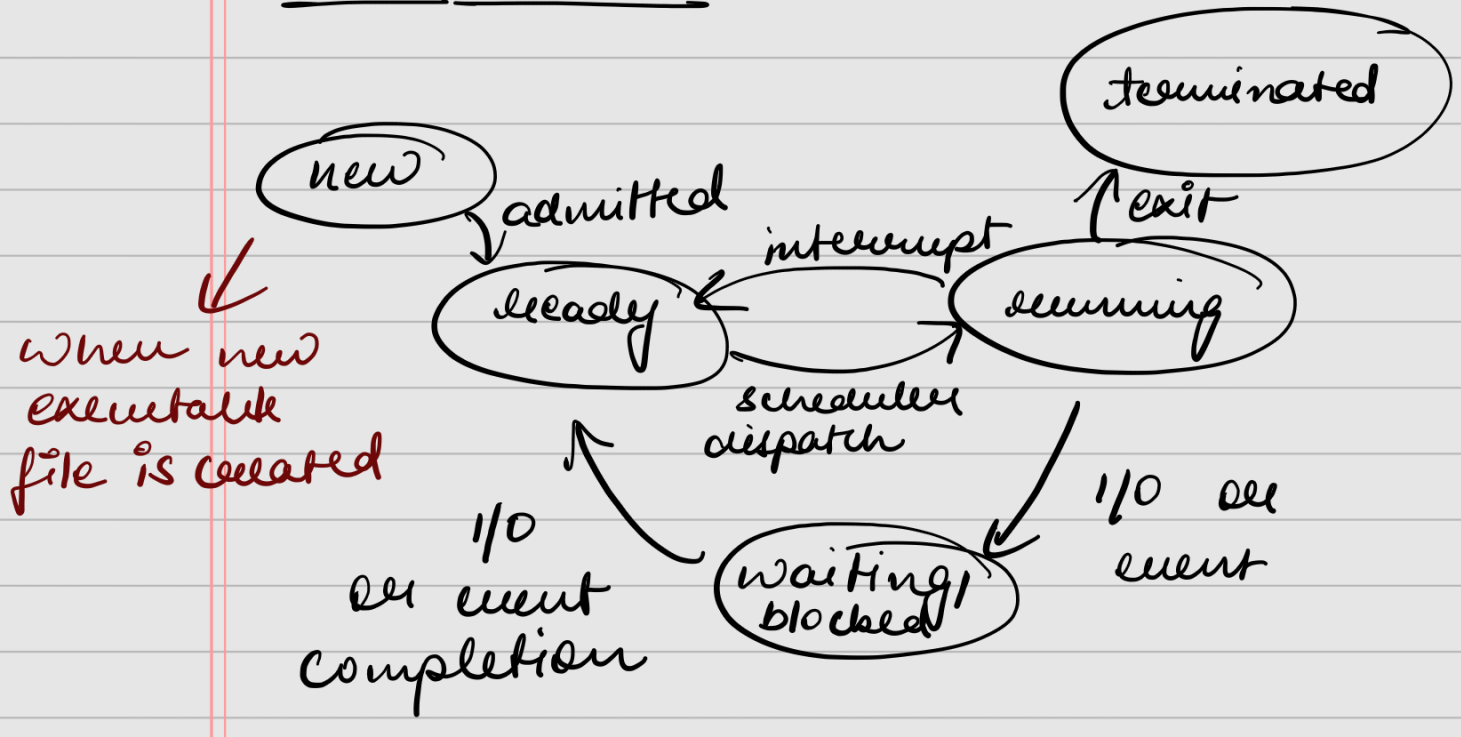
3. Priority
4. Program Counter
5. Memory Pointers

* 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.

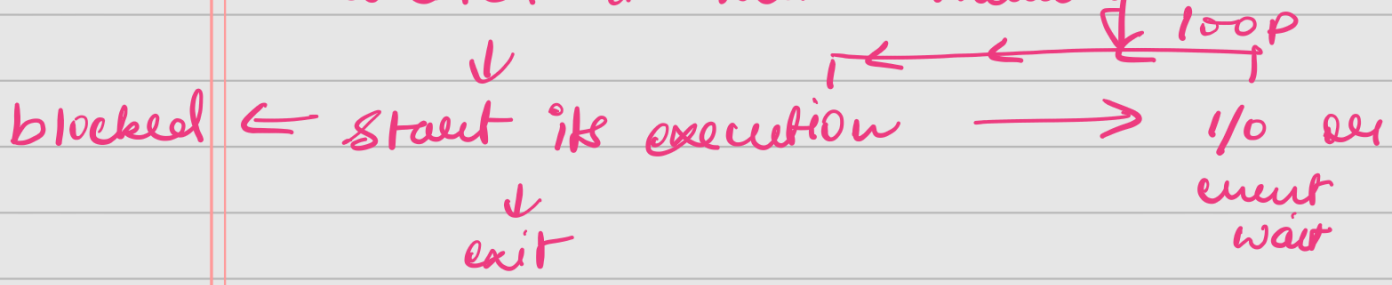
5 - State Model

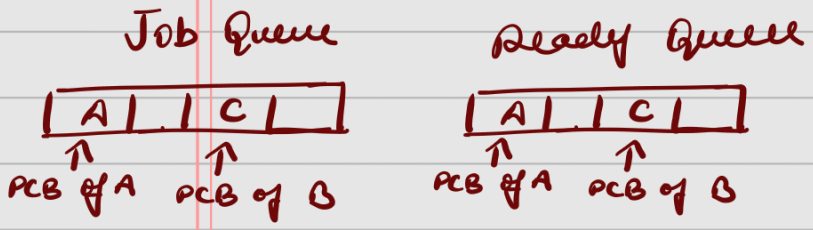


- Only when a program is in the ready state, can it be brought to running state.

Info inside PCB are stored in a linked list.

program
↓
executable file
↓
allocated to main memory





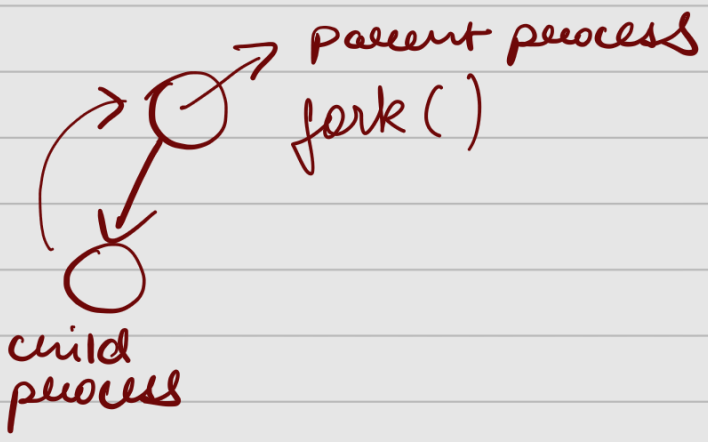
Ready state
→ memory is allocated for the program

- 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 seperate queue for itself, I/O queue.
- On ready state → job queue

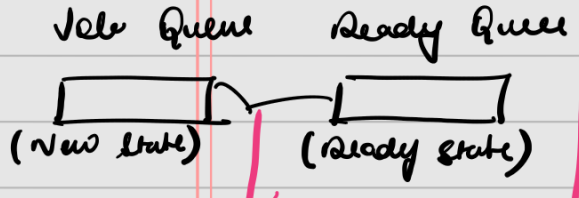
On running state → I/O queue

When a process is running :-

- Call for an I/O operation
- create a child process
- forcebly removed due to errors



Schedulers



Long Term Scheduler (or)
- Take processes from job queue and load them into memory
- Decides the degree of multiprogramming

Short-Term Scheduler (or) CPU scheduler
Select process from ready queue and load them into CPU.

- I/O activities are more consuming than CPU.

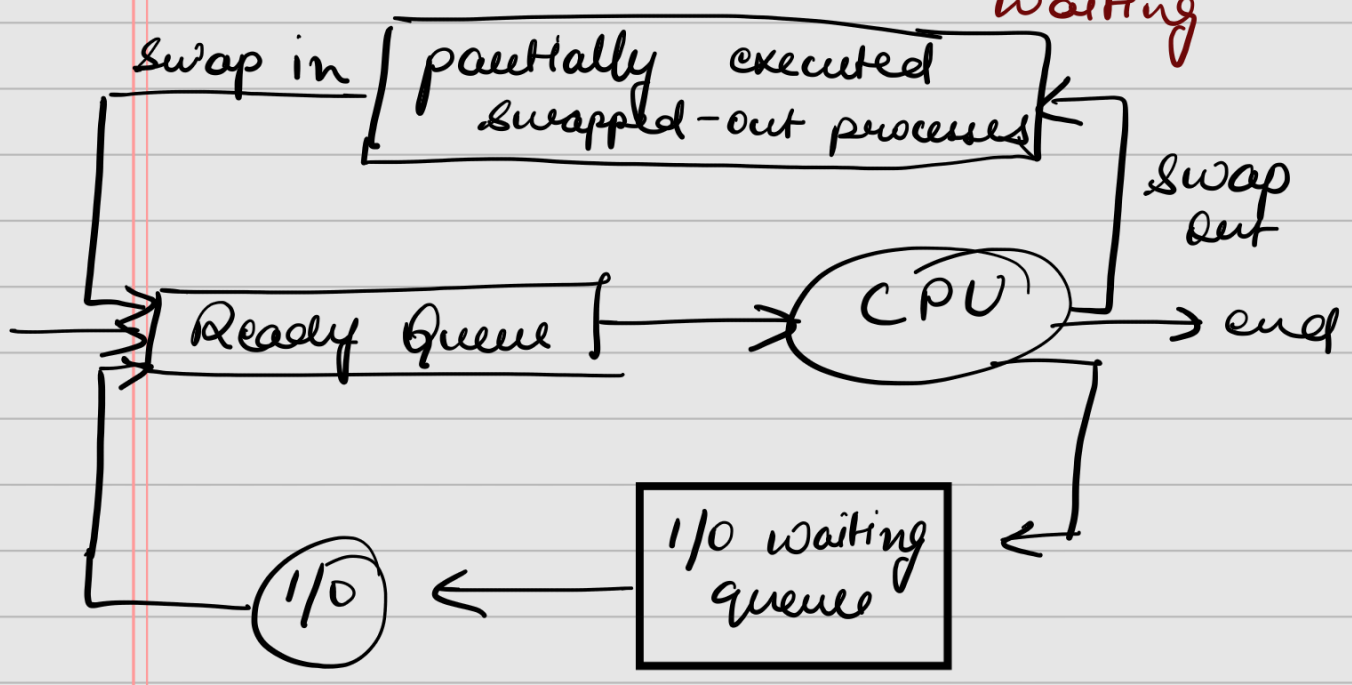
Objective of OS is to have a good balance b/w CPU & I/O bound.

Running state
Waiting state

Medium-Term Scheduler
to reduce degree of multiprogramming

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.

(PCB)
Process A
(gaming app)

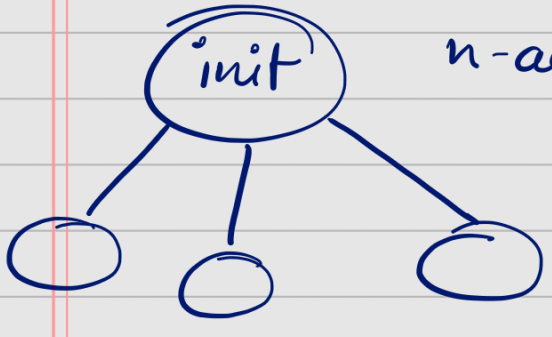
(PCB)
Process B
(accessing any hardware)

* If both process are in same mode, then its not context switching

27/1/25

Process Creation

pid - process id
init process - pid = 1



n - any tree

fork() → system call

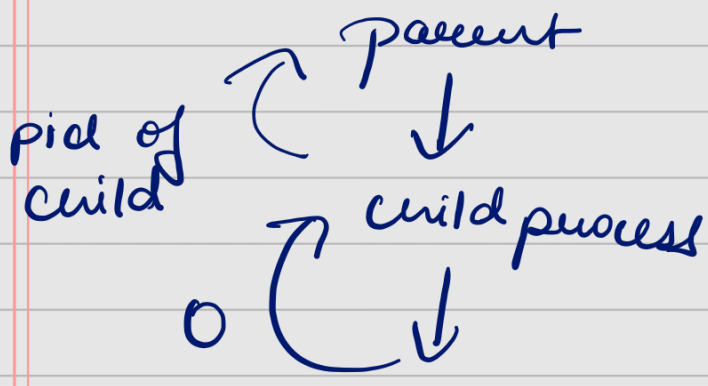
ps → all the processes in the system

ps -el → all the running processes

Process Creation Steps

1. Assigning the pid
2. Allocating memory space for the process.
3. Initializing the PCB → transition process

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



Angle of the parent

exit() → terminates the process

Cascading Termination →

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

wait() → The parent waits until the child process finish execution.

→ returns the pid of the child that has finished its execution

Child wants to terminate first →

- resources given to the child are deallocated by the OS. However the entry still remains in the PCB. } → parent terminate 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

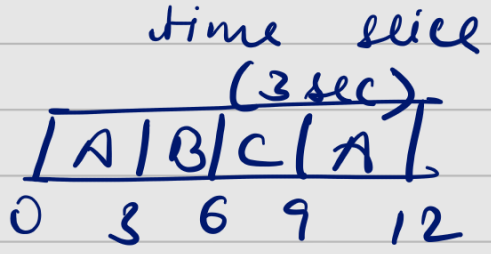
Process switch

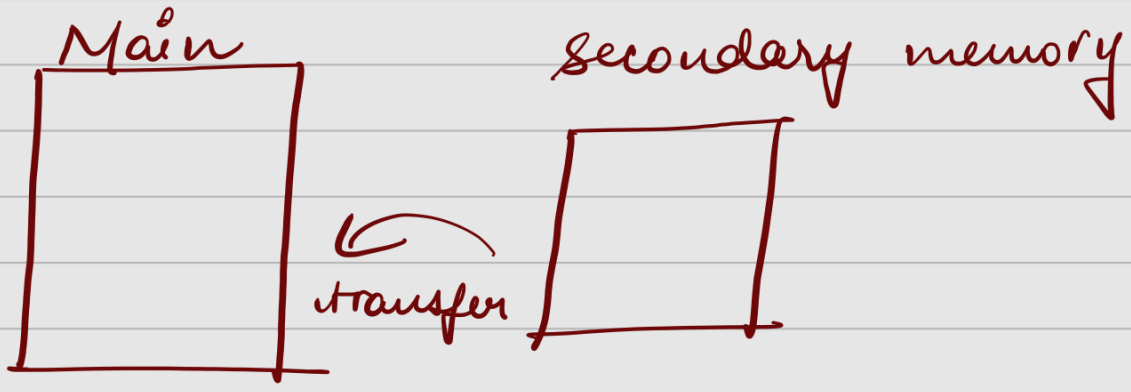
- Interrupt
  - Clock Interrupt
  - I/O Interrupt
  - Memory fault
- Trap (error is encountered)

Ex - when tries to access invalid file or an i/o device.

- when multiple processes are running but not doing significant task, OS tries 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

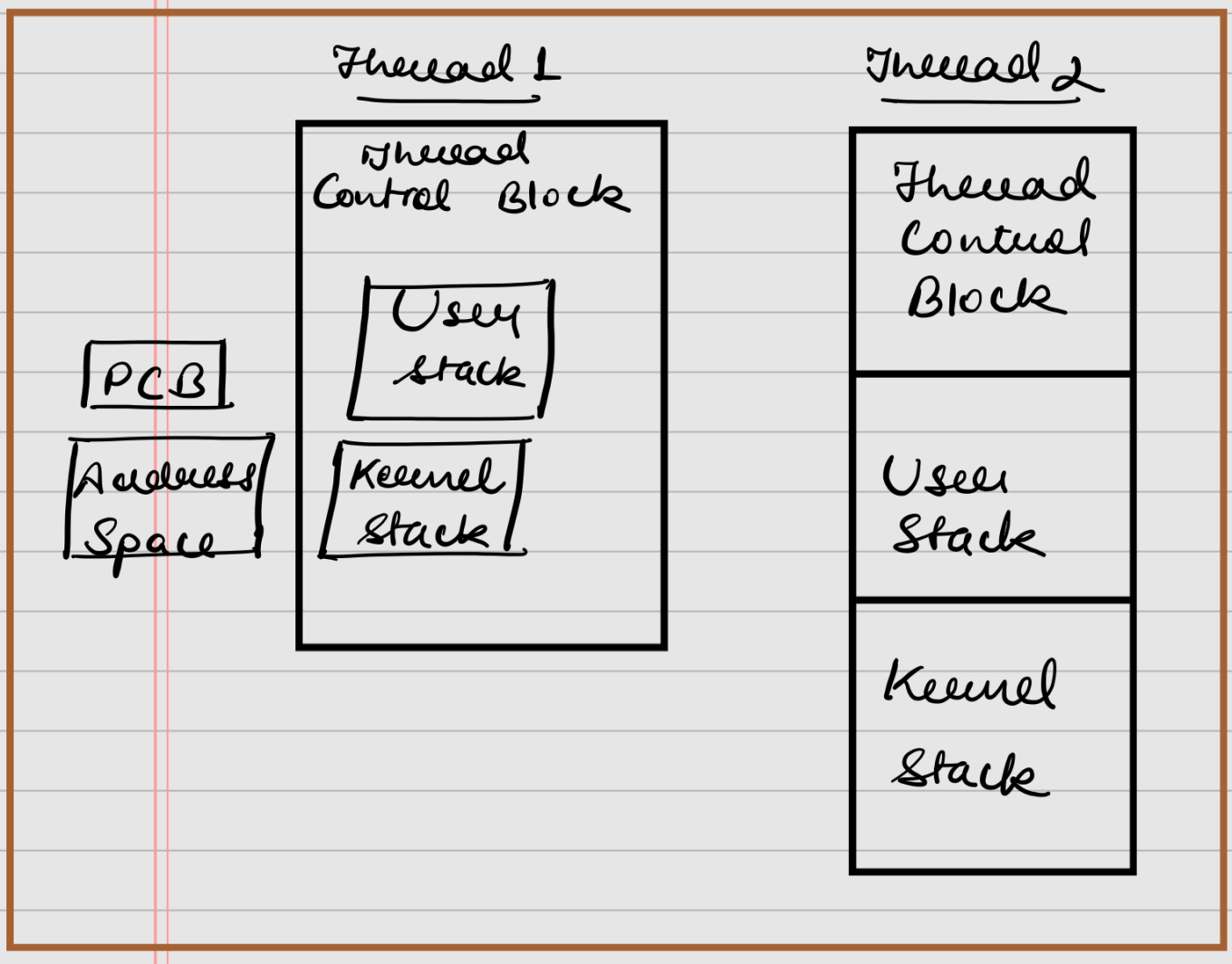
- A process can have multiple threads.

$x = a + b$ → Thread 1

$y = a - b$ → Thread 2

output = x * y

Multithreading → ability of the os to support multiple concurrent execution paths within a single process.



Benefits

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

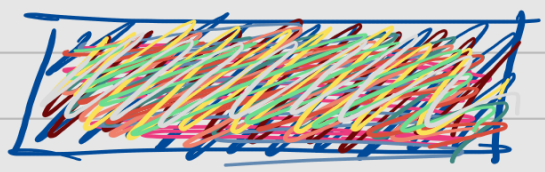
Relationship b/w Threads & process

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

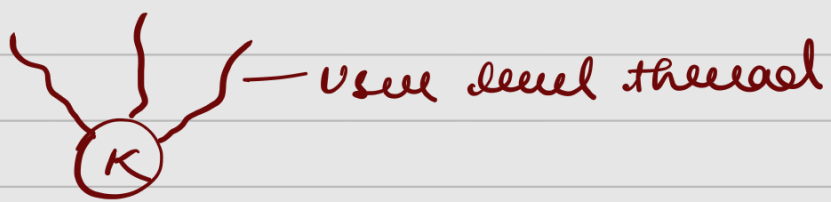
Multithreading Model

User Threads → support is provided at the user level.

Kernel " → support is provided at Kernel level.

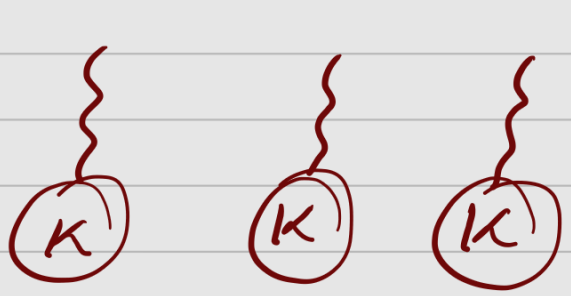


1.) One - Many model



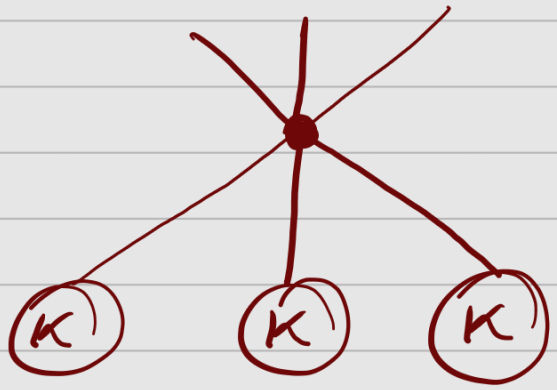
Kernel - level thread

2) One - One Model



(more concurrency)

3) Many to Many



Every user level thread can access every kernel level thread.

- True concurrency is more possible.

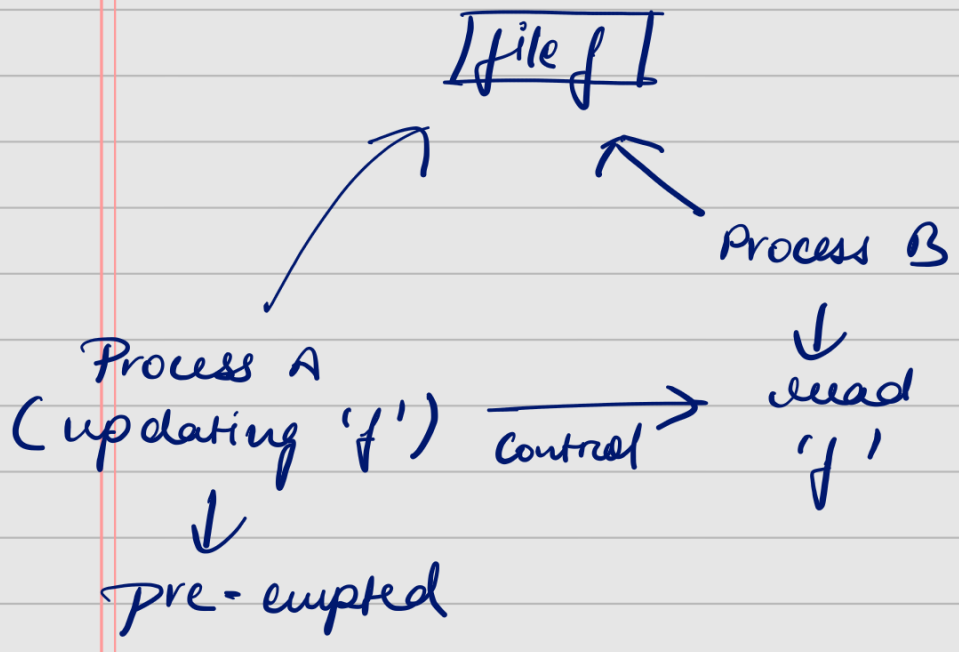
29/1/25 Scheduling

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

(voluntarily releasing it)

↳ process terminates
↳ " goes to i/o

Race Condition



Mutual Exclusion ← Race Condition

Scheduling Criteria

1. CPU utilization

2. Throughput → no of processes completed per unit time

maximized ↑

3. Turnaround Time → Time from submission to completion of a process.

4. Waiting Time → Sum of all the times that a process is waiting

5. Response Time →

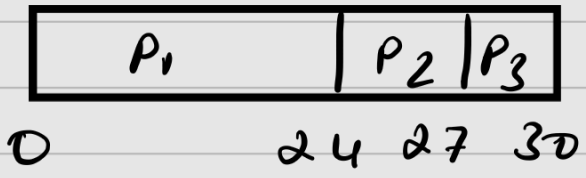
minimized

First-Come First Serve (FCFS)

FIFO Queue

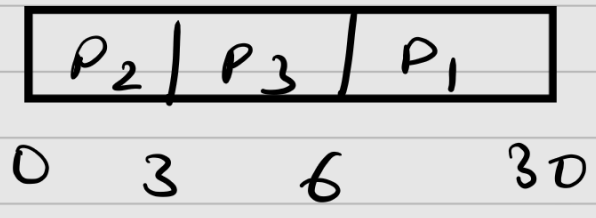
Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3
⋮	⋮

convoy effect



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

P_2, P_3, P_1



$$= \frac{0+3+6}{3} = 3$$

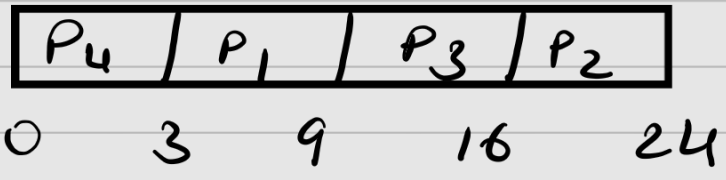
Shortest-Job First (SJF)

Avg. Waiting Time

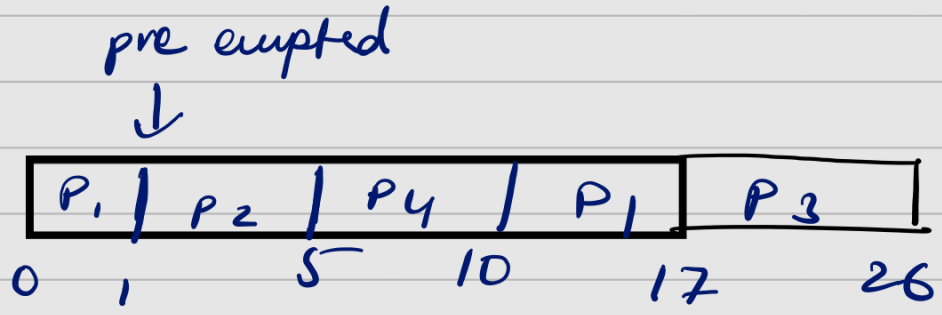
$\Rightarrow \frac{3+16+9+0}{4}$

$\Rightarrow$ 7 ms

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



Process	Arrival Time	Burst Time
P_1	0	8 (7)
P_2	1	4 (3)
P_3	2	9
P_4	3	5



Average waiting time $= (10-1) + (1-1)$

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

4

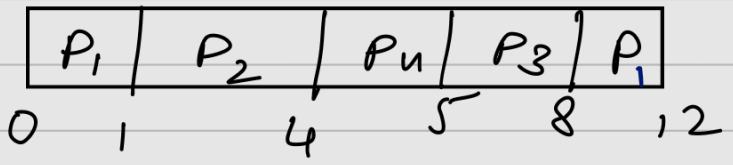
5/2/25

SJF

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

– SRTF (avg Turnaround Time)

Process	A.T	B.T
P_1	0	5 4
P_2	1	3 2 1
P_3	2	3
P_4	4	1 0



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

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

4

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

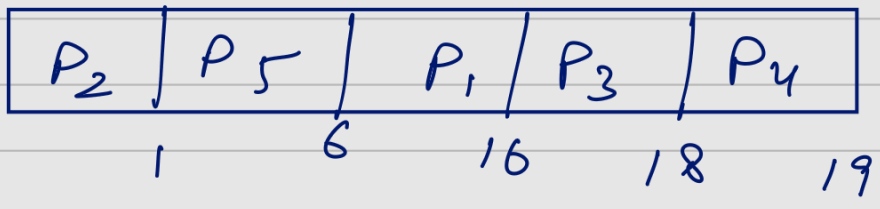
$$\Rightarrow \frac{22}{4} = \frac{11}{2}$$

Priority Scheduling

- FCFS for tie breaking

0 - highest
n - lowest

Process	Burst Time	Priority
P_1	10	3
P_2	1	1 → h.p
P_3	2	4
P_4	1	5 → l.p
P_5	5	2



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

$$\Rightarrow 8.2$$

Process	A.T	B.T	Priority
P_1	0	7 6	5 — least
P_2	1	5, 4 3 2 1	3
P_3	3	3	4
P_4	5	2 1	2
P_5	6	6 5	1 → highest

P_1	P_2	P_2	P_4	P_5	P_4	P_2	P_3	P_1

0 1 3 5 6 12 13 14 17 23

Waiting Time = $(17-1-0) + (13-4-1)$

$+ (14-3) + (12-1-5)$

$+ (6-6)$

5

$\Rightarrow 16 + 8 + 11 + 6$

$\Rightarrow 8.2$

10/2/25 Priority Scheduling

- Indefinite blocking or Starvation

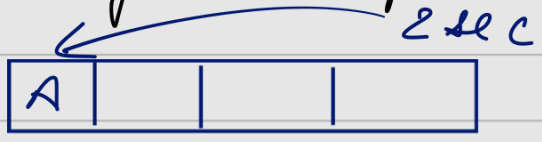
- ↓ Ageing → to reduce effect of Starvating.

Every 10 min the process with lower priority will be increased by 5/10 min so that it has to not wait for long.

Round Robin (Time Sharing)

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

→ Using Time Splice



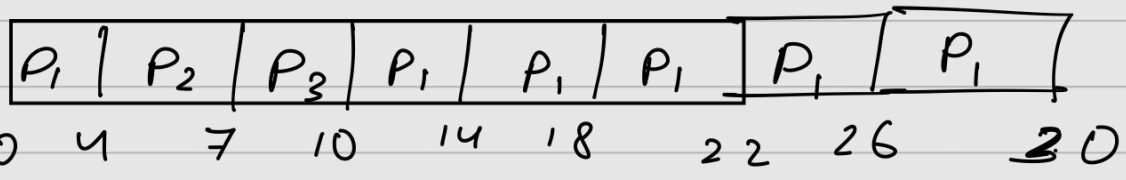
Ready Queue

Circular Queue following (FIFO)

After 2 sec

Process	B.T
P_1	24
P_2	3, 5
P_3	3

Time Slice = 4ms



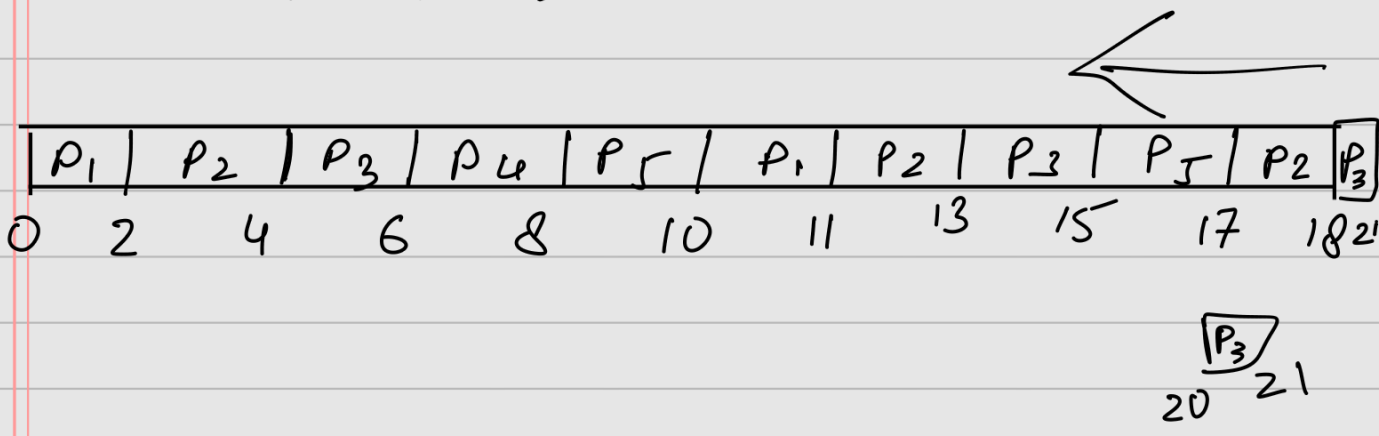
$$\text{Waiting Time} = \frac{(10-4) + 4 + 7}{3}$$

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

Q.

Process	B.T	T.S
P_1	8 1 0	2 ms
P_2	5 3 1 0	2 ms
P_3	7 5 3	2 ms
P_4	2 0	2 ms
P_5	4 2 0	2 ms

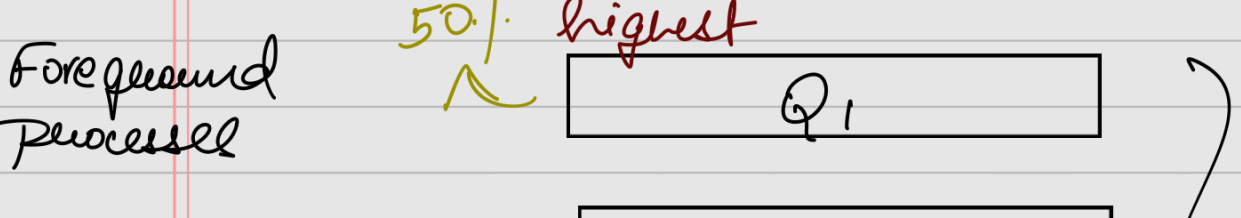
Time slice = 2 ms

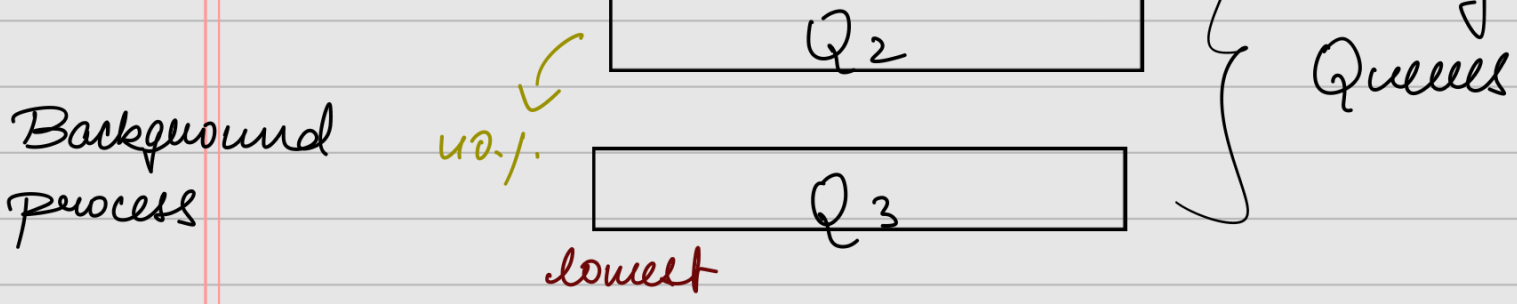


$$w.T = \frac{(20-6) + (17-4) + (15-2) + (6-0) + (10-2)}{5}$$

$$\Rightarrow 10.8$$

Multi-level Queue Scheduling

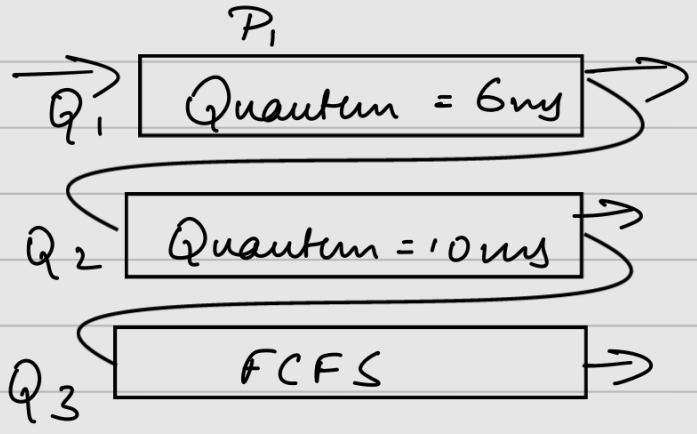




- Each of the queues may have different scheduling procedures based on OS.

- The CPU will only go from higher to lower priority queue, when the current queue is completely empty.

Multilevel Feedback Queue Scheduling



- First process in allocated to Q_1.
If the process < quantum. It will finish in one go.

Otherwise rest of the process will be put in Q_2. It will finish there.

Otherwise the rest of the process

will be shifted to Q_3.

11/02/25

Multiple Processor Scheduling

All processors

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

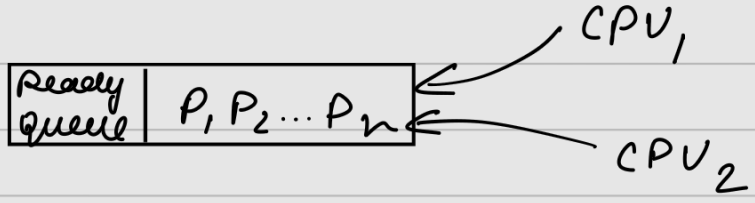
Approaches

- Asymmetric multiprocessing — Master Server (all scheduling related decision. I/O Decision, System Activities) — all other processor

  - All scheduling decisions, I/O decisions, system activities are handled by master server (one processor)

- Symmetric multiprocessing (SMP)

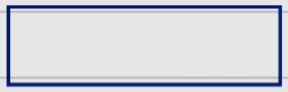
  - Each processor is self scheduling.
    " " has its own scheduler.



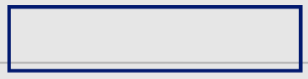
Mutual Exclusion - when multiple

processors try to access the same process.

Processor Affinity



$Q_1 \Rightarrow CPU_1$



$Q_2 \Rightarrow CPU_2$

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 needed be shifted.

- Hard affinity - Processes have to shift, they given a choice for main processor and subset processors.

Load Balancing → trying to distribute workload among all available resources.

3 CPU → 2 are working
1 is idle, however

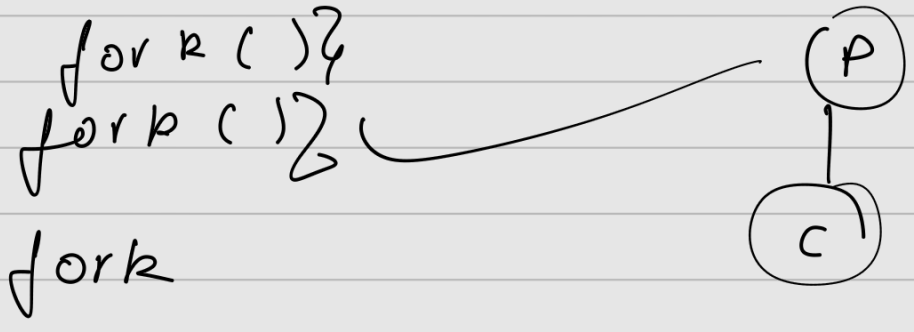
processes are waiting to be scheduled.

→ Push Migration → specific program

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

Pull Migration →

Idle processors pull processes from over - burdened processors.

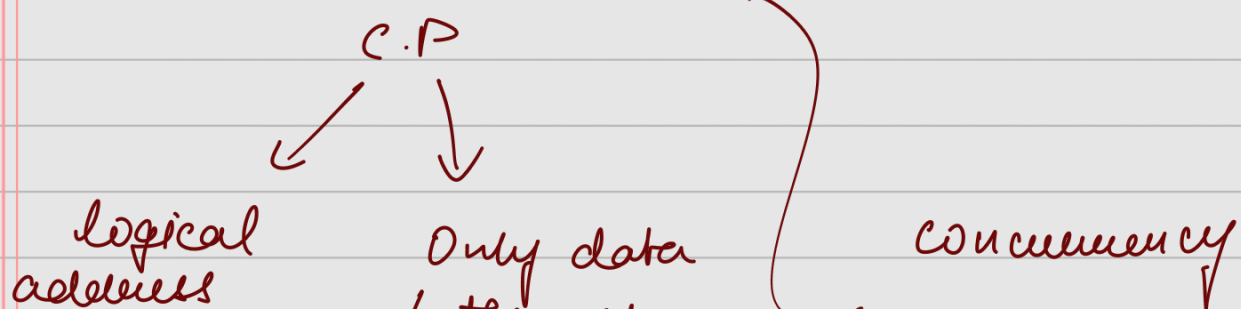


Module 3

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 process 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

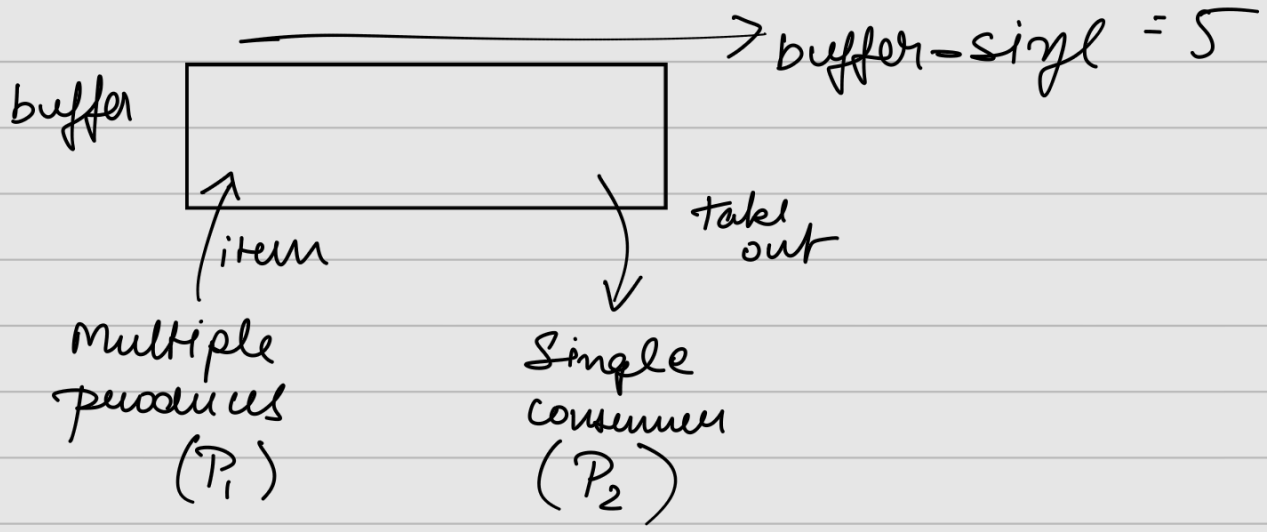
Global Variables

a = 1 , b = 2

sequence of execution

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

Producer - Consumer 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)
```

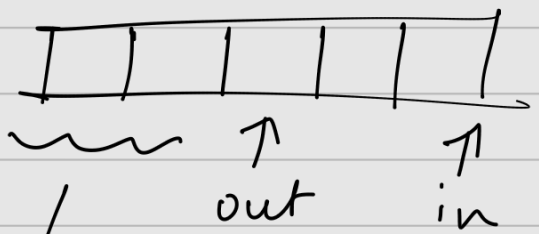
Consumer

```
while (true) {
```

```
while (counter == BUFFER.SIZE);
buffer [in] = produced;
in = (in + ) % Buffer_size;
counter ++;
}
```

```
while (counter == 0)
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 ← buffer [in] = produced
consumed = buffer [out]

critical section ↙ in++,

remainder section ← use (buffer [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 → 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 C.S, it should not loop forever.

Soln:-

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

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

exit:s ←

Process j

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

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

Defination
Differentiate b/w
& sums on scheduling [Avg waiting time Avg turnaround time]
& on fork

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 ⇒ 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 → P_1, P_2

```
do {
```

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

Drawbacks

- one shared resource
- Spin wait

Semaphores

- Binary Semaphores (0,1)
- Counting Semaphores (any value)

A semaphore is an integer value used for signalling amongst processes

Two operations:- wait () → P
signal () → V

Semaphore (S) → resource

```
wait (s) {
    while (s<=10);
    s--;
}
```

```
signal (s) {
    s++;
}
```