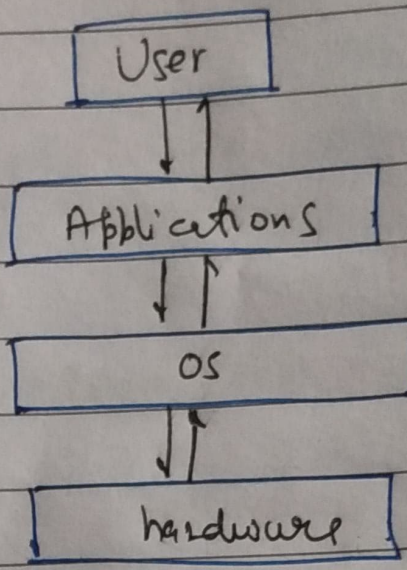
system software → Operating System

Page No.
Date

- Interface b/w User & hardware



- performs all basic tasks like
  - → Resource Management (memory / process / like
  - → Abstraction → Protection
  - → handling input / output
  - → controlling peripheral devices

example → Windows, Linux, macOS
→ Android, IOS

Single Tasking
Multi Tasking
Multi Threading
Multiple Processing
Multi User

Types of OS

1) Single Tasking (MS-DOS)
- CPU can work only to programs that are loaded into memory.
- here, only 1 prous other than OS can exist.
- inefficient (CPU is fast, I/o devices slow)

2) Multi-programing & Multi-Tasking
- have multiple prous in memory and they can be run concurrently. (Priority, assigning)

3) Multi-Threading
- multiple threads running within a prous
- more responsive - better CPU utilization

[Thread → smallest unit of execution that can be assigned to CPU → smaller than proces]

- Prous switching is costly. Thread solves this problem. less costly.

Single processor

4) Multiple-processing OS
- used mostly.

- Latest Windows have all the features - 2, 3, 4, 5

5) Multi-User
- Unix, latest Win
- multiple user can access same OS

Page No.
Date

MultiTasking: multiple task / process at same time
ex listem to music, browse web

MultiThreading: multiple things within a process
ex download something in browser & brow

ex **MS Word**: Typing, saving, formatting, spellchecking at same time

ex **WebServer**: Apache httpServer used thread pools
ex **IDE**: compiler error also
ex **Games**: handling multiple objects

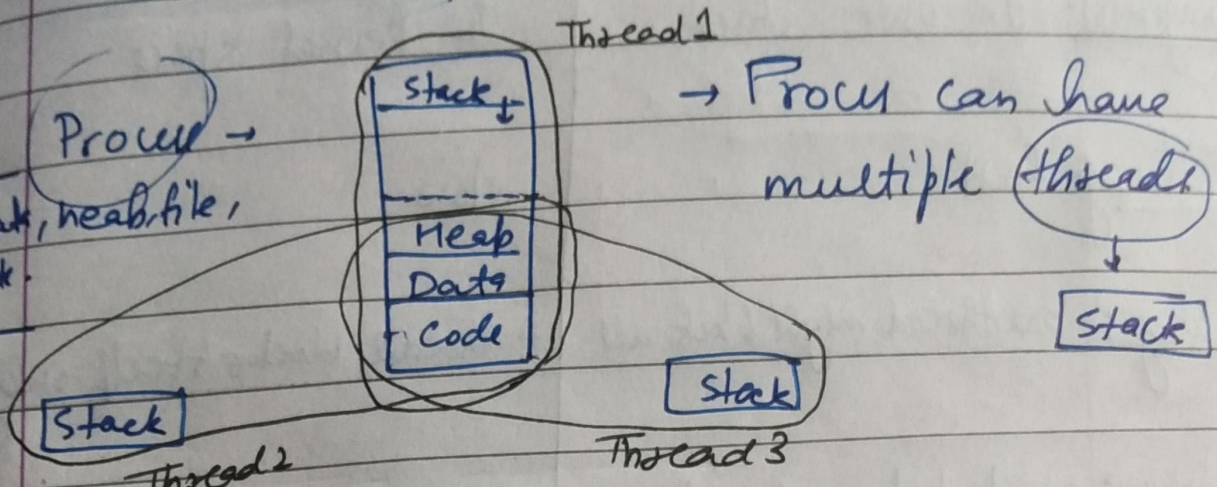
Adv

① Parallelism ② Responsiveness
③ Better CPU utilization ④ saves cost on process switching

DisAdv

① Diff to write Test & Debug Code
② lead to deadlocks & race cond^n, when have same variables

→ code loaded from harddisk to RAM to become a process.



→ Thread is only stack ∴ is lightweight

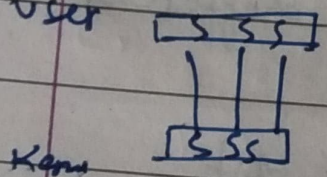
virtual to

- mapping from physical address
  - These mapping is per process
- ∴ threads do not need different mapping in one process
- Easy to create / terminate - fast
- Process switching is expensive.

Page No.
Date

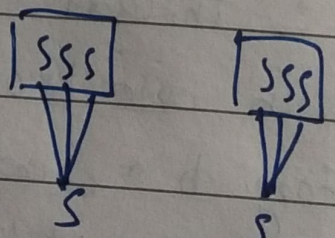
User Thread vs Kernel Thread

	User Thread	Kernel Thread
Management:	In user space	In kernel space
Context switching:	fast	slow
Blocking:	one thread might block all	A thread blocks itself only.
Multiprocessor:	can't take adv of multiprocessor	Take full adv.
Creation / Termination:	fast	slow

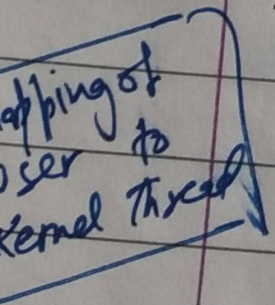


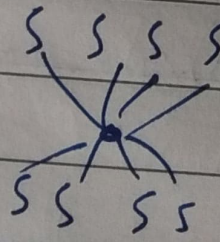
[most comm

one to one



many to one





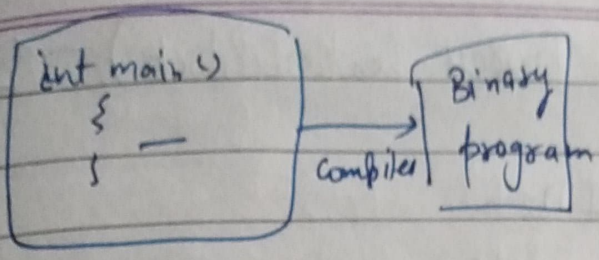
many to many

libraries → Java thread library, Posset. (C, C++)] User Threads

o.s assign process to CPU

Page No.
Date

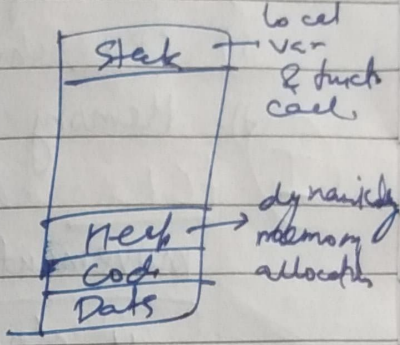
Program



Process

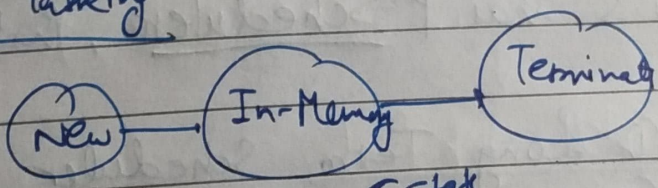
Binary program is loaded into RAM to run by processor

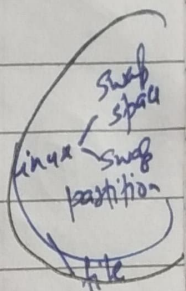
A process is a program in execution



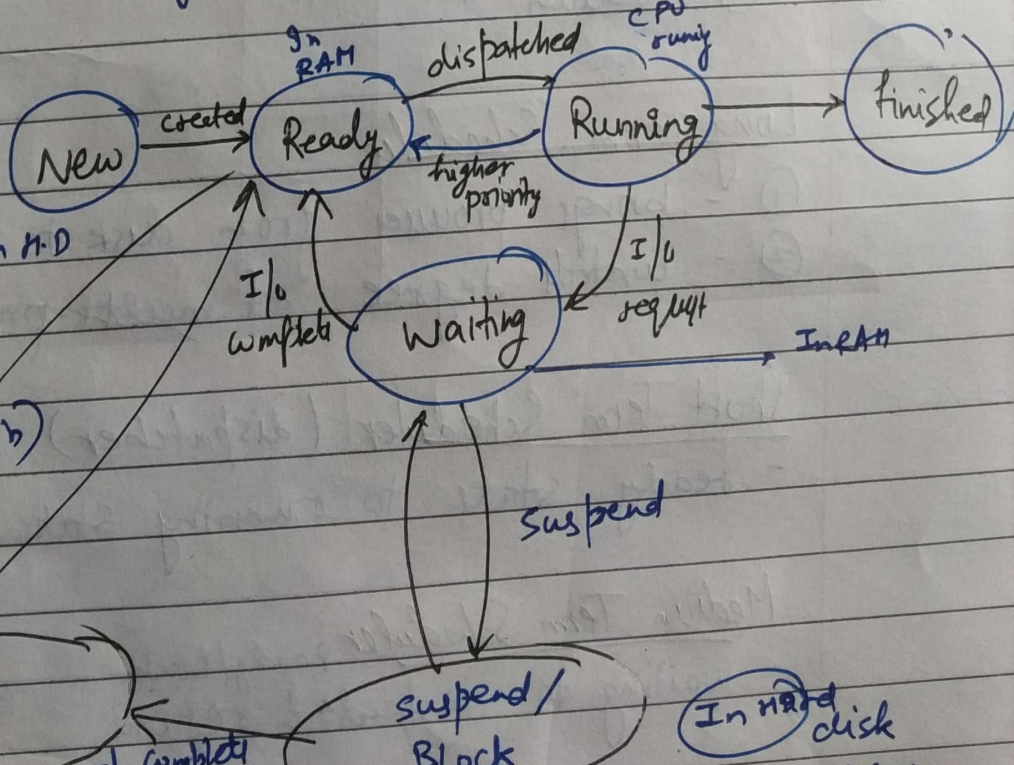
Process States

① Single Tasking





② multiprograming ← 5 state / 7 state



Page No.
Date

PCB (Process Control Block) – data structure

1) process ID
2) process State
3) CPU register
4) Accounter information
5) I/O information
6) CPU scheduling information
7) Memory information.

movement of states (one State to other)
it done by O.S.
O.S has P **Scheduler**, that do. →

↳ ① Long Term Scheduler
② Short " "
③ Medium " "

Long Term Scheduler
① - brings process from disk to RAM
② - controls degree of multi-programming

Short Term Scheduler (dispatcher)
- ready state to running state

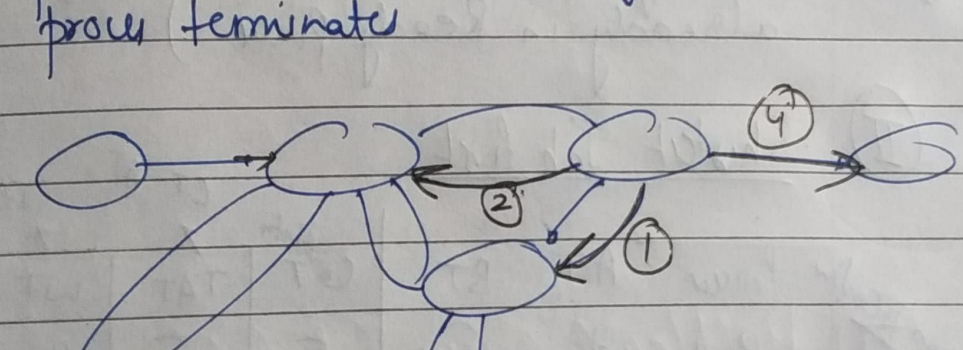
Medium Term Scheduler Ready/Block.
- waiting to suspend & resume

Page No.
Date

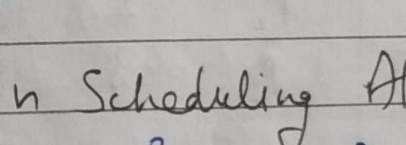
Queues
- Ready Queue
- Job Queue
- I/O Queue

Scheduling Algorithim
- for short term scheduler or for job in ready queue

When a process is picked by scheduler?
1. when some process moves from running to waiting
2. running to ready
3. process is moved to ready state (new →)
4. process terminates



Different time in Scheduling Algorithm



1. Arrival Time (1)
2. Completion time (6)
3. Burst time - time taken by CPU for a process (2+1) = 3
4. Turn Around Time (6-1) = 5 [Start to end / completion T - Arrival T]
5. Waiting Time - time process spends in ready queue (5-3) = 2

TAT - BT

6. Response Time → 0 [Arrival time & first time when it gets to CPU]

Page No.
Date

Goals of Scheduling Algorithm

1. Max CPU Utilization
2. Max Throughput → no. of jobs finished per unit of time
3. Min TAT
4. Min WT
5. Min RT
6. Fair CPU Allocation (No starvation)

(I) FCFS Scheduling (first come first serve)

- non-preemptive

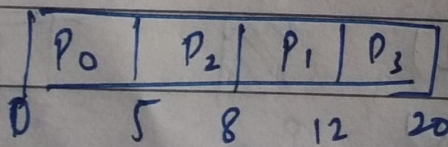
(Talking about process that are already in Ready Queue)

(Still Not Batch process)

(II) Shortest Job First

- non-preemptive

Process	AT	BT	CT (Completion time)	TAT (CT − AT)	WT (TAT − BT)
P_0	0	5	5	5	0
P_1	1	4	12	11	7
P_2	2	3	8	6	3
P_3	3	8	20	17	9



Avg TAT

$\Rightarrow \frac{5+11+6+17}{4}$

$\Rightarrow 39/4$

Avg WT = $\frac{0+7+3+9}{4}$

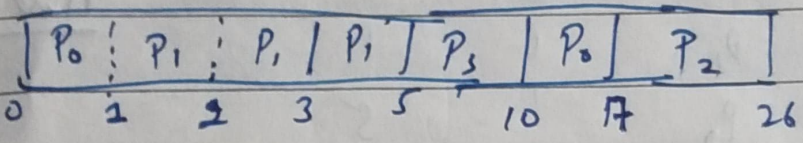
- preemptive Shortest Job first

(III) Shortest Remaining Time first.

SRTF

Process	AT	BT
P_0	0	8
P_1	1	4
P_2	2	9
P_3	3	5

Whenever a new job arrive, we compare



- Impractical many time - better Avg WT

(IV) Priority Scheduling { ~~non-preemptive~~ / pre-emptive

If same priority, give priority to which come first

Starvation of low priority

Sol - Aging → Increase priority if it is in queue for a long time. (WT)

- In FCFS we consider AT as priority
- In SJF we " BT " "

Page No.
Date

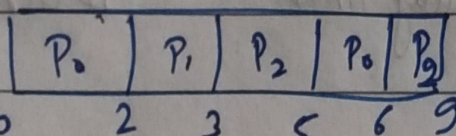
(V) Round Robin Scheduling - preemptive

Ready queue (circular)

Time quantum
- smaller
- larger (FCFS)

Process	AT	BT
P_0	0	3
P_1	1	1
P_2	1	5

Time quantum = 2



- Avg WT ↑
- every process get some response
- no starvation

(VI) Multilevel queue Scheduling

- have diff queue
- put diff process in diff queue
- flexibility to apply diff scheduling algo in diff queue
- dynamic (feedback queue)
  - Process can move b/w queues.

Page No.
Date

Process Synchronization

Global variable

P1 P2

Race cond^n

Critical Section

When accessing the shared global value

To prevent Race Cond^n

disabling interrupt (✗)

(1) Locks (Mutex) ← software / hardware ✓ fast → Peterson (1) / Bakery (multiple)

(2) Semaphore - wait & signal.

(3) Monitor - manage threads — class { sync } ≡ sync (java)

(I) Lock

```
int amount = 100;
bool lock = false
```

Software ⇒

P1

```
void deposit (int x)
{
    while (lock) {wait}
    lock = true;
    amount += x;
    lock = false
}
```

P2

```
void withraw ()
{
    ==
    amount -= x;
}
```

(✗) No mutual Exclusion — Problem

hardware ⇒ TSL Lock

```
void deposit (int x)
{
    while (test(lock)) {wait}
    amount += x;
    lock = false;
}
```

```
bool test (bool * lock)
{ bool old = *lock
  *lock = true;
  return old;
}
```

test & set

In Java → busy waiting
- check for lock to open
- no time bound

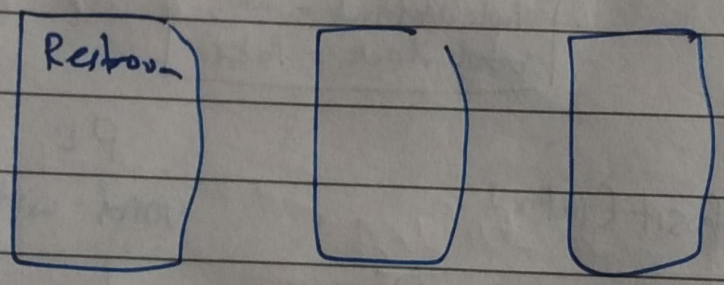
Page No.
Date

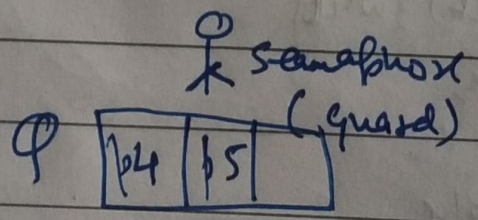
(II) Semaphore — count var & queue

count var → no. of available resource

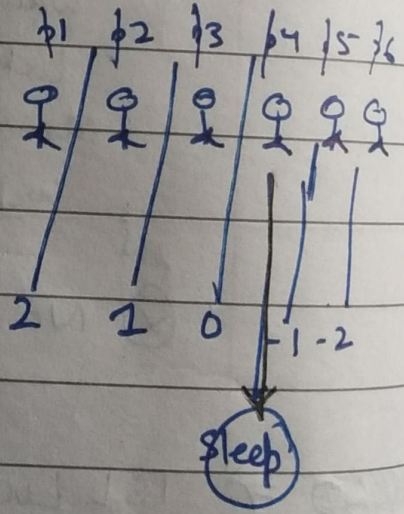
wait() → before using resource (−1) count
signal() → after using resource (+1)

```
if (count = 0)
    { don't wait
      sleep(); // I will wake you up
    }
```





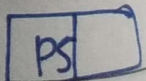
count = 3



let p2 left restroom

(+1) wakes process from queue

count = -1



```
struct sem {
    int count
    queue q;
}

sem $ (3, empty)
```

```
void wait ()
{   s.count--;
    if (s.count<0
    { ① Add caller proc in Q
      ② sleep (1);
    }
}
```

```
void signal ()
{   s.count++
    if( s.count <= 0)
    { ① Remove a proc from q
      ② wakeup (p);
    }
}
```

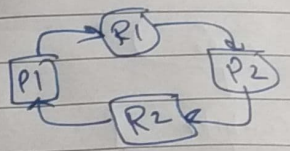
(III) Monitors class {

= sync
= sync
= sync

}

- built on top of locks
- implemented by multithreading systems

Deadlock



Prevention

1) Remove Mutual Exclusion (Pr
   ↳ coz some resource can't be shared
   ↳ Spooling → printer (Job Queue) - push (full)
   (minimize mutual exclusion)
2) Remove hold & wait
3) Remove circular wait cond.
4) Preemption of resource

Avoidance

1) Wait & Die
2) Wound & Wait
3) Banker's Algo – Resource Allocatio

~~Don't Do Anything~~ - Ostrich Algo