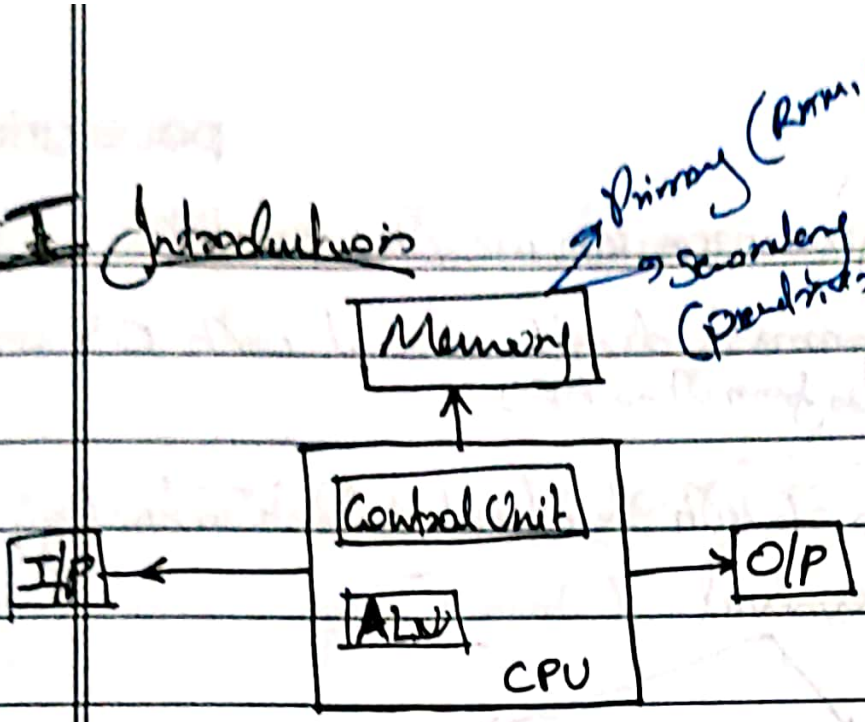
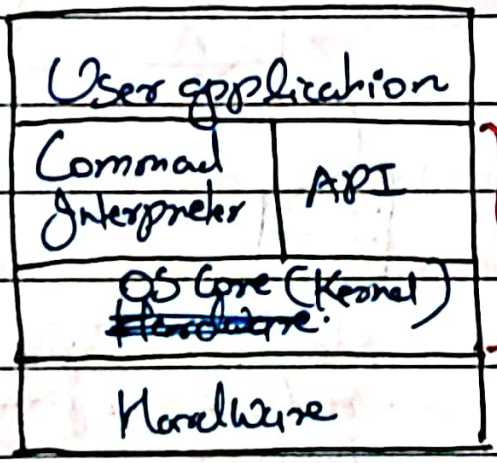
Date: / /

I Introduction



- Essential computing model of Von-Neumann architecture & Stored Program Concept
  (any program to be executed must be stored in Primary Memory)
- Stored Program Concept follows Sequential flow of Control
  (One program is stored in memory, program instructions will be executed purely on a sequential basis)



- Command Interpreter – Interprets commands submitted by user applications. Eg in text based OS – Shell ($) in LINUX & UNIX, In GUI based OS, Desktop with icons, etc.

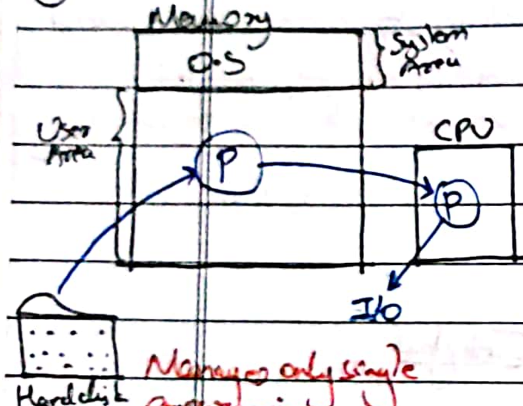
papergrid Date: / /

- API - Programmers can use this API in their programs to directly interact with OS service.
  Eg: fork(); is from this API.

- Types of OS - With the advent of Disk Technology 2 major OS evolved

UNI-PROGRAMMED OS

Eg: Disk OS (DOS)

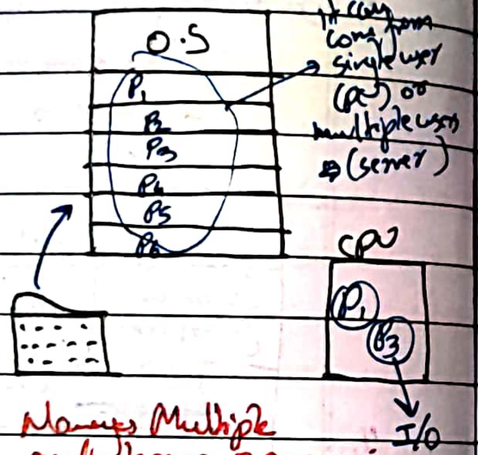


Memory only single program in memory.

Here when P goes for some I/O operation CPU becomes idle.

MULTI-PROGRAMMED OS

Eg: Unix, Linux, Windows, etc.



Memory Multiple ready to run programs in Memory.

Here when P3 goes for I/O operation P1 is loaded from memory & processed by CPU, thus reducing idle time of CPU (max CPU utilization)

In both OS, CPU can process only one program at a time, but in multi-programmed OS, time is efficiently divided b/w programs.

papergrid Date: / /

- Throughput - (No. of programs/processes completed per unit time.)

- Multiprogrammed OS → Non-Preemptive (until program completes/performs I/O, CPU can't be released) → Pre-Emptive (in preemption, CPU can be released by a program forcefully) (It improves responsiveness of interactiveness)

- We have our objective of this course, for the study of design & implementation policy of a typical Pre-emptive based Multi-programmed Operating System which can run on a Von-Neumann architecture.

* Architectural Requirements of Multi-Programmed OS

1. IO Devices : I-O devices supports different modes of data transfer → ① Program Controlled (CPU regularly checks if IO wants to communicate with CPU)
   ② Interrupt Initiated (I/O creates interrupts, so CPU listen to I/O). (discussed in COA)
   But these 2 requires CPU.
   ③ DMA (Direct Memory Access) doesn't require CPU.

Using this I/O and Memory can directly communicate with each other. So I/O prefer DMA setup for Multiprogrammed OS.

2. Memory – All memory systems should have provisions of Address Translation.
↓ discussed in Memory Management Chapter later.

3. CPU – Every processor participating in a multiprogrammed OS environment should have the capability to support dual mode operation.

1. User Mode / Non Privilege Mode
↓ it is Pre-Emptive

2. Kernel Mode / Privilege Mode / Supervisory Mode.
↓ It is non Preemptive

- In the Control Word called Program / Process Status Word (PSW), one bit is called Mode bit, which specifies whether the processor is in user / kernel mode.

- Whenever any user application program needs any OS service, immediately mode is shifted to kernel, the OS service is executed & mode shifted back to user mode. This Mode Shifting is a very important activity that happens a lot of times during the course of using an user application.

* Mode Shifting

(Eg) main() {
int a, b, c;
a = 1;
b = 2;
c = a + b;
fork();
f(c);
}

(a = 1; b = 2; c = a + b; → user mode; fork(); → kernel mode; f(c); → user mode; f is an user defined function.)

This function is called System call. → System call is a call to OS for availing an OS service.

During compilation the compiler will translate every high level language system call into a special privilege instruction called as SVC (Supervisory Call)
↓
1. During runtime execution of system call will generate a software interrupt
2. Every interrupt has an Interrupt Service Routine (ISR) (discussed in COA) – Here the running user process is blocked
3. ISR → first goes to PSW & changes the Mode bit to Kernel Mode.
4. ISR then consults a table & finds the address of fork() system call instructions in the Memory. This address is put into program counter & from there we go to the address & execute fork().
At the end of fork() instruction is a SVC which changes the mode bit back to user mode and unblocks the process.
5. Since ISR has the return address, we return to the next instruction (here, f(c);)

- Thus here we switch from user to kernel mode & back to user mode.

User programs are service users. OS is a service provider.
} Thus mode shifting is used for services to reach the user programs. This Sys. call is like an interface b/w the user application & OS services.

Fork System Call

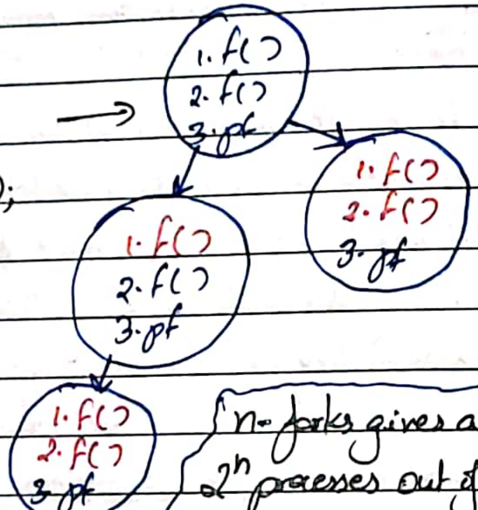
(Eg)
```
main(){
    printf("Before");
    fork();
    printf("Hello");
}
```
→ Parent: pf(Before); fork(); pf(Hello);

Child: pf(Before); fork(); pf(Hello); — Child process will have the same code, but only start executing from the next statement after the fork() that created it.

Output: Before Hello Hello.

We can't know if the first Hello is from parent or child, depends on processing.

(eg2)
```
main(){
1.  fork();
2.  fork();
3.  pf(Hello);
}
```
→



Note: variables of parent & child will be stored differently in memory.

n-forks gives a total of 2^n processes out of which 1 is parent & $2^n - 1$ are child processes.

- Return Value of fork()
  - (-ve) error, fork() failed due to create child process, Maybe due to insufficient resources in OS, etc.
  - child (0)
  - parent (>0, +ve)

(Practice Qs based on fork() — Visit page-115)

III Process Management

* Process Concepts : Program refers to .exe file
- Instructions
- data → dynamic (memory allocated during runtime in heap)
  → Static (memory allocated before runtime in static memory)

- Program v/s Process (Difference)

What is a process? Ans → 1. Program in execution → Execution doesn't mean while running on CPU, it means while utilizing resources of a computer. So if a program is in memory / I/O operation also it is in execution.
2. From a single program, we can have multiple processes.
3. Process is a unit of CPU utilization.
4. Process is a schedulable unit — Act of making a selection & giving control of the process to the CPU is called scheduling.
5. Process is a locus of control of OS
6. Animated Spirit — Program is just a passive entity on disk. Process is an active entity on Memory.

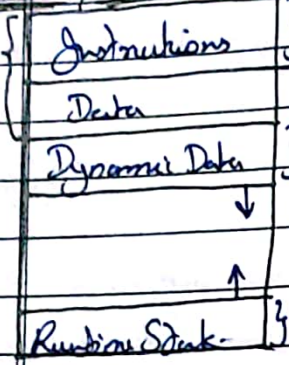
- Developer Perspective — Process as an ADT (Abstract Data type)

Process = < its Definition, its Representation or Structure, will support a defined set of operations, will also be associated with a well defined set of Attributes >

papergrid

Date: / /

Process View / Structure in Memory



* Process Operations

(i) Create(): (Resource allocation, min resources to start process)
(ii) Schedule()
(iii) Run()
(iv) Block()
(v) Suspend() (push process out of memory)
(vi) Resume() (resume suspended process)
(vii) Terminate() (resource deallocation)

* Process Attributes

(i) Identification: process id, group id, parent id, etc.
(ii) CPU related attributes: Eg: Type/category of process, process priority, program counter value, state of process etc.
(iii) Memory related attributes: Size of the process, Memory limits (Static & ending address etc)
(iv) File related attributes: List of files used by the process.
(v) Device: List of Devices used by the process.

papergrid

Date:

- Different OS can have diff. set of attributes. These attributes are kept in a Table called PCB (Process Control Block) which is a very important data structure (or) (Process Description)

Pid, Gid, etc.	Links → link discussed later.
Type	State → many states
Priority	Program Counter
Size	File → These volume of info in PCB is called Process context / Process Environment (or) Content of PCB.
Device	....
.....	.....
.....	.....

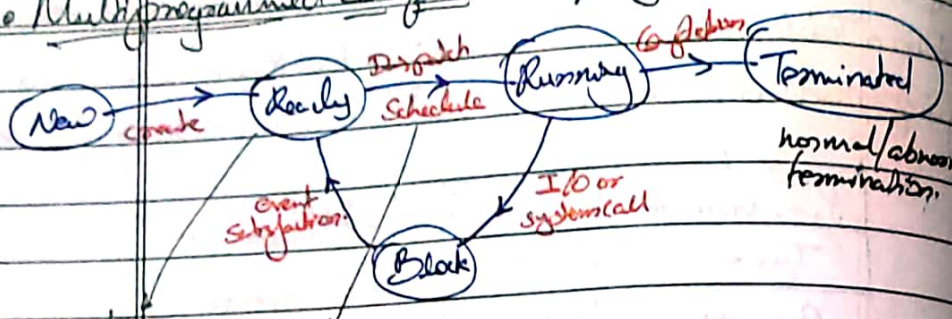
- Each process will have its own unique PCB
- PCB is an important Data structure maintained by OS on behalf of the process.
- PCB is like the id card of the process. It is associated to the process from its creation to its termination.

* Process Transition Diagrams

Different States of a process → New, Ready, Running, Block, Suspend, Terminate, Suspend Ready

(Non Preemptive)

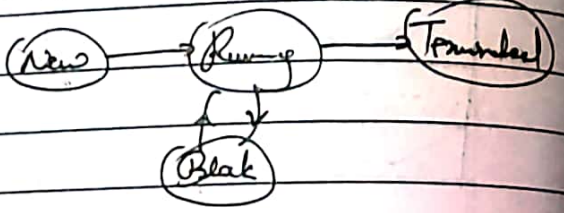
• Multiprogrammed OS process state diagram.



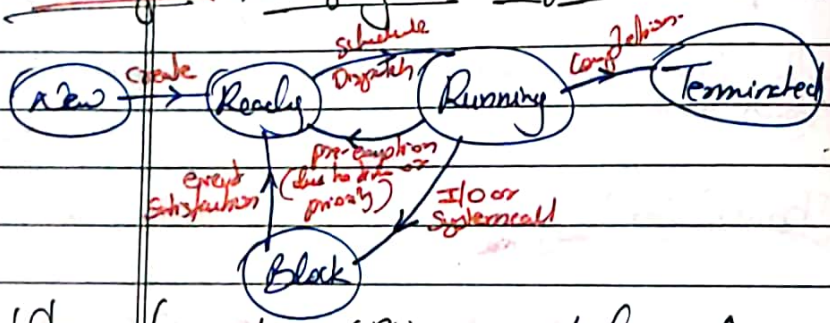
There can be multiple processes ready to run.

Schedule selects one of the processes to run on CPU.

• Uni-programmed OS → (no Ready state)



• Pre-Emptive Multiprogrammed OS process state transition diagram.

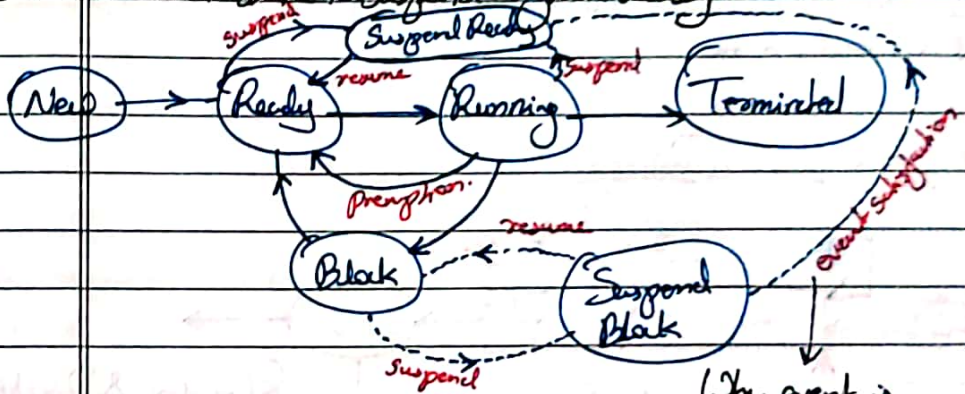


• When we have only one CPU, we can only have 1 process in the Running state. At a time the process is Ready, Blocked/Running state, the process will be present in the memory.

• Sometimes the OS will have to swap out some of the partially completed processes from the memory to the secondary storage like disk (Swap out/Suspend).

This improves the degree/level of performance of the system due to deadlock of resources. Here the process will be temporarily suspended for sometime. This process is called Swap out/suspend. Later when the OS believes that it has enough room/resources, it resumes the process back so as to continue its execution from where it got suspended. (it is not restarted, it continues from the same point).

• We can suspend from Ready, Running or Blocked state. The most desirable state to suspend is from Ready.



When event is satisfied, blocked changes to ready state. But since it is suspended it changes to Suspend ready state.

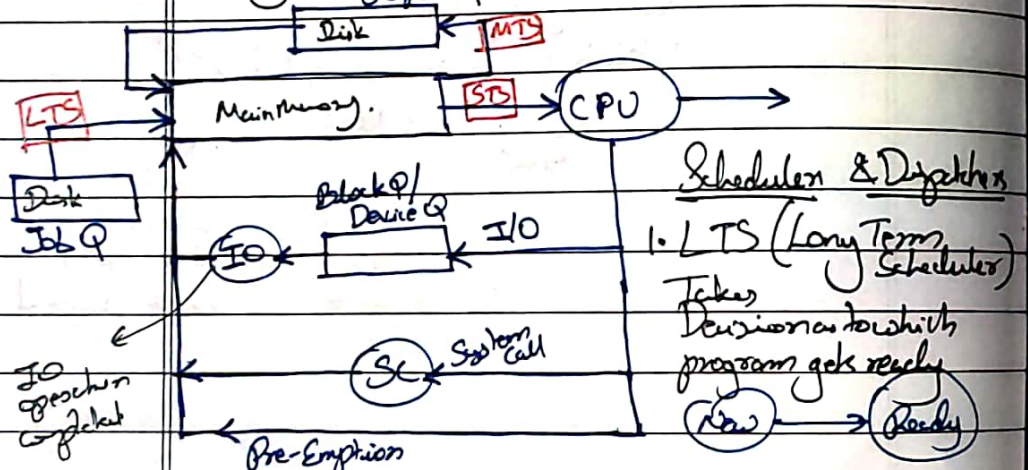
papergrid

Date: / /

* Scheduling Queues

- In-Memory Queue
  - Ready Queue: Queue of PCBs of processes that are ready to run. In Implementation wise it is a linked list (singly, doubly/circular) of PCBs ready to run.
  - Block Queue (Device Queue): I/O For device for I/O operation, it keeps its own Queue of those process that got blocked due to its I/O operation (One queue per device)
- On Disk Queue
  - Suspend Queue (List of suspended processes)
  - Job Queue (New - Ready to be loaded to memory)

* State Queuing Diagram



Scheduler & Dispatcher

1. LTS (Long Term Scheduler)
   Takes Decision as to which program gets ready
   New → Ready

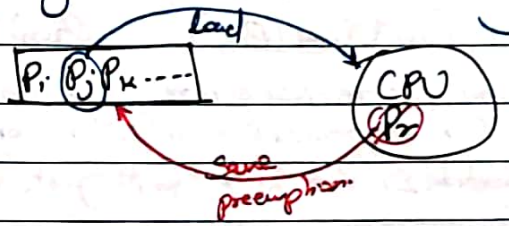
papergrid

Date: / /

2. STS (Short Term Scheduler) / CPU Scheduler
   Ready → Running decision

3. MTS (Medium Term Scheduler) - Responsible for swapping out the programs from disk to memory & vice versa.

- Degree of Multiprogramming → No. of processes in the Memory System at this is controlled by LTS.
- Dispatcher : It carries out Context Switching



Time required by the Dispatcher to load the P_j PCB and save the PCB (P_i) is called Context Switching Time / Dispatch Latency

papergrid
Date: / /

* CPU Scheduling

Goals of STS / CPU scheduler
1. Maximize CPU utilization
2. Minimize: WT, TAT, Response Time

WT_1	BT_1	$IOBT_1$	WT_2	BT_2	WT_3	BT_3
RQ	CPU	IO	RQ	CPU	RQ	CPU

(AT) — (TAT) — (CT)

$BT = BT_1 + BT_2 + BT_3$
$WT = WT_1 + WT_2 + WT_3$
$IOBT = IOBT_1$

Arrival Time (AT), Completion Time (CT)
Turn Around Time (TAT) = CT − AT

Waiting Time (WT), Burst Time (BT), IO Burst Time (IOBT)

Response Time → Time Hw receive of a req. by a process & the time at which it provides its initial response

~~(Time spent Hw ready state & getting the CPU for the first time)~~

$$WT = TAT - (BT + IOBT)$$

- No: of processes = n (P_i, i from 1 to n)
- A_i = AT of process P_i · C_i = CT of process C_i
- WT (P_i) = WT of process P_i · X_i → BT of P_i · Y_i → IOBT of P_i

$$\text{Avg. TAT} = \frac{1}{n}\sum_{i=1}^{n}(C_i - A_i)$$

$$\text{Avg. WT} = \frac{1}{n}\sum_{i=1}^{n}\left[(C_i - A_i) - (X_i + Y_i)\right]$$

papergrid
Date: / /

→ Schedule Length (L)

- For 'n' processes we can have n! no: of schedules which are non-preemptive.
- For 'n' process theoretically we can have infinite no: of preemptive schedules

(Time taken by CPU to complete the process)

$$\text{Schedule Length } (L) = \text{Max}(C_i) - \text{Min}(A_i)$$

Max(Ci): Completion time of last process
Min(Ai): Arrival time of first process

$$\text{Throughput } (\mu) = \frac{n}{L} \quad (n = \text{no: of processes})$$

1. First Come First Serve (Scheduling Technique) (FCFS)

Here Selection criteria : Arrival Time.
Mode of operation is inherently non-preemptive.
If two processes have equal AT, then Conflict resolution needs to be implemented, for eg:- lower process id.
(Problems should use Gantt chart, dt. pg 15 of note OS, if you want to check).

- Schedule Overhead (δ) - Amount of time required by the process to come from the Ready Queue to the CPU is δ. (In some question δ will be given, otherwise 0)

papergrid

Date: / /

- %CPU overhead = Total 'S' × 100 / L = x
- %CPU Idleness = idle time of CPU × 100 / L = y
- CPU efficiency = 100 − (x+y)

for better understanding visit Page no: (17) 02

2. Shortest Job First (SJF) or Shortest Process Next (SPN)

Selection Criteria : Process with the least CPU BT is taken first.

Mode of Operation : Non-Preemptive.

Conflict Resolution : Will be given in the Question

3. Shortest Remaining Time First (SRTF) (imp → Gate Q's)

(Pre-Emptive Shortest Job First)

gives min. avg. waiting time. (Pg 113)

- Selection Criteria : Burst Time (Shortest)
- Mode of Operation : Pre-Emptive.
- Conflict Resolution : Given.

• When S=0 the Scheduling length for both SRTF & SJF will be the same and the performance of SRTF wrt. Avg TAT will be better than SJF when S=0. But when S>0 then its possible that performance of SJF may become better than SRTF because more preemption in SRTF.

papergrid

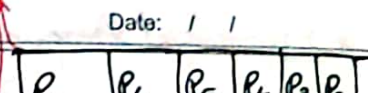
Date: / /

Gantt charts.

(Eg)

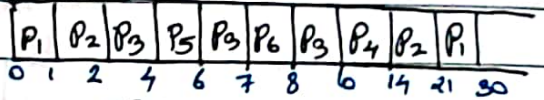
Pno	AT	BT
1	0	10
2	1	8
3	2	5
4	3	4
5	4	2
6	7	1

(SJF) → CPU →



L = 30

(SRTF) → CPU →



L = 30

TAT(P1) = 30 TAT(P2) = 20

TAT(P3) = 8

4. Longest Remaining Time First (Criteria: BT, Mode: Pre-emptive) (only one Gate Q)

→ SJF (Performance)

Drawback → leads to starvation to longer processes

Advantages

→ Maximum Throughput

→ Minimization of Avg. TAT and Avg. WT

In view of these 2 we can say that SJF is an optimal algorithm.

→ Practical limitations of SJF → Since BT cannot be known Apriori, we can't implement SJF. But we can implement using predicted BTs.

Predict using Exponential Averaging Technique (PTO)

∴ SJF is used as a Benchmark to measure the performance of other Scheduling techniques.

papergrid

Date: / /

* Exponential Averaging Technique

Let P → be a process t_i = BT_i of the Process.

τ_i = Predicted BT_i of Process

τ_{n+1} = Next Predicted BT_{n+1} of the Process.

τ_1	t_1		τ_2	t_2	τ_3
WT_1	BT_1	IOBT	WT_2	BT_2	WT_3
RQ	CPU	IO	RQ	CPU	RQ

P → (row above); preempt.

t_1 is predicted as τ_1 during WT_1 before BT_1

Based on averaging algo. average

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

$0 < \alpha < 1$

The terminating cond^n of this recurrence eq is τ_1 and its value is called the seed value.

5. Priority Based Scheduling

Criteria: Priority
Mode: Non-Preemptive / Preemptive

• Priorities can be either Static & Dynamic. Static priority once assigned cannot be changed, this can cause problems like starvation to lower priority processes, so to overcome that we can use dynamic priority, where the priority of a process goes on increasing as time passes & at some point it definitely gets executed & is not starved. This process of dynamically increasing the priority is called aging process / aging algorithm.

papergrid

Date: / /

6. Round Robin CPU Scheduling Technique

Criteria: Arrival Time + Time Quantum Mode: Pre-Emptive

(to be completed within given Time Slice / TQ)

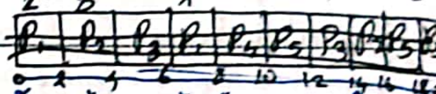
(Eg) Here the Ready Queue is maintained in First In First Out Queue

(Eg)

PNO	AT	BT
1	0	4
2	1	5
3	2	3
4	3	2
5	5	6

TQ = 2

RQ: P1 P2 P3 P1 P4 P2 P5 P3 P2 P5

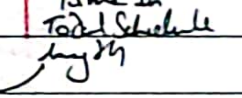


P_1	P_2	P_3	P_1	P_4	P_2	P_5	P_3	P_2	P_5	P_5

0 2 4 6 8 10 12 14 15 16 18 20

* Performance of Round Robin

Large Value of TQ	Small Value of TQ	Very large TQ (larger than largest BT of any process in the schedule)	Very Small TQ (TQ << S) S = Schedule overhead
- Less Context Switching Overhead - Poor interactiveness / responsiveness	→ More Context Switching Overhead → Improves Interactiveness or Responsiveness	This works exactly like FCFS but is not a very responsive system.	This decreases efficiency of CPU, because the total scheduling overhead predominates over the Burst Time in Total Schedule length



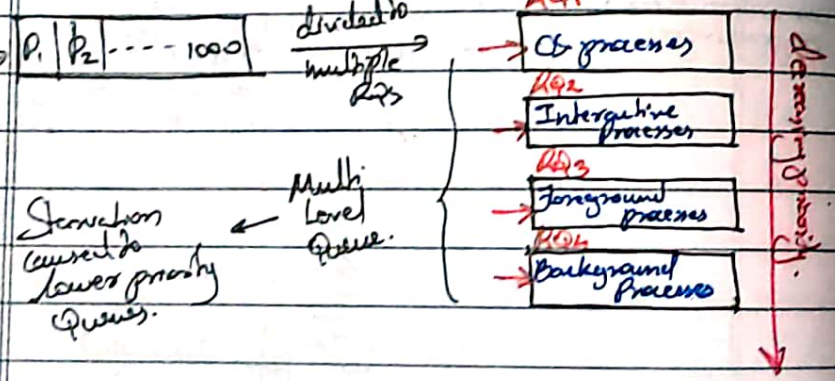
$$\frac{\Sigma S}{L} > \frac{\Sigma BT}{L}$$

(Thus efficiency drops drastically)

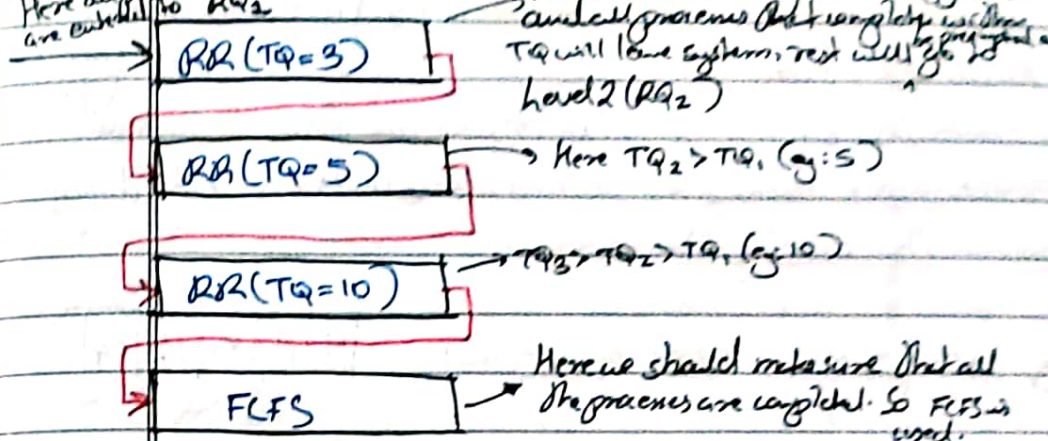
* Multi level Queue Scheduling

- All previous scheduling methods were based on Single Ready Queue System.
- Drawbacks of a single RQ → e.g. To select a particular process of a type, we have to filter out all the processes. So lot of filteration time will be involved.
- Also in single RQ, we will have to use the same scheduling technique in all types of processes.
- In multi level Queue Scheduling, we use multiple RQs, each dedicated for diff. category of processes. Now we can use diff. scheduling techniques for diff. queues.

But this may lead to Starvation → For eg: if RQ1 for OS process has highest priority then until RQ1 is empty processes in RQ2 will be starved. Thus lower Queues can experience Starvation. So to overcome this, we use Multi-Level Feedback Queue.



Multi Level Feedback Queue



Here also lower levels have lower priority. Only after the above levels are empty, process in below levels are taken up by CPU.

- Because generally the OS processes are shorter than Background & other processes, by varying TQ we can adjust them priority for the process. Here also low level Queues will be subjected to starvation but the interactiveness of this process will increase in comparison to the previous approach. Here the process finished in the last level gets a substantial amount of BT completed during their way down the levels. whereas in Multi level approach some processes might not get a chance to start at all.

* Highest Response Ratio Next (HRRN) Scheduling

Criteria: Response Ratio = $RR = \frac{w+s}{s}$ w = waiting time, s = Service time (BT + I/O)

Mode: Non-Preemptive

- Even with SJF with predicted BT, we can have starvation. HRRN favours shorter processes and also limits the waiting time of processes thus avoiding starvation.

Ex:

PNO	AT	BT
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

HRRN Gantt Chart → | P_1 | P_2 | P_3 | P_5 | P_4 |

0 3 9 13 15 20

No need to calculate RR because at these times only they become available.

Now P_3, P_4 & P_5 are available, so

$RR_3 = \frac{(9-4)+4}{4} = 2.25$

$RR_4 = \frac{(9-6)+5}{5} = 1.6$ $RR_5 = \frac{(9-8)+2}{2} = 1.5$

$RR_3 > RR_4 > RR_5$, So next P_3 executed.

Similarly P_5 & then P_4 (At time 13: $RR_4 = \frac{(13-6)+5}{5} = 2.4$

$RR_5 = \frac{(13-8)+2}{2} = 3.5$

$RR_5 > RR_4$ ∴ next P_5 & then P_4)

Service Time = CPU Burst Time + I/O Time required by a process.

(most important chapter of OS)

* Process Synchronization & Coordination

→ Inter Process Communication (IPC) (communication b/w 2 process)

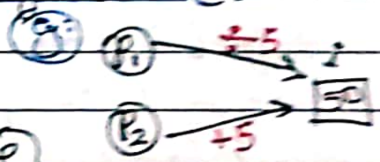
Some of the standard IPC mechanisms are –

1. Pipes – Logical entity like file present in the memory along. It has 2 ends (read, write). One process writes at one end and the other process reads at the other end.
2. FIFO Queue – In whatever order a sender writes, the reader gets it in the same order.
3. Shared Memory Segment   4. Message Passing System etc.

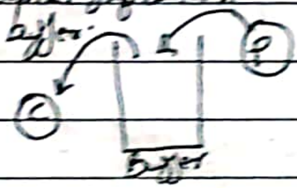
* Problems due to lack of Synchronization b/w processes in any IPC environment
1. Inconsistency   2. Loss of Data   3. Deadlocks / Lockups

* Types of Synchronization → 1. Competition (compete for shared resource)
2. Cooperation

Ex:



Ex: In Producer – Consumer Synchronization they share the same buffer.

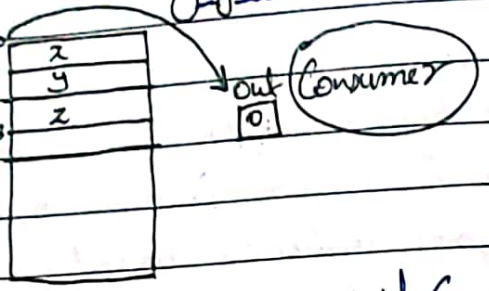


Note: In producer consumer synchronization we can find both cooperation (for buffer) & competition as explained in next page.

papergrid

Date: / /

* Case Studies → 1. Producer-Consumer Implementation




```
#define N 100
int Buffer[N];
int count = 0;

void Producer (void) {
    int itemp, in = 0;
    while (1) {
a.      itemp = Produce_item();
b.      while (count == N);
c.      Buffer[in] = itemp;
d.      in = (in+1) % N;
e.      count = count + 1;
    }
}
```

Busy waiting

Circular buffer

```
void Consumer (void) {
    int itemc, out = 0;
    while (1) {
a.      while (count == 0);
b.      itemc = Buffer[out];
c.      out = (out+1) % N;
d.      count = count - 1;
e.      Process_item(itemc);
    }
}
```


papergrid

Date:

Now consider preemption occured for Producer after (d) and preemption occured for consumer after (e). Now the next time if Producer is called first e. count = count+1 is run. Now count = count + 1 → Load Rp, count

(let count current value = 5)

[count = 5, Rp = 6]

Increment Rp

preempted here. ← Store count, Rp

Now at Consumer reaches CPU and it runs from (d)

count = count - 1 → Load Rc, count

decrement Rc

[count = 5, Rp = 6, Rc = 4]

Preempted ← Store count, Rc

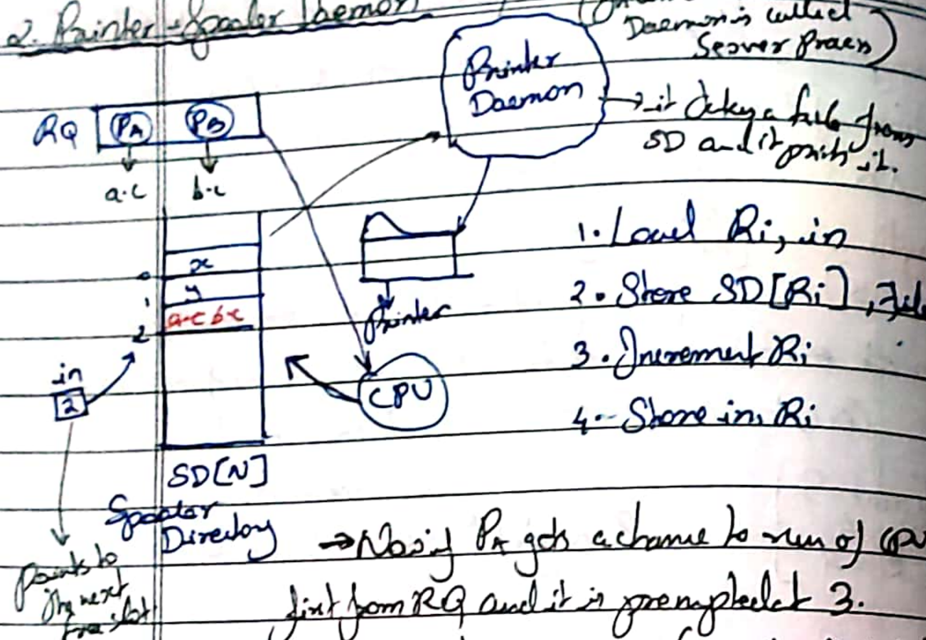
Now if P is executed first, the final value become count = 4 or if C is executed first, the final value will become count = 6 and both doesn't give the actual value that is 5

So this possibility of having a value other than (5-1+1=5) 5 can be said to be Inconsistency

Thus we need synchronization for (P) & (C) to avoid inconsistency.

2. Printer-Spooler Daemon

(Daemon: infinitely running process in the background providing some service. (In windows Daemon is called Server Process))



1. Load Ri, in
2. Store SD[Ri], FileName
3. Increment Ri
4. Store in, Ri

→ Now if PA gets a chance to run of CPU first from RQ and it is preempted at 3. Now at position 2 → we have "a.c" file to be printed and value of (in = 2) itself.

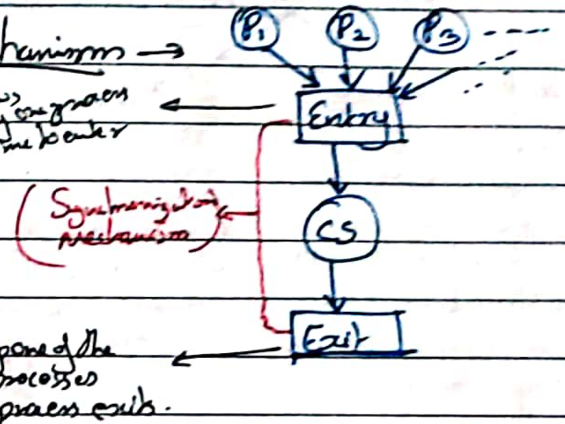
→ Now if PB gets a chance to run on CPU since in was not incremented, "b.c" file overwrites "a.c" at position in = 2.

This causes **Loss of Data**

* Necessary Conditions for Synchronization Issues to Arise

1. Critical Section (CS) (Piece of code where shared resources are accessed)
2. Race Condition (competition b/w multiple process to access shared resources)
3. Pre-Emptive Environment (Problems arise only due to preemption)

* Synchronization Mechanism →




```
VoidProcess(i){
  while(1){
  (a) NonCS;
  (b) Entry;
  (c) CS;
  (d) Exit;
  }
}
```

* Requirements of Synchronization Mechanism / Critical Section Problem

1. Mutual Exclusion: No two processes may be simultaneously present in their critical section
2. Progress: No process running outside CS must prevent other process from entering CS (If a sol" satisfies progress, then it will not have deadlock)
3. Bounded Waiting: No process has to wait forever to access its critical section.

* Classification of Synchronization Mechanisms

→ Non-Busy Waiting / Blocking

Busy Waiting: The process will be continuously monitoring the CS to find out if CS is in use or not (loop)

Non-Busy Waiting / Blocking: The process will test the availability of CS only once, if CS is free it enters, else it will get blocked. (if-then-else)

Both of these have subcategories as

Software (User Mode)
The implementation doesn't req. any special support from hardware or OS. Normal user level instructions are used.
Eg: Lock Variable, Strict Alternation, Peterson Solution, Dekkers Algorithm, etc.

Hardware
Special privileged hardware instructions are used.
Eg:- Test and Set Lock (TSL), SWAP instruction, etc.
(They execute atomically without preemption)

OS Based
Eg: Semaphore, sleep & wakeup, monitors.

→ Process Synchronization - Lock Variable (Busy Waiting, S/W, Multi process Solution)

```
void Process (int i){
  while(1){
    (a) Non-CS();
    (b) While(lock!=0);     // Entry
    (c) lock=1;
    (d) <CS>
    (e) lock=0;             // Exit
  }
}
```

(lock=0, CS free; lock=1, CS in use)

Process:

```
1. Non-CS
2. Load Ri, lock;     // Entry
3. Comp Ri, #0;
4. JNZ step2;         // Jump to step 2 if Ri ≠ 0
5. Store lock, #1;
6. <CS>
7. Store lock, #0;    // Exit
```


1. Mutual Exclusion: Not Guaranteed.
2. Progress: Guaranteed
3. Bounded Waiting: Not Guaranteed. (One process may continuously take control of the CPU & thus the other process may wait forever)

Note: Any Busy waiting mechanism, weather it satisfy all 3 req. of synchronization mech or not, wastes CPU cycles. When a process is busy waiting in the while loop until its preempted due to its max TQ, that much CPU cycles goes to waste.

→ Process Synchronization - Strict Alternation (Busy Waiting, S/W, 2 process soln, it can work with only 2 process)

```
int turn = rand(i,j);
void Process (int i){
  int j = NOT(i);
  while(1){
    (a) Non-CS();
    (b) While(turn!=i);   // Entry
    (c) <CS>;
    (d) turn=j;           // Exit
  }
}
```

Here processes i & j take turn to enter CS.

1. Mutual Exclusion: Guaranteed
2. Progress: Not Guaranteed because if one process for eg: P_0 enters & exits, now turn=1, Say P_1 dont want to enter CS and wants P_1 enter & exits only then will become 0, so that P_0 can enter again. Say if P_1 doesn't want to enter CS, P_1 blocks P_0 also from entering CS.
3. Bounded Waiting: Guaranteed.

(Also Since both lock and strict alternations are both incorrect, we take a 3rd mechanism Peterson Solution)

(Both these satisfy which are 3 requirements.)

papergrid

→ Process Synchronization – Peterson Solution (Busy Waiting, S/W, 2 process Soln)

```
#define N=2
#define TRUE 1
#define FALSE 0
int flag[N] = {FALSE};
int turn;
```

Initially both processes have flag = FALSE. When a process wants to enter CS, its flag becomes TRUE.

Only when both the processes have flag value TRUE, turn comes into play. In such a cond^n, the value of turn decides which process is to enter CS. So both the processes will try to update the value of CS and the last process to update the variable will have turn's value as its id.

But, here the first process who updates turn will get the CPU first (FCFS), the other process will busy wait.

```
void Process (int i){
    int j = NOT(i);
    while (1){
        (a) Non CS();
        (b) flag[i] = TRUE;
        (c) turn = i;
        (d) while (flag[j] = True && turn = i);
        (e) <CS>;
        (f) flag[i] = FALSE;
    }
}
```

(while the other process is interested & this process is the last to update turn, then → Busy wait)

1. Mutual Exclusion: Guaranteed
2. Progress: Guaranteed
3. Bounded Waiting: Guaranteed

Drawbacks → 1. Only 2 processes
2. Busy waiting leads to loss of CPU cycles.

papergrid

Date: / /

* Synchronization Hardware Solutions

1. TSL (Test and Set Lock)
2. Swap (both these are privileged instructions supported by the h/w architecture that executes atomically)

Both of these are → • Busy Waiting • H/w Sol^n • Multi-process Soln • Lock based Sol^n → [Lock = True (CS in use), False (CS is free)]

(a) TSL

```
• Bool TSL(Bool *target){
      Bool returnValue;
      returnValue = *target;
      *target = True;
      return (returnValue);
  }
```

(Atomic Execution, no preemption in this) → (return the value of target before changing it)

↓

```
• Bool lock = False;
  void Process (int i){
      while (1){
          (a) NonCS();
          (b) While (TSL(&lock) == TRUE);
          (c) <CS>
          (d) lock = FALSE;
      }
  }
```

1. Mutual Exclusion: Guaranteed
2. Progress: Guaranteed
3. Bounded Waiting: Not Guaranteed (due to this particular implementation)

③ SWAP (TSL is only lock, SWAP is Lock-Key mechanism)

```
Bool lock = False;   → Global lock
void Process(int i) {
    Bool key;   → local key / each process with its own key.
    While(1) {
        a) Non-CS();
        b) Key = True;
        c) While(Key == True) {
               SWAP(&lock, &Key);
           }
        d) <CS>;
        e) lock = False;
    }
}
```

1. Mutual Exclusion: Guaranteed
2. Progress: Guaranteed
3. Bounded Waiting: Not Guaranteed for this particular implementation

Note: For the actual algo/implementation of TSL & SWAP, Bounded waiting is also guaranteed ⇒ So in general TSL and SWAP are correct solutions.

* Priority Inversion Problem — Occurs in Preemptive priority based Scheduling & Busy waiting sol's (Peterson, Dekker, TSL, SWAP)

~~Consider P_L having lower priority than P_H. While P_L running on CPU inside CS and P_H enters RQ, P_L will be preempted and P_H will take control of CPU. But since P_L is inside on CS, P_H won't be able to access CS (if it needs accessing CS) and P_H will busy wait until P_L finishes inside CS. But since P_H priority > P_L priority, P_L could never get~~ (Pg no: 113)

~~CPU control until P_H ends. Thus this leads to a~~

Livelock Situation (Here processes will be either running/ready in livelock, in deadlock the states will be blocked)

The priority inversion can be solved by Priority Inheritance (developed by NASA). P_L inherits the priority of P_H, finishes and when P_L comes out P_H gets its higher priority back from P_L.

* OS Based

* Sleep() and wakeup(), → (To avoid busy waiting)

① primitives → sleep() instruction blocks a process & adds it into a separate blocked Queue.

wakeup() (wakes a process from the front of the blocking Queue)

(Eg) Producer-Consumer problem implementation using Sleep & Wakeup to avoid Busy waiting.

```
void Producer(void) {
    int temp, in=0;
    while (True) {
        a. itemp = Produce.item();
        b. if(count==N) sleep();
        c. Buffer[in] = itemp;
        d. in = (in+1) % N;
        e. Count = Count+1;
        f. if(count==1) Wakeup(Consumer);
    }
}
```

```
void Consumer(void) {
    int itemc, out=0;
    while (1) {
        a. if(count==0) sleep();
        b. itemc = Buffer[out];
        c. out = (out+1) % N;
        d. count = count-1
        e. if(count==N-1) Wakeup(Producer);
        f. Process.item(itemc);
    }
}
```

But on analysis - we can have a Deadlock State here.

For e.g. if Consumer is executed first when buffer == 0. Said to after if (count == 0) & just before sleep() let Consumer be preempted.

Now let producer get CPU and it fills the buffer after N iterations (any wakeup call will go waste because there are no sleeping processes in the queue (blocked)).

Now when count == N, Producer goes to sleep & Consumer gets CPU again and immediately it also goes to sleep.

Now both C and P are sleeping forever & we have a deadlock situation.

Thus Sleep and Wakeup suffers from both Inconsistency & Deadlock problems.

- To avoid premature preemption of Sleep() & Wakeup, we can make them atomic and thus avoid deadlocks. This will be discussed later.

* Semaphores (synch. prim.) → Counting Semaphore (Int values from −∞ to ∞) / → Binary Semaphores (int only '0' & '1')

(Software sol. for synchronization)

→ Semaphores are a General Purpose Utility implemented within the OS kernel (it is an OS resource) primarily used for solving the problem of CS but also used for solving synch. req. of classical IPC problems, concurrency mech., etc.

→ Semaphore is also called Dijkstra's Synchronization variable.

→ Implemented as an ADT.

→ Semaphore operations: Down (or Wait, or P) / Up (or Signal, or V)

⇒ Semaphore operations are atomic.

* Counting Semaphores (CSEM) <−∞ to ∞>

- Typedef Struct {
  int value;
  Queue Type L; → (Blocked queue: for the blocked during Down operation)
  } CSEM;

```
CSEM S;
S.value = 4; // Semaphore should be initialized by user
Down(S);
<CS> // Some statement
```

(Application)

- Down (CSEM S) {
  1. S.value = S.value − 1
  2. if (S.value < 0) {
  put this process (PCB) in the Queue L (S.L) and Block the process by executing sleep();
  }
  else return;
  }

- Up (CSEM S) {
  1. S.value = S.value + 1
  2. if (S.value ≤ 0) {
  Select a process from the Queue L and wakeup (S.L)
  }
  else return;
  }

After carrying out a set of Down op. if the value of S becomes some negative value then that value = no. of blocked processes in Queue.

papergrid

Date: / /

* Binary Semaphores (0 or 1)

```
Typedef Struct {
    enum Value(0,1);
    QueueType L;   → Blocked Queue.
} BSEM;
```

```
BSEM S;
S.Value = 1;
Down(S);
<Si>.
```

(User Application)

```
Down(BSEM){
    if(S.value == 1){
        S.value = 0;
        return;
    }
    else{
        put this process(PB) in S.L and Block it by sleep();
    }
}
```

```
Up(BSEM){
    if(S.LL is not empty){
        Select a process from S.L and Wakeup();
    }
    else{
        S.value = 1;
    }
}
```

Cases of BSEM

I. S = 1;
P(S);
S = 0
Status = Success

II. S = 0;
P(S);
S = 0
Status = unsuccess <Blocked>

III. S = 0;
V(S);
S = 0: when Queue is not empty
S = 1: when queue is empty

IV. S = 1;
V(S);
S = 1
Status = successful.
[always when S = 1 Queue is empty]

I. S = 0;
V(S);
S = 1
Status = successful

papergrid

Date: / /

* Classical IPC problems with Semaphore based Implementation.

1. Producer - Consumer Problem

```
#define N = 100
int Buffer [N];
CSEM Empty = N;
CSEM Full = 0;
BSEM mutex = 1;
```

↓ Common semaphore shared by Consumer & Producer.

Producer Code

```
void Producer(void){
    int itemp, in = 0;
    while (1){
        itemp = produce.item();
        Down(Empty);
        Down(mutex);
        Buffer[in] = itemp;
        in = (in+1) % N;
        Up(mutex);
        Up(Full);
    }
}
```

→ if there are multiple positions, then it can lead to deadlock

Consumer Code

```
void Consumer(void){
    int itemc, out = 0;
    while(1){
        Down(Full);
        Down(mutex);
        itemc = Buffer[out];
        out = (out+1) % N;
        Up(mutex);
        Up(Empty);
        Process_item(itemc);
    }
}
```

Note: When we use multiple semaphores in the operation, then predominantly it is highly important that we have to maintain the semaphores order property. If the order is not maintained, potentially it may lead to deadlock

papergrid

Date: / /

2. Reader - Writer Problem

Here two processes Reader & Writer access a shared database <CS>
(Reader → read values/records; Writer → update/create new records)

If we have only one reader & one writer we only need one semaphore to solve the problem. But if we have multiple readers & writers we have 2 scenarios →

(a) First Reader - Writer

- Reading and Reading is not a problem, but (R-W, W-R and W-W) are problems
- So if we keep on entering readers, the writer process has to wait thus this will cause starvation to the writer process. (drawback)

Case 1: t_i : either r_1 or w_1 trying to access db, it will be allowed access.

Case 2: t_i : (r_1) & (w_1) contending for access (either r_1 or w_1 is allowed access)

Case 3: t_i : (r_1) ✓ (r_1 entered, w_1 waits)
t_{i+1} : (r_2) ✓ (r_1 and r_2 entered, w_1 waits)
t_{i+2} : (r_3) ✓ (r_1, r_2 and r_3 entered, w_1 waits) → starvation down

Case 4: t_i : (w_1) ✓ : r_1 waits
t_{i+1} : (w_2) ✗ : w_2 waits
t_{i+2} : (r_2) ✗ : r_2 waits

(at any point in time only one writer will be allowed)

papergrid

Date: / /

(b) First Writer - Reader (Second Reader-Writer)
(avoids Starvation)

Case 1: t_i : either r_1 or w_1

Case 2: t_i : (r_1) & (w_1) contending for access → (w_1) is allowed

Case 3: t_i : (r_1) ✓ enters t_{i+2} : (r_3) ✓ enters t_{i+4} : (r_4) ✗ waits since a writer is already waiting, thus avoids starvation to writer.
t_{i+1} : (r_2) ✓ enters t_{i+3} : (w_1) ✗ waits

This method by which we make the other readers wait, so that the DB becomes vacated is generally called Principle of Turnstile.

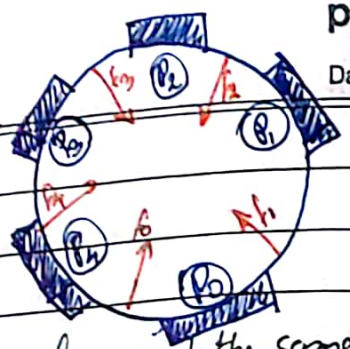
* Implementation of First-Reader-Writer Problem

```
int rc = 0; // readers count
BSEM mutex = 1; // used by Rs to regulate rc as CS
BSEM db = 1; // used by Rs and Ws and Ws and Ws to access DB as CS

void Writer() {
    while(1) {
        Down(db);
        <DB-Write>;
        Up(db);
    }
}

Void Reader(void) {
    While(1) {
        Down(mutex);
        rc = rc+1;
        if (rc==1) Down(db);   <- 1st reader is the leader of all other readers.
        Up(mutex);
        <DB-Read>
        Down(mutex);
        rc = rc-1;
        if (rc==0) Up(db);
        Up(mutex);
    }
}
```

When the 1st reader is allowed access to db, before entering CS it allows other readers access by performing Up on mutex.

papergrid
Date: / /

* Dining Philosophers Problem



Philosophers need both forks to eat when they get hungry. They take left fork first & then the right fork (or vice versa)
Here if all the philosophers become hungry at the same time and tries to take their forks we get a deadlock situation.
To prevent this deadlock we have two solutions:

① Non-Semaphore based soln ② Semaphore based soln

Even no. of philosophers | (N-1) philosophers L-R
L-R and odd no. of philosophers R-L | and one philosopher R-L

Maximum Concurrency: Max no. of philosophers who can eat without deadlock $= \lfloor \frac{n}{2} \rfloor$ //

* Semaphore Based Solutions

```
#define N 5
#define Left (i+N-1)%N
#define Right (i+1)%N
#define THINK 0
#define HUNGRY 1
#define EATING 2
BSEM mutex = 1;
BSEM S[N] = {0};
BSEM State[N] = {THINK};
```

initial states.

```
void Philosopher (int i){
while (1) {
  (a) Think (i);
  (b) Take-forks (i);
  (c) eat (i);
  (d) put-forks (i);
}
}
```


papergrid
Date: / /

```
(b) void Take-forks (int i){
Down (mutex);
State [i] = HUNGRY;
test (i);
Up (mutex);
Down (S[i]);
}
```

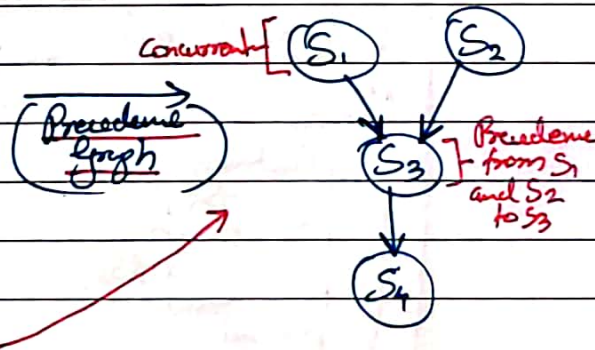
```
void test (int i) {
if (State [i] == HUNGRY &&
    State [LEFT] != EATING &&
    State [RIGHT] != EATING ){
State [i] = EATING;
Up (S[i]);
}
```

```
(d) void Put-forks (int i){
Down (mutex);
State [i] = THINK;
test (LEFT);
test (RIGHT);
Up (mutex);
}
```

* Concurrency Mechanisms (to achieve Concurrency in programming)

(eg)

```
{
S1 : a = b+c;
S2 : d = e*f;
S3 : k = a+d;
S4 : l = k*m;
}
```



Here S1 and S2 can be concurrently executed because there are no dependencies b/w them. But S3 depends on S1 & S2, & S4 depends on S3

A2: Statement(S) → Read…
→ Write Set: W(S) = a

Eg: a = b+c

Now Consider • S_i → $R(S_i)$, $W(S_i)$ • S_j → $R(S_j)$, $W(S_j)$

I. $R(S_i) \cap W(S_j) = \phi$
II. $R(S_j) \cap W(S_i) = \phi$
III. $W(S_i) \cap W(S_j) = \phi$

S_i & S_j are concurrent only if they 3 satisfy (Bernstein's Concurrency Conditions)

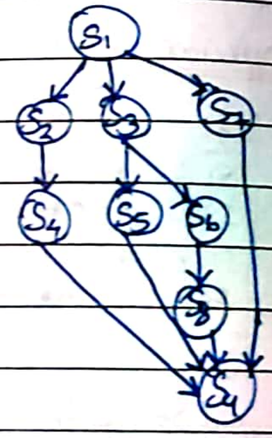
$R(S_i) \cap R(S_j) = \phi$ / Not ϕ

Types of Concurrency →
- Real/Physical Concurrency (need 2 CPUs/processors) (multiprocessor/multicore architecture needed)
- Pseudo Concurrency (Interleaved/Preemptive) (Execution to achieve Pseudoconcurrency)

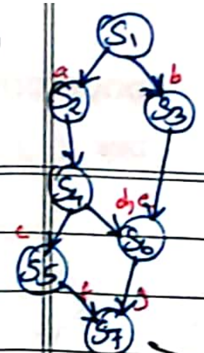
Concurrency Mechanism – (Parbegin-Parend / Co-begin – Coend)

Eg 1

```
S1;
Parbegin
  begin S2; S4; end
  begin
    S3;
    Parbegin
      S5;
      begin
        S6; S8;
      end
    Parend
  end
Parend
S7;
S9;
```



Eg 2



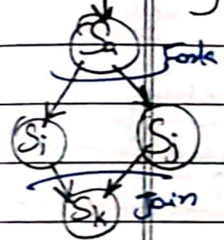
This cannot be represented by using parbegin & parend alone.
We need to use Semaphores here.

Thus Semaphore is a General Purpose Utility.
It's not just used for critical section problems, but also concurrency problems.

initially → BSEM a, b, c, d, e, f, g = {0}.

```
Parbegin
  begin S1; V(a); V(b); end
  begin P(a); S2, S4; V(c); V(d); end
  begin P(b); S3; V(e); end
  begin P(c); S5; V(f); end
  begin P(d); P(e); S6; V(g); end
  begin P(f); P(g); S7; end
Parend.
```

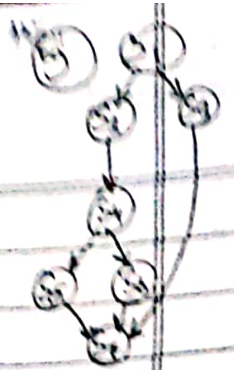
Concurrency Mechanism – (Fork and Join)



fork L: Starts computations concurrently at (i) after fork (ii) @ Label L.

```
count = 2;   → no. of branches to join
Sa;
fork L;
  Si
  goto M;
  ⋮
L: Sj;
M: Join(count);
Sk;
```

```
Join(int count) {
  Count = Count - 1
  if (count != 0)
    exit;
  else
    return
}
```

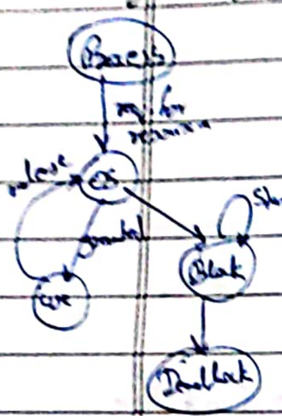


→ count=3;
S1;
fork L1;
S2;
S4;
fork M1;
S5;
goto Z;
L1: S3;
goto Z;
M1: S6;
Z: Join(count);
S7;

* Deadlock (Multiple processes waiting for an event that will never happen)

(most undesirable feature of OS)

During deadlock it is not just that processes are blocked but also resources will become unavailable for other processes. Thus efficiency & throughput levels drop drastically.



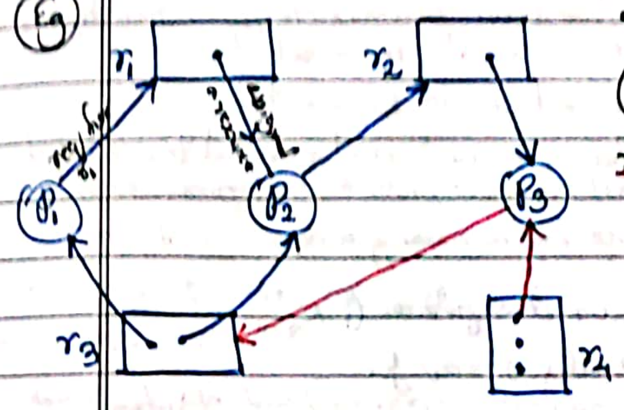
* Necessary Conditions for a deadlock to happen (if deadlock → all these cond^n true; but if all these cond^n true → need not be deadlock)

ⓐ Mutual Exclusion
ⓑ Hold and Wait (Hold a resource & wait for another)
ⓒ No Preemption
ⓓ Circular Wait (processes waiting for each other to finish forms a cycle)

(If any of these 4 cond^ns does not exist, deadlock can't happen)

* Resource Allocation Graph (RAG)

(Eg)



- here r1 and r2 are single instance resources (Eg CPU), r3 and r4 are multiple instance resources (Eg registers)

Here (without red edges) → P1 and P2 are blocked but P3 is runnable. So after P3 runs, P1 & P2 also will become runnable.

(With red edges) → P1, P2 and P3 are blocked forever.
P1–r1–P2–r2–P3–r3–P1 forms a cycle.

Note: In a graph that has only single instance resources, cycle becomes a necessary & sufficient cond^n. i.e., if a cycle is present, it means deadlock also present.

* Deadlock Handling Strategies

(i) Prevention
(ii) Avoidance (Bankers Algorithm) (costly deadlock handling strategy; rarely used)
(iii) Detection & Recovery (they are used in critical computing, where we can't afford to lose our data or time)
(iv) Ignorance (Ostrich Algorithm) Essentially no strategy. When deadlock occurs, just restart, next time deadlock may be avoided. Used in contemporary desktop OS like Windows, Linux, Unix, etc.

 Date: / /

* Deadlock Prevention (By dissatisfying one or more necessary cond^n for deadlock)

ⓐ Mutual Exclusion – It's noncompromisable for a multiprogram OS. ∴ it can't be dissatisfied.

ⓑ Hold and Wait – Implement hold or wait.

2 approaches → ⓐ Process requests and must be allocated all resources prior to its commencement.

ⓑ Release all the resources before making a new request.

Eg) a process P runs in two phases (P) → Phase I : $\langle r_1, r_2 \rangle$: 30min; Phase II : $\langle r_2, r_3 \rangle$: 20min

approach ⓐ → Initially P req. for all 3 resources (r_1, r_2, r_3) and only when it gets all these 3 together it starts executing.

Drawbacks – (i) Starvation (Wait can be long time)
(ii) Resource Inefficiency – Here r_3 sits idle for 30min & r_1 sits idle for 20min.

approach ⓑ → After phase I completion both r_1 and r_2 are released. Now it req. for r_2 and r_3 again, when both are available it starts executing.

Drawback – Only Starvation (Resources are utilized efficiently here)

ⓒ No Preemption of Resources → Self Preemption (req. for a resource & it is not free, then the process preempts all its other resources)
↓
Forceful Preemption (Take resource forcefully from another process)

 Date: / /

ⓓ Circular Wait → Assign a unique-id to each resource.

→ Never allow a process to req. a lower numbered resource than the last one allocated.

Eg)

Resources	R-id
A	10
B	8
C	5
D	12
E	9
F	8
G	20

time = t → Process → C (5), E (9), A (10)   5<9<10

time = t+1 → Process → C (5), E (9), A (10), G (20)   5<9<10<20

Now Process req. for B (8), but B comes between C and E (So all conflicting resources are released E, A, G) (C is still held, now req. for B, then E, then A and then G.

So we get Process → C (5), B (8), E (9), A (10), G (20)   5<8<9<10<20 //

if B comes after C and E cycle will be formed. C E A G B – 5 9 starvation

Drawback – Starvation

* Deadlock Avoidance (Bankers Algorithm)

Bankers Algorithm = Safety Algorithm + Resource Request Algorithm

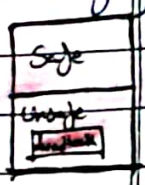
$TC = O(n^2 \times m)$ [n = no. of process, m = resources types]

papergrid

Date: / /

→ Safety Algorithm (used to test whether system is in safe state or not)

Basic objective of Banker Algo. is to provide always system in the safe state.



Parameters of the Algorithm

1. n = no. of process 2. m = no. of Resources

3. Maximum [1…n, 1…m] = M[i,j] = k means Process 'i' requires maximum of 'k' copies of Resource 'j'

This is an optimistic approach as Banker algo. works on the basis of optimistic

4. Allocation [1…n, 1…m] = Alloc [i,j] = a

$P_i \xleftarrow[alloc]{} a(R_j)$

5. Need [1…n, 1…m] = Need [i,j] = b = $P_i \xleftarrow[need]{} b(R_j)$

Need = Max − Alloc

6. Request [1…n, 1…m] = Req [i,j] = b = $P_i \xleftarrow[req]{} b(R_j)$

Req ≤ Need

7. Total [1…m] = Total [j] = z (Total z copies of Resource 'j')

8. Available [1…m] = Avail [j] = e (e copies of R_j is free/available)

Avail ≤ Total

papergrid

Date: / /

n = 5, m = 3, Total <ABC> : <10; 5; 7>

Eg

to	Max A B C	Alloc A B C	Need A B C	Avail A B C	
P_0	7 5 3	0 1 0	7 4 3	3 3 2 P_1	Now the available resources are <ABC> = <10 5 7> − <7 2 5> = <3 3 2>
P_1	3 2 2	2 0 0	1 2 2	5 3 2 P_3	
P_2	9 0 2	3 0 2	6 0 0	7 4 3 P_4	
P_3	2 2 2	2 1 1	0 1 1	7 5 5 P_0	
P_4	4 3 3	0 0 2	4 3 1	10 5 7 P_2 ↓	
		7 2 5			

Now with <3 3 2> we can satisfy P_1, so now we have.

$332 + 200 = 532$, with 532 satisfy P_3, so now we have.

$532 + 211 = 743$, satisfy $P_4 \rightarrow 743 + 002 = 745$, satisfy P_0

weight 745 + 010 = 755 and finally satisfy P_2 weight 10,5,7 yes.

↓

Thus the System is Safe → Safety Algo. says, the system is safe iff the need of all process can be satisfied with the available resources in some order.

→ Resource-Request Algorithm – Now from the RQ, any process can be scheduled to run on CPU next. So the Res-Req algo. acts like a banker. The request by the next process from RQ is granted iff the resulting state after satisfying the req. is also safe. If the resulting state is unsafe, the req is denied and the next process req. is resumed for analysis.

Thus overall we get the Banker's Algorithm

Algo Res-Req (Pi, Avail, Alloc, Reqi, Need) {

1. Reqi ≤ Needi
2. Reqi ≤ Availi
3. Assume to have satisfied Reqi
   - (a) Availi = Availi - Reqi
   - (b) Needi = Needi - Reqi
   - (c) Alloci = Alloci + Reqi
4. Run Safety Algo.
5. If system is safe then Grant else Deny

}

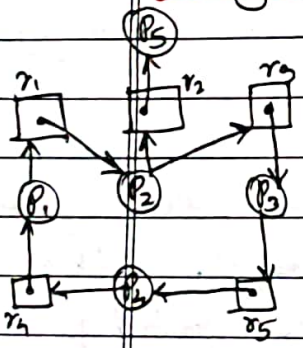
Thus Banker's Algo. always gives the system in Safe State

Note: Unsafe state may/may not have deadlock
Safe state will never have deadlock

* Deadlock Detection & Recovery

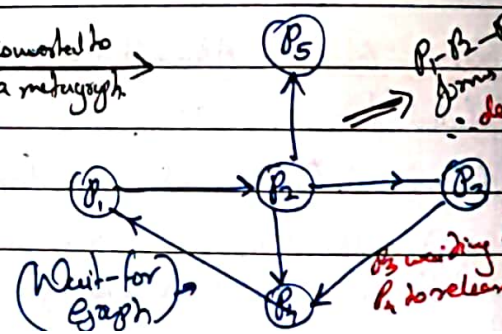
(i) Single Instance of Resource Type

(Fig with 5 a)



(Resource Allocation Graph - (RAG))

converted to a wait-for graph



(Wait-for Graph)

P1-P2-P3-P1 forms a cycle ∴ deadlock

P3 waiting for P4 to release r5

Deadlock detection is activated due to multiple reasons like, the utilization of the processor, if the job is delivering majority of process get blocked

We run cycle detection algo. on this Wait-for-Graph and if any cycles are detected then deadlock is confirmed. (Pg 45 @)

Thus P1, P2, P3 and P4 are infected processes or deadlock processes.

The deadlock processes are the output of detection algo. & the input of recovery algo.

(ii) Detection in Multi Instance Resource Types (General Detection Algorithm)

(Eg 1)

	Alloc (ABC)	Req (ABC)	Avail (ABC)
P0	0 1 0	0 0 0	To do ⟨0, 1, 0⟩
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	
	7 2 6		

Here as we can see at time t0 P1, P3 and P4 are blocked. So majority of the process are blocked. So it is a symptom for deadlock.

In Banker's Algorithm, the output is either safe or unsafe, but in detection algorithm, output is either safe or deadlock.

Here the system is said to be safe if the seq. of all Processes can be satisfied with available resources, otherwise deadlock.

Given eg is safe, but if we change P2 req to ⟨0 0 1⟩ then we get a deadlock state.

papergrid

Date: / /

Deadlock Recovery Strategies

(a) Process Termination

(i) Kill All - Drawback → Starvation → Restart all killed process and loose progress.

Advantage → Once all killed we can be sure that there will be no deadlock at the point so, no need to rerun the deadlock detection.

(ii) Kill One at a Time - Consider a situation P_1 completed 95%, P_2 - 80%, P_3 - 30%, P_4 - 5%. Killing P_4 won't loose much progress. In Kill all we loose all progress. So here first kill P_4, rerun detection algo. if deadlock present, then kill P_3 ... etc. until we overcome the deadlock → Can suffer from Starvation

(b) Resource Preemption - Preempt the least resource held by a process & add them to the pool of available resources. Now deadlock detection is rerun ..... etc. The processes are not completely killed they can restart from the point of their preemption.

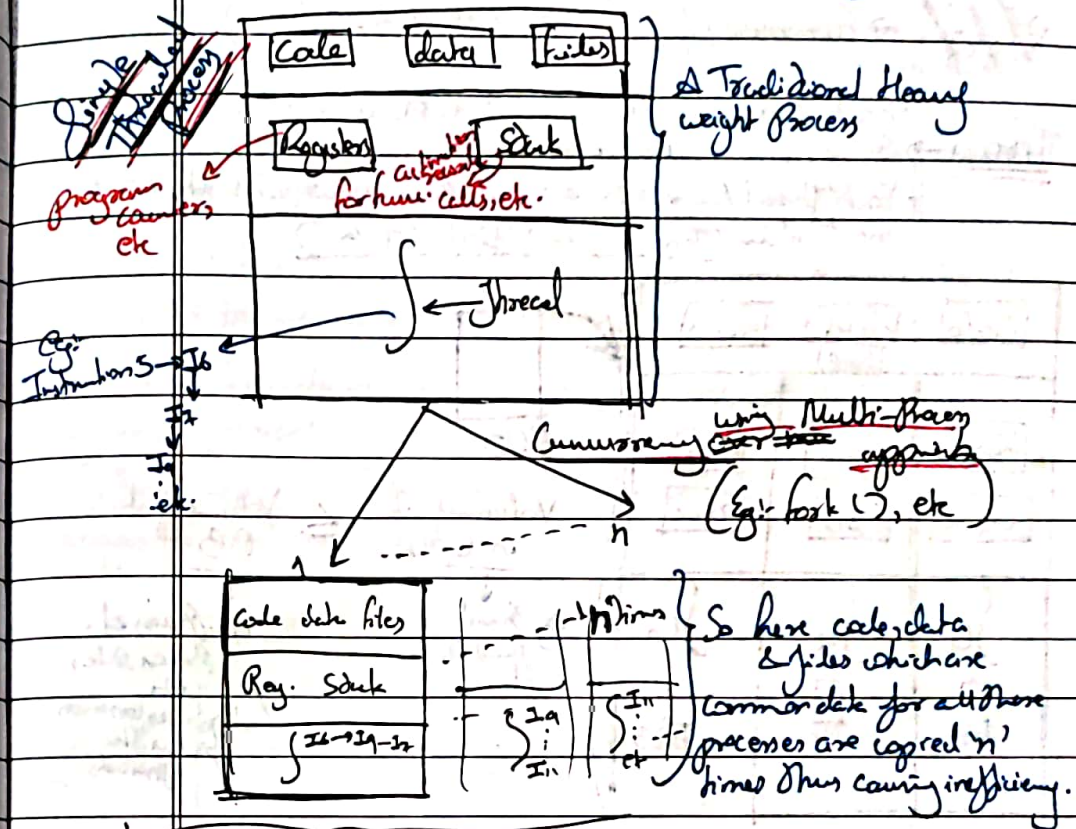
Can suffer from Starvation.

During preemption of a resource, the process has to roll back its execution to its previous point. So rolling back is an additional overhead.

papergrid

Date:

* Threads (Thread can be defined as a light weight process)



These 'n' process are scheduled using Round-Robin or other CPU Scheduling techniques & pseudo concurrency is achieved. When load is less this may be good. But for high loads this can cause a bottleneck.

Multi-threading advantages:
→ Resource Sharing (using/among the same address space for code, data & files)
→ Economical (cost effective).

Thread →
- Each Thread is also an unit of CPU utilization
- It is also a schedulable unit.
- Each Thread being a process will have its own attributes which are stored in Thread Control Blocks (TCBs)

Volume of info in TCB ≤ Volume of info in PCB

TCB ↓ eg:- thread id, thread states, etc.

PCB ↓ eg:- Process id, Process state, etc. (attributes common for all the threads)

→ Threads have less no. of context switching.
→ Threads have faster context switching.
} ∴ better performance.

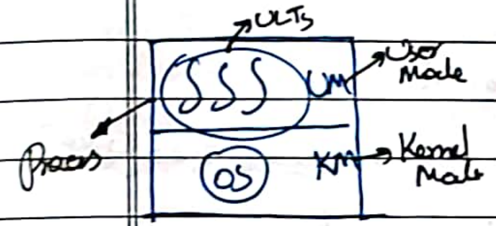
→ Threads could also utilize multicore processor architectures.

[Note:- A monitor schedules the execution of process in a multiprocessor OS]

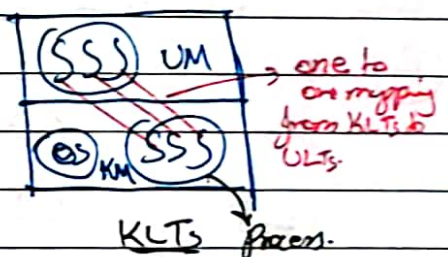
Types of Threads → User Level Threads (ULTs)
→ Kernel Level Threads (KLTs)

Benefits of ULTs – 1. Flexibility (OS will only see the process, threads will be entirely managed by users)
2. Transparency
3. User Level Thread Switching is Superfast because OS is not involved.
∵ No of mode shifting from user to kernel and back downer.

Drawback : Since OS can only see the entire process, if one of the threads of ULTs request for some IO operation, the entire process will be blocked.
↓
To overcome this problem we go for Kernel level Threads and compromise flexibility, Transparency, Userlevel thread switching, .
Here the OS itself will have separate Kernel Level Threads that manages the corresponding ULTs.



ULTs



KLTs

[ULTs Eg :- Java Threads, P threads, etc]

- Due to variety of benefits from threading, there was shift from Multi-processing to Multi-threading.