Introduction: 1 to 10
Process Managment: 11 to 19
CPU Scheduling: 20 to 40
Process Syn: 41 to 76
Deadlock: 77 to 87
Memory Management: 88 to 132
Disk Scheduling: 133 to 154

Date 11/01/23

Introduction

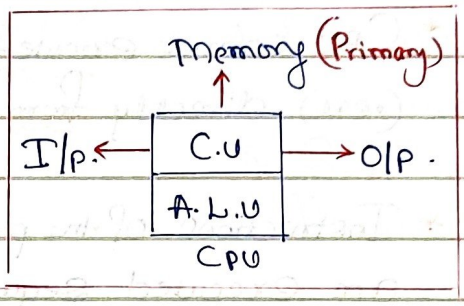
What is an Operating System?

* Is an interface between user and the hardware
* Resource managers
* Set of utilities to simplify the application development
* Set of program(s) / Software
* Operating System acts like a government.

User/Programmers ⟷ OS ⟷ Hardware.

1. **Control unit (C.U)**: It is circuitry that directs operations within a computer's processor.
   * Timing / control signals
   * Sequencing / execution of micro-operations.

   (operations that are carried out on the data, stored in registers (CPU))



VON-NEUMANN Architecture.

2. **Arithematic Logic Unit (ALU)**: It is a part of CPU that carries out arithematic and logic operations on the Operands in computer instruction words

Memory:

Primary (also called "Main memory") — eg: Ram, Rom, Cache, registers.

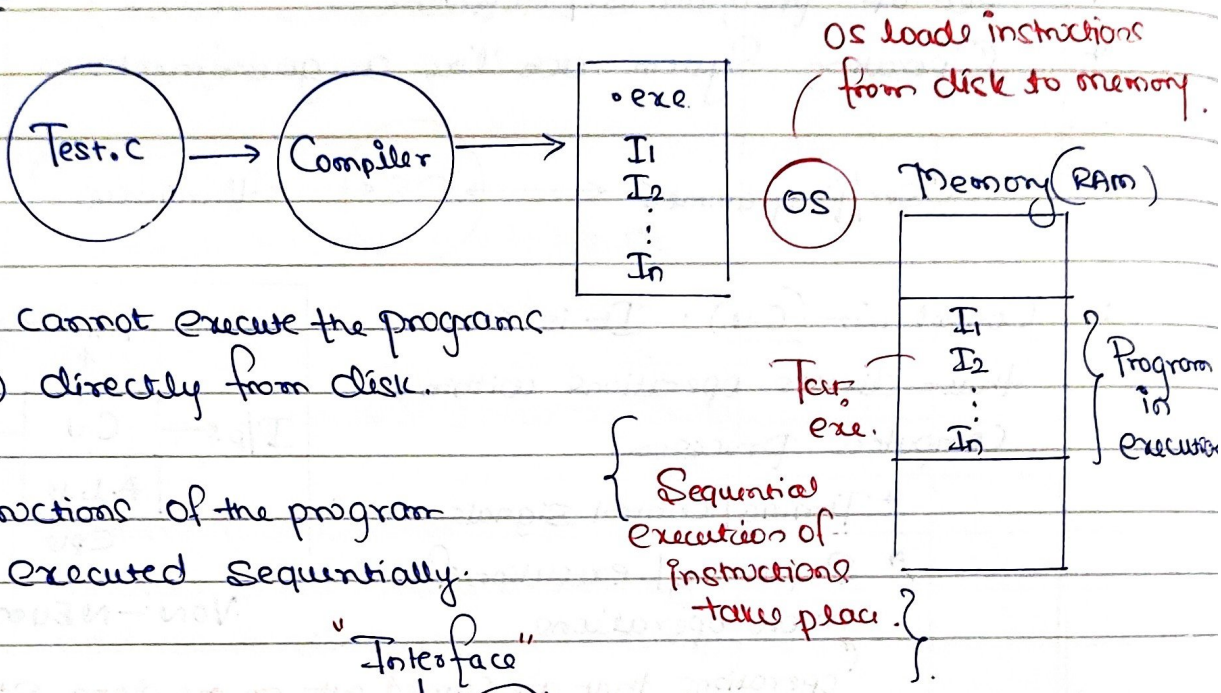
Secondary ("Auxilary memory") — eg: hard disks, Pendrive etc.

Primary memory	Secondary memory
* Volatile (not Rom)	* Non volatile
* faster to access $\times 10^{-9}$ (ns)	* Slower to access (ms) $\times 10^{-3}$.
* Smaller Size	* larger Size.
* more expensive	* less expensive

Note: as per Von Neumann architecture, all Secondary storage devices are part of Ip/op devices.

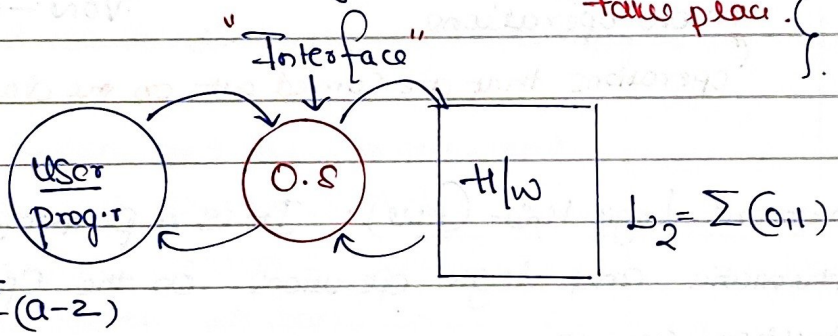
Q How are programs executed in Von Neumann architecture?

→ The programs are executed by the CPU by Stored program concept



* CPU cannot execute the programs (.exe) directly from disk.

* Instructions of the program are executed sequentially.

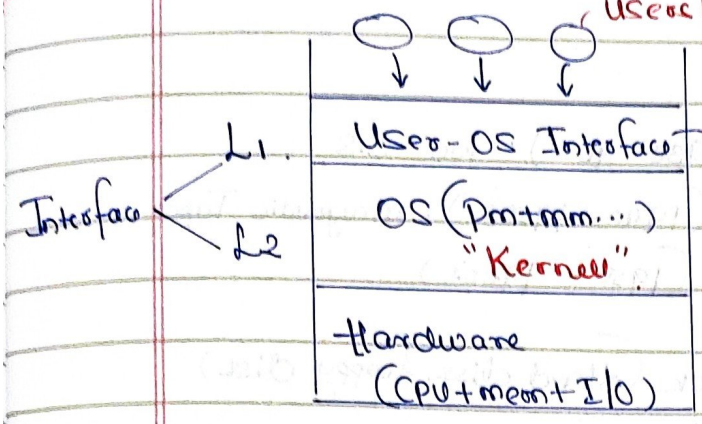


Modules of OS:

- Process manager (CPU)
- Memory manager (memory)
- File manager } Devices
- Devices manager
- Protection manager (Resources)

Collection is called "Kernel / Nucleus"

Kernel is a central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time.

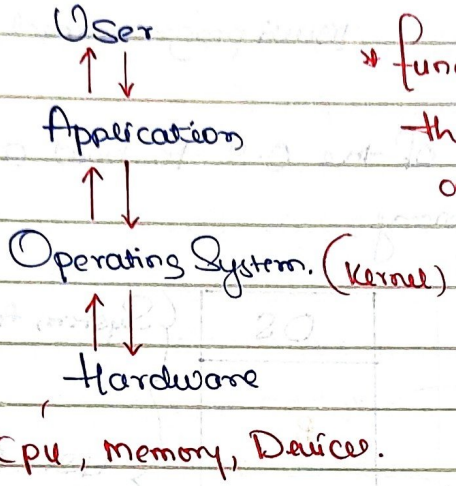


Functions of OS:

1. Process management
2. Memory mgmt.
3. Error detection
4. Security
5. File Management.

Functions and Goals:

1. Convinience (ease of use)
2. Efficiency
3. Reliability
4. Robustness
5. Scalability
6. Portability



* function is something that is carried out to achieve the goals.

* Primary (main) goal of an Operating System: Convinience (for personal computing)

* Convinience is not primary goal of all other computing environment

Other Computing environment:

(a) Real time Systems (RTS): * Operate in real time (fixed deadlines)

↓

main goal: efficiency

* Ex: missile, Satellite, nuclear System control, control

← Hard - Real time System }

(b) Mobile Operating System main goal: battery efficiency.

Tim Types of OS:

1. Batch OS
2. Multiprogrammed OS
3. Time Sharing OS
4. Real time OS
5. Network OS
6. Distributed OS

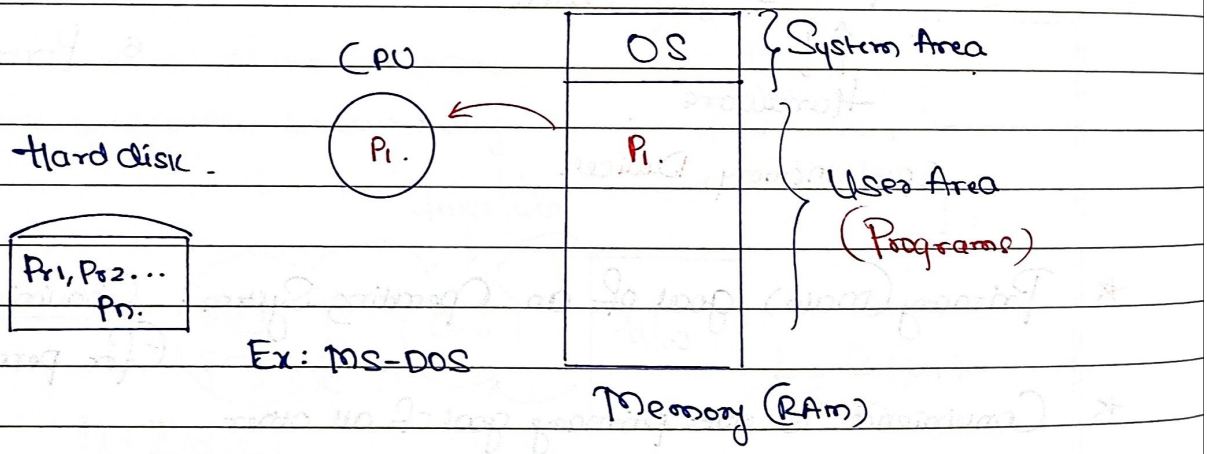
} Advanced

Generations of Computers:

1st Generation of Computers (1930-40s): No O.S
2nd Generation of Computers (1940-1950's): magnetic Tapes
3rd Generation of Computers (1950 - 1960's)
↓
Magnetic Disk (Hard disk, floppy disk)

↙ Uni-programmed ↘ Multiprogrammed

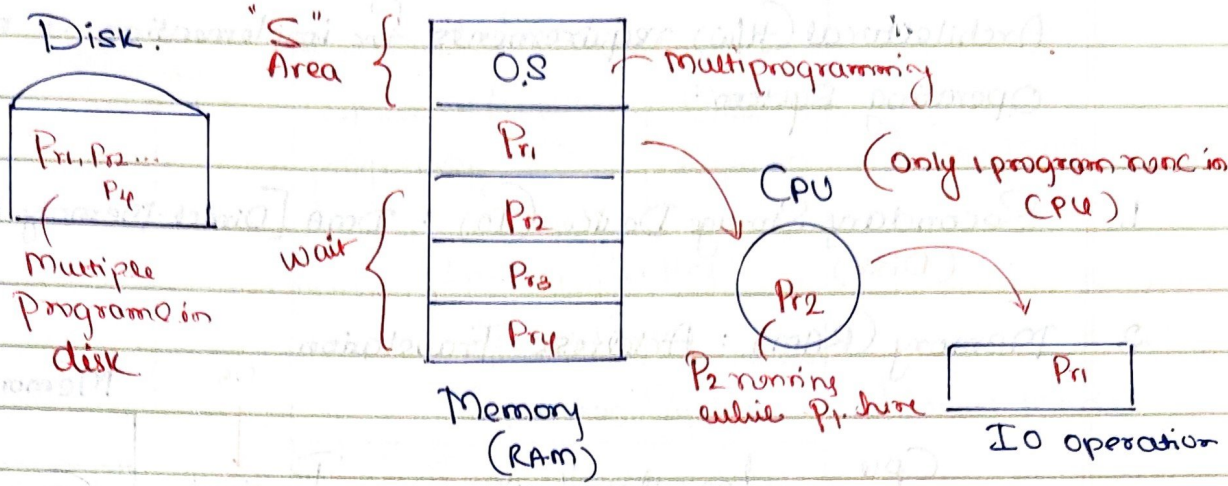
Uni-programming: Ability of the O.S to hold a single program in memory



Drawback:
* Single program in memory is not enough (sufficient) to keep the CPU always busy.
* Throughput and efficiency will be less.
↑ no of programs completed / Unit time.

Multiprogramming: Ability of the operating system to hold (manage) multiple ready to run programs in memory.

Objective: * maximize CPU utilization
* maximize throughput.

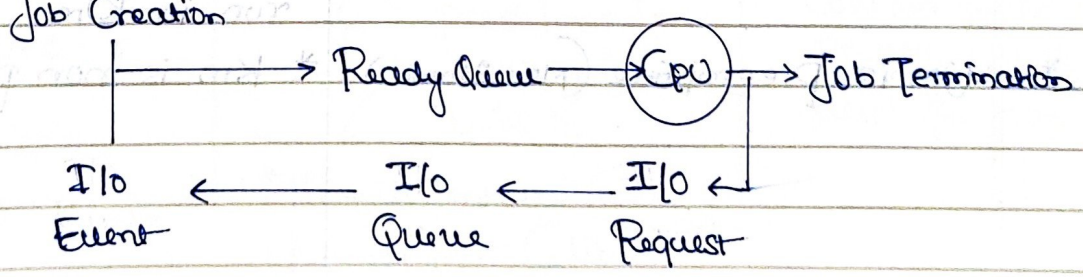


Multiprogramming: Multiplexing of CPU among different programs in memory.
(another definition)

Types of multiprogramming:

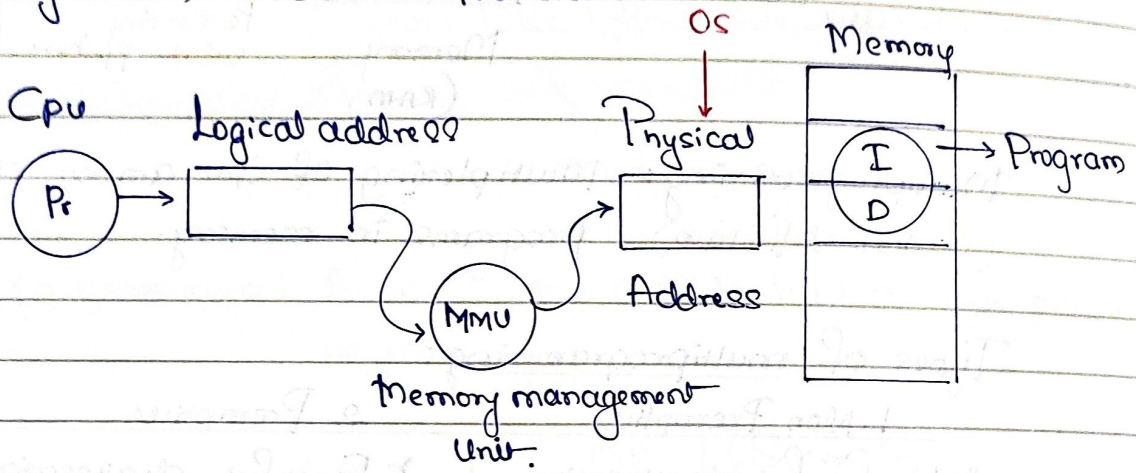
1. Non Premptive	2. Premptive
* No forceful deallocation * Running program on CPU will not be forced to leave CPU. It will release CPU voluntarily: 1. Completed execution 2. needs I/O syst. call. * Drawback: leads to starvation to other waiting programs. - lack of interactiveness, responsiveness.	* Forceful deallocation. * Program running on CPU, can get forcefully deallocated from CPU (Pre Emption), So that waiting programs can get their chance to run on CPU Objective: * improve interactiveness * increase responsiveness Premption is done based on: 1. Priority 2. Time

Multiprogramming (Schematic View)



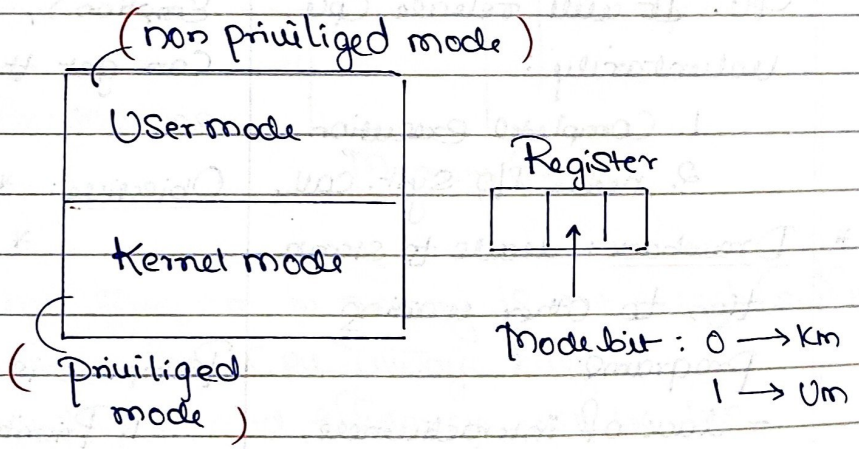
Architectural (H/W) requirements for implementing a multi programmed Operating System:

1. Secondary Storage Device (IO) : DMA [Direct Memory Access]
   (Disk)
2. Memory (RAM) : Address Translation

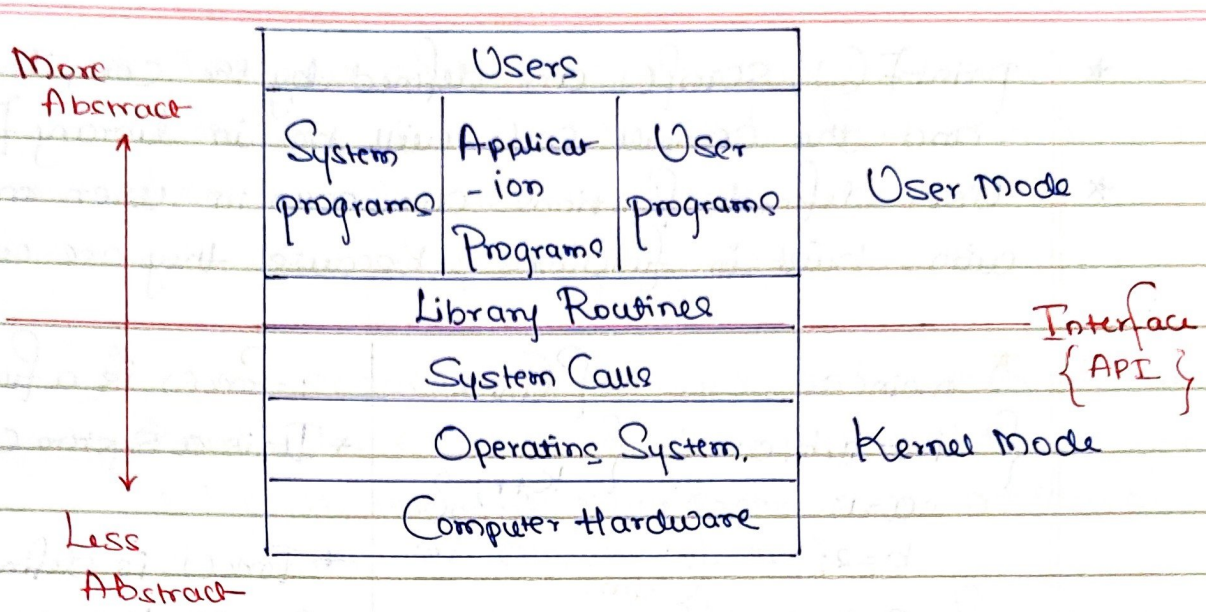


3. CPU (Processor) : Dual mode Operation

   Every CPU of multiprogrammed OS should have two mode of execution.



User mode	Kernel mode
* All user applications run in Um.	* All OS routines programs run in K.m.
* Um is preemptive (Non Atomic)	* Km is non premptive (Atomic)



* Many a times it becomes necessary to shift the mode from user mode and kernel mode and back from kernel mode to user mode.

Mode Shifting: Shifting of mode from user mode to kernel mode and back is called mode shifting.
Mode Shifting (Process) is needed to avail OS services.

API : Application Programmer's Interface / System Call Interface

Implementation of mode shifting process via API/SCI.

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

```
f(int k)
{ k++;
  printf("%d",k);
}
```

predefined fn. - library fun. - built in.

* Implimentation is in library file (.lib)

```
f();
```

'user defined function

* <Stdio.h> file contains function prototypes, which is used for type checking

* printf(), scanf() are defined by the compiler implementor and the compile code will be in library file.
* User defined functions are run in user mode along with built in functions because they are written by users.

```
main()                    f(int k)
{ int a,b,c;              { k++; }
  a=1;
  b=2;                    } Um.
  c=a+b;
  f();
  fork(); ← Km.
  printf("%d",c);         } Um.
}
```

fork() is a function,
* It is a System call } OS routine
* Fork() is defined / implemented in OS Kernel
* Execution of a fork() results in Creating a child process.

```
main()
{ :
  :
  f(); → BSA
  :
  fork(); → SVC
  :
}
```

* CT: Compile time
* BSA: branch and save address } Instructions
* SVC: Supervisor Call (privilged inst.)

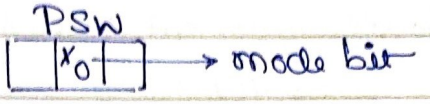
{ All pre defined functions are translated into instructions called BSA (non privilged instruction) — User mode at runtime. }

{ Software interrupt instructn }.

* fork() is converted into SVC because it is a not user defined — S/w interrupt at runtime.

* Interrupt generated by software instructions are called Software interrupt.
* ISR: Interrupt Service routine: A program of OS to Service the interrupt.

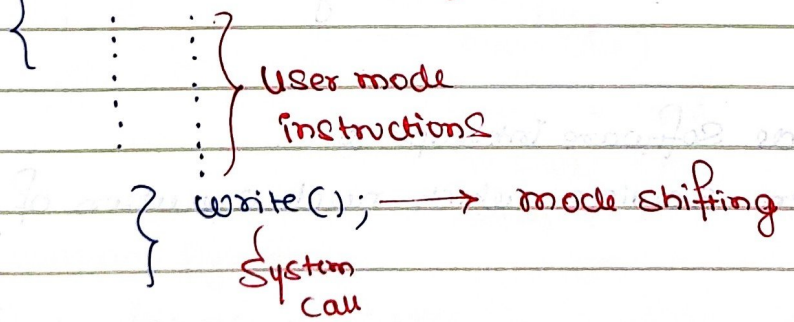
ISR: * ISR will go the register and convert the mode bit from '1' to '0' to shift from user mode to kernel mode and vice versa.



* ISR examines an interrupt and determines how to handle it, executes the handling and then returns a logical interrupt value (returns the address of fork from the dispatch table).

* All the Operating Systems - Windows, Linux, Unix follow the same paradine
* The fork() will be executed at kernel mode in the dispatch table. All instructions will be executed of the fork and the last instruction will change the mode bit from '0' to '1' after execution and then fork is executed sucessfully at kernel mode and returned to user mode and then remaining code will run.
* To change the mode from kernel mode to user mode then no Software interrupt is required, interrupt is required for changing modes from Um to Km only.

printf(): Library file can also use System-call.

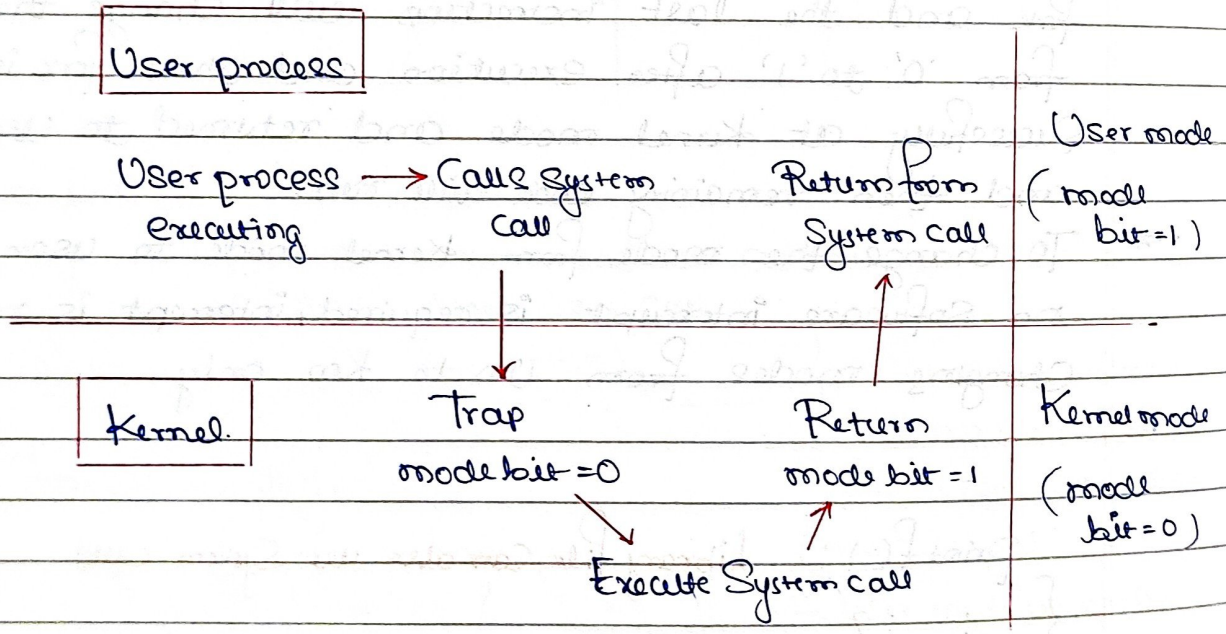


Note: every Priviliged instruction does not generate interrupt

1. Kernel mode: In kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference

any memory address. Kernel mode is generally reserved for lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic, they will halt the entire PC.

2. User mode: In user mode the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.



Q1. A processor needs software interrupt to
: Obtain system services which need execution of priviliged instructions.

Q2 A CPU has 2 modes priviliged & non priviliged. In order to change the mode from priviliged to non priviliged:
: A priviliged instruction (which does not generate an interrupt) is needed.

Process Management.

Q3. System calls are usually invoked by using.
: Software interrupt

Q4. A computer handles several interrupt sources of which of the following are relevant for this question:
* Interrupt from CPU temperature sensor (raises interrupt if CPU temperature is too high)
* Interrupt from mouse (raised interrupt is if is moved / button is pressed).
* Interrupt from keyboard (" " key is pressed / released)
* " " Harddisk (" " disk read is completed).

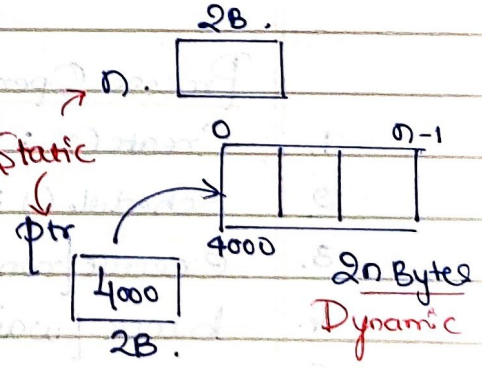
Which one of these will be handled at the highest priority?
: Interrupt from CPU temperature

Program vs Process

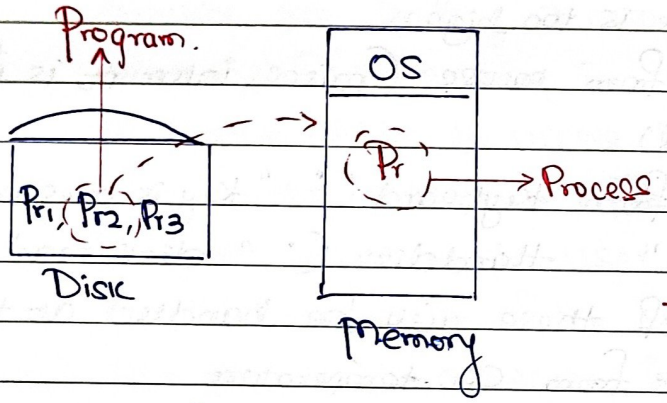
- Program (.exe)
  - Instructions
    - Ex: Load, Store, Add, mul
  - Data (operands)
    - Static
      - (Fixed / known size)
      - * memory allocated at load time (before RT).
    - Dynamic
      - * Varying size / fixed size
      - * memory allocated at run time.
- Process: Program in execution

Static and Dynamic

Ex: `int n; int *ptr;`
`scanf("%d", &n);`
`ptr = (int*) malloc(sizeof(int)*n)`



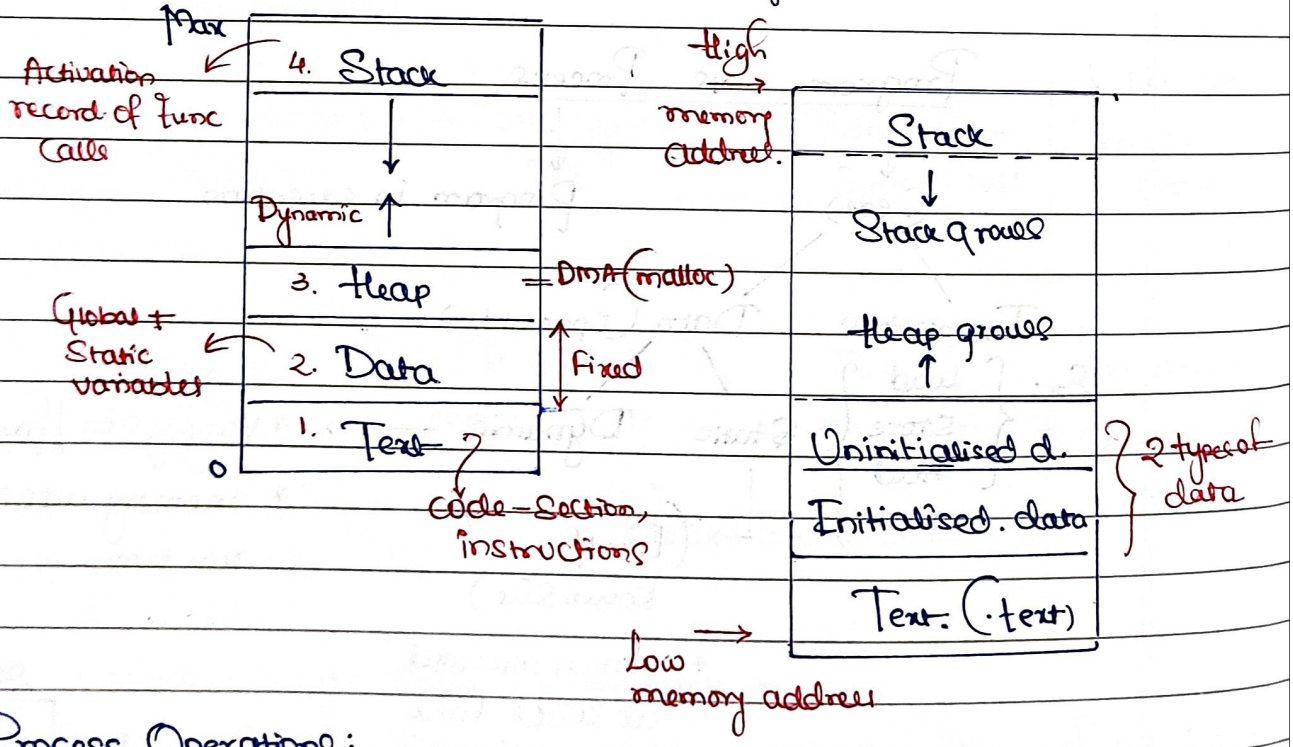
Process:
* Program in Execution
* When program is loaded from Disk to main memory then program is converted to process.
* Instance of a program.
* Locus of control
* Animated Spirit.



* Program is passive.
* Program is always on disk.

* Process is an abstract data type.

Representation of process in memory.



Process Operations:

1. Create() : resource allocation
2. Schedule() : the act of selecting process to run on CPU
3. execute/run() : the act of executing instruction from code section
4. block/wait : for I/O operation / System call
5. Suspend : act of moving process from memory to disk.

6. Resume : act of moving process from disk to memory
7. Terminate : act of resource deallocation.

Process as an entity is associated with several attributes :

1. Identification : <Pid, gid, Ppid...>
2. CPU related : <Program counter, priority, state, burst time...>
3. Memory related : <Size, limits, ...>
4. File related : <list of files, ...>
5. Accounting : <resources ....>

And many more attributes but these are some important ones.

PCB <Process Control Block> :

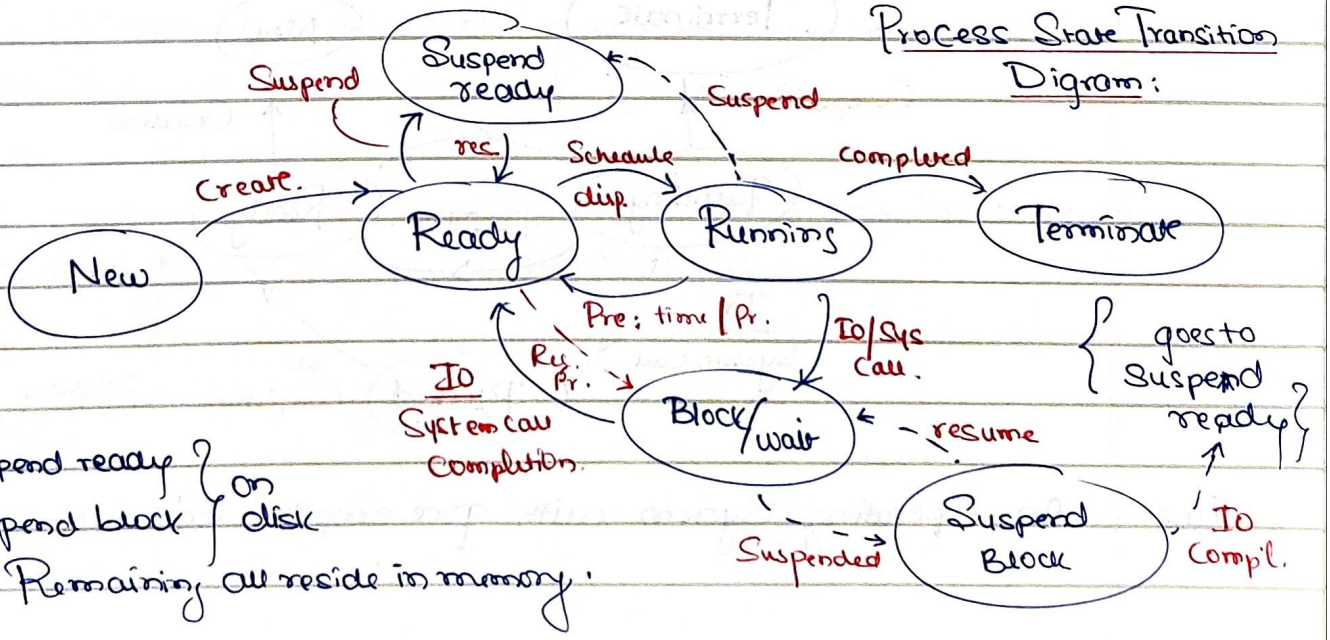
* Contains all attributes of process.
* each process has its own PCB.
* PCB is Stored in memory.

★ Process Context : content of PCB

Process States.

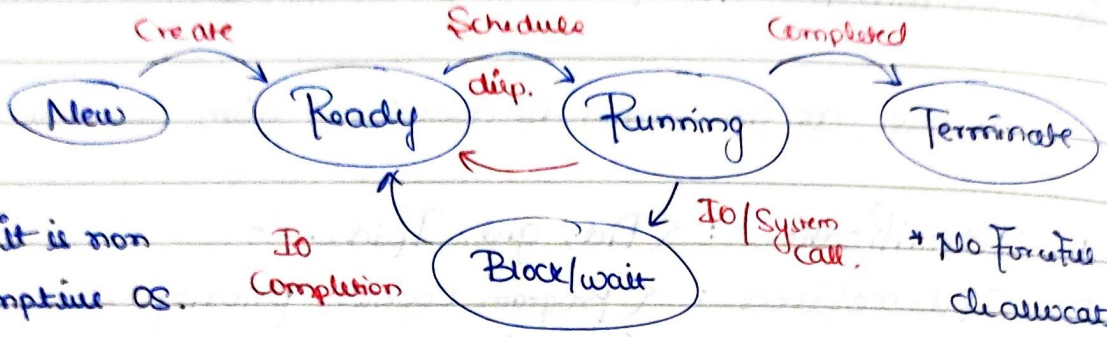
1. new : a process is created and resource is allocated
2. ready : ready to run on CPU
3. running : executing instructions on CPU
4. block/wait : process needs to perform IO/ system call.
5. terminate : resource deallocation.

Process State Transition Digram :



Suspend ready } on disk
Suspend block }
Remaining all reside in memory.

Process State transition diagram of uni programmed O.S:



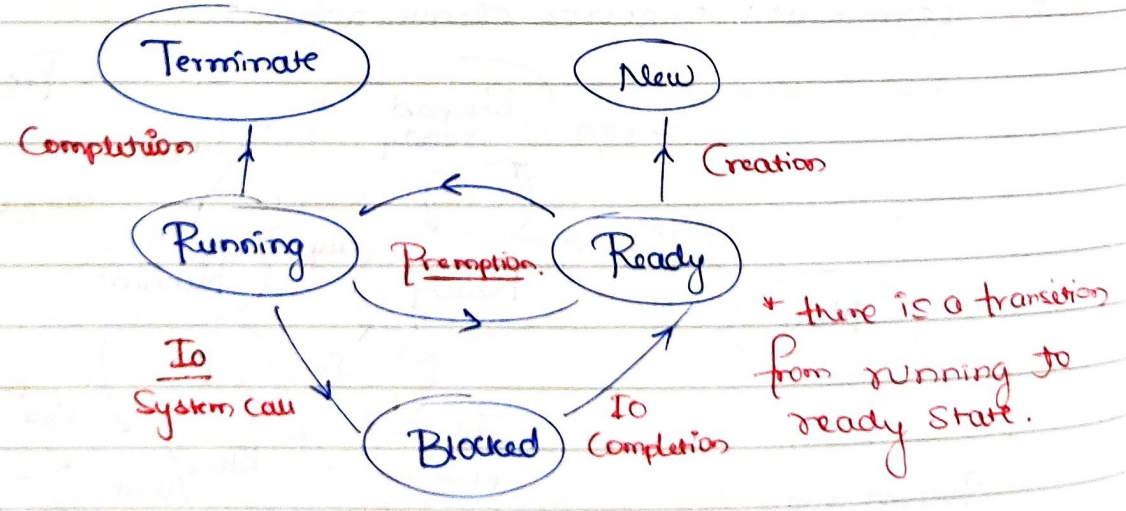
Note: it is non preemptive OS.

* just adding a transition from running to ready state, then the System becomes pre-emptive.

Notes: 1. As long as the process is in ready + running + block states, it is in main memory.

2. There can be many ready, block processes.
3. Maximum number of running process depends on no. of CPUs.
4. The process may go from ready state to block/wait state because of any missing resource.

Q1. The process state transition diagram given below is representation of: Ans: An Operating System with preemptive Scheduler.



Ans An Operating System with pre emptive Scheduler.

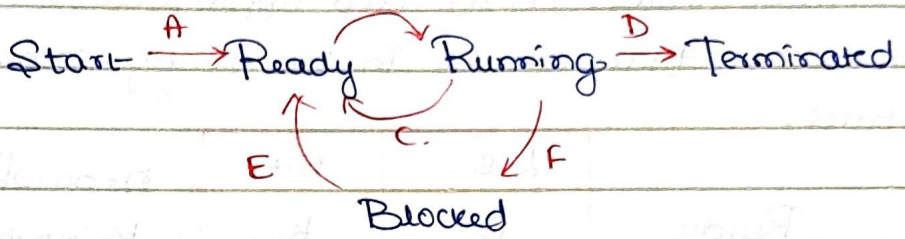
Q2. In the following process state transition diagram for a uniprocessor system, assume that there are always some processes in ready state:

(i) If process makes a transition D, it would result in another process making transition A immediately.

(ii) A process P2 is blocked state can make transition E while another process P1 is in running state.

(iii) The OS uses preemptive scheduling

(iv) The OS uses non preemptive scheduling.



Which of the above statements are true? → II and III.

Q3. Which combination of the following feature will suffice to characterise an OS as multiprogrammed OS?

(a) More than one program may be loaded into main memory at the same time for execution.

Ans: only 'a'

(b) If a program wait for certain events such as I/O, another program is immediately scheduled for execution

(c) If the execution of program terminates, another program is immediately scheduled for execution.

Q4. The maximum number of processes that can be in ready state for a computer system with 'n' CPUs:

Ans: Independent of n. (theoritically there can be infinite no. of programs in ready state).

* Ans will be n if 'running state' was asked.

Qs. Consider the following statements about process state transitions for a system using preemptive scheduling.

I. A running process can move to ready state
II. A ready process can move to running state
III. A blocked process can move to running state
IV. A blocked process can move to ready state.

Which of the above statements are true? Ans: I, II, IV

Q6. Consider a system having 'n' cpus ($n \geq 1$) and 'k' processes ($k > n$). Calculate lower bound and upper bound of the number of processes that can be in the ready, running and blocked states.

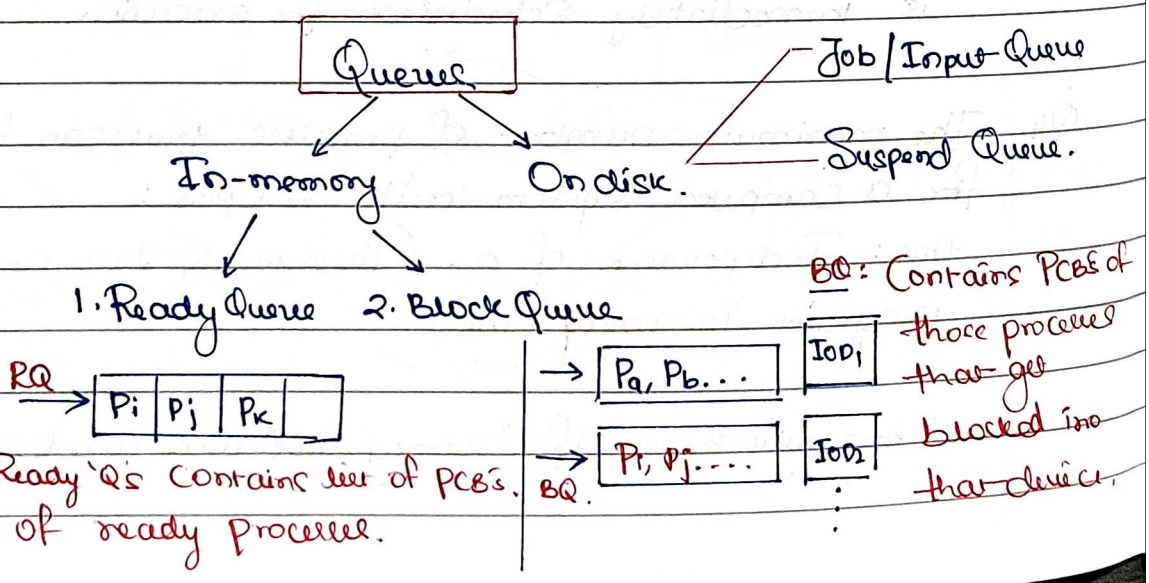
	L.B.	U.P
Ready	0	k
Running	0	n
Block.	0	k

n = no. of cpus ($n \geq 1$)
k = no. of processes. ($k \geq n$)

	L.B.	U.P.
Ready	0	k
Running	0	k
Block	0	k

n = no. of cpus ($n > 1$)
k = no of processes ($k < n$)

Scheduling Queues and State-queuing diagram



Date 19/01/23

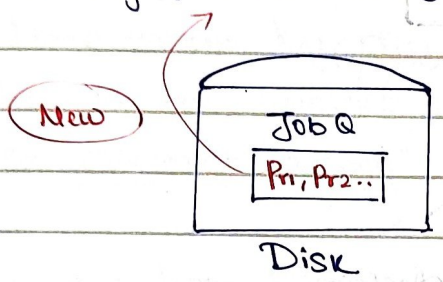
Queue on disk

1). Job Queue
Input Queue

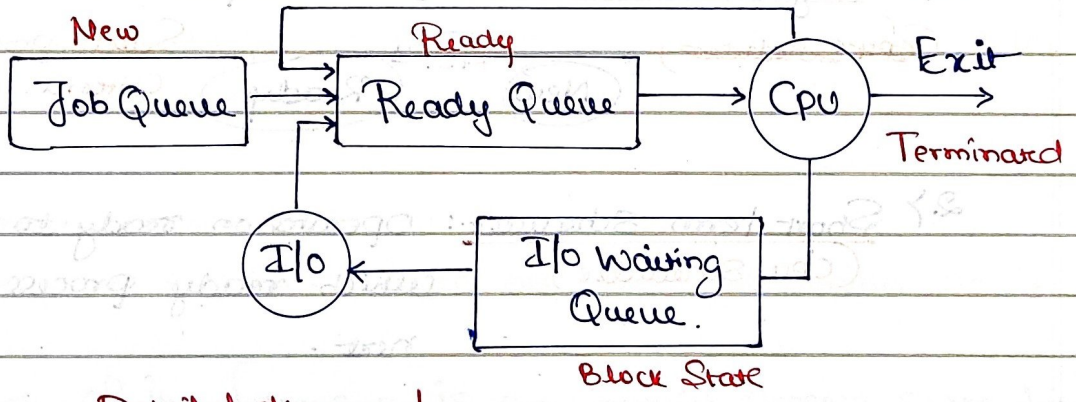
* Contains programs that are ready to be loaded in memory

2.) Suspend Queue

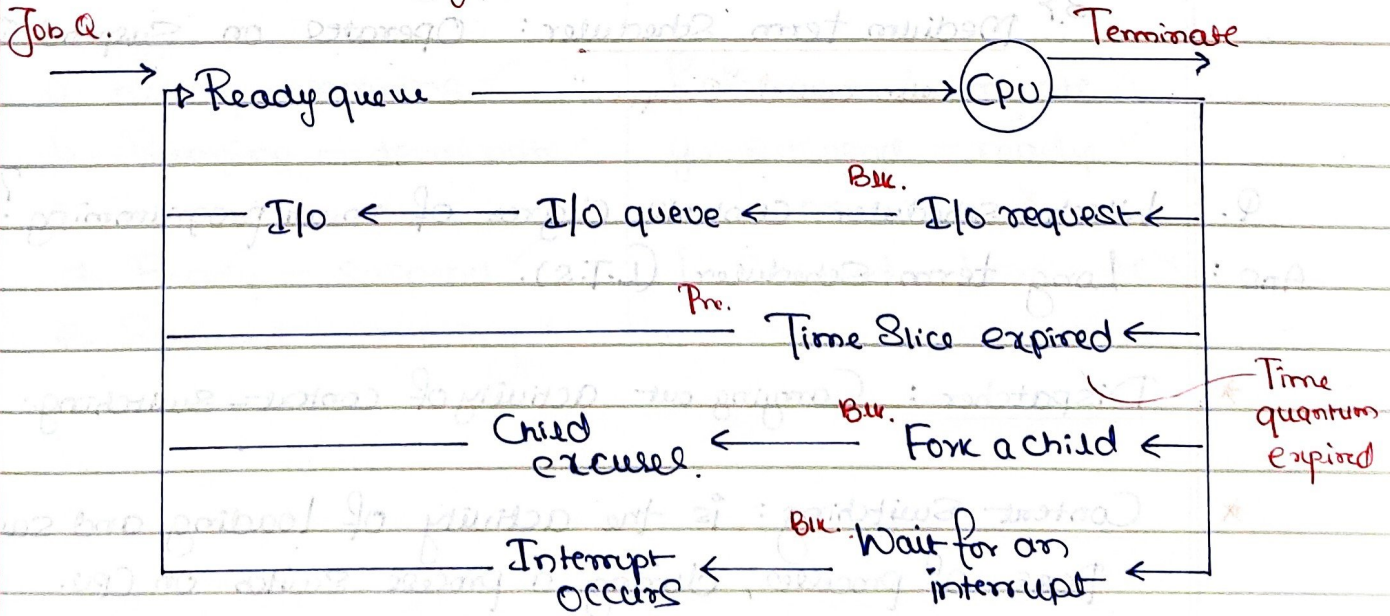
* Contains list of processes that get suspended from memory into disk.



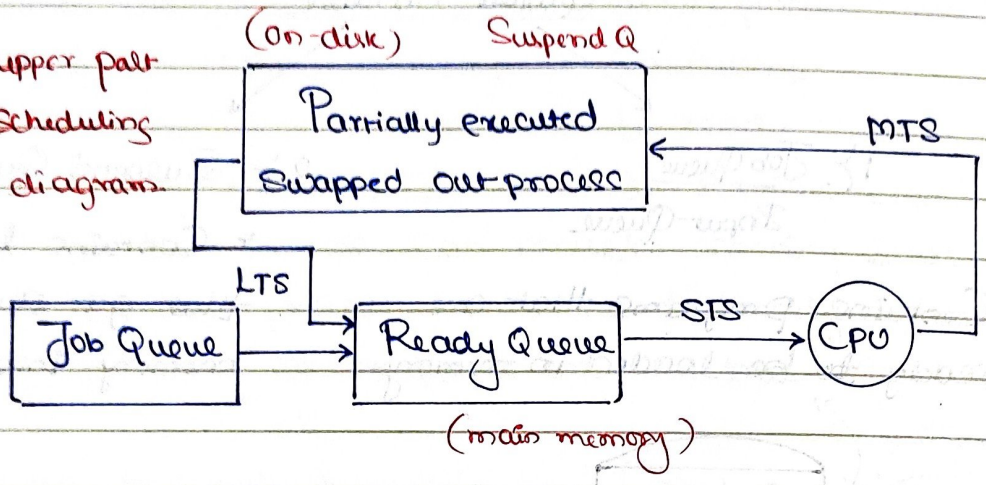
State Queuing Diagram:



Detailed diagram ↓



The upper part of Scheduling queue diagram.



Schedulers and Dispatchers

* Scheduling means making a decision
* Scheduler is a component of O.S that makes decisions.
* There are three types of schedulers:

1.) Long term scheduler (LTS): operates on Job-Q.
(~~CPU Scheduler~~)

Selects process based on a criteria

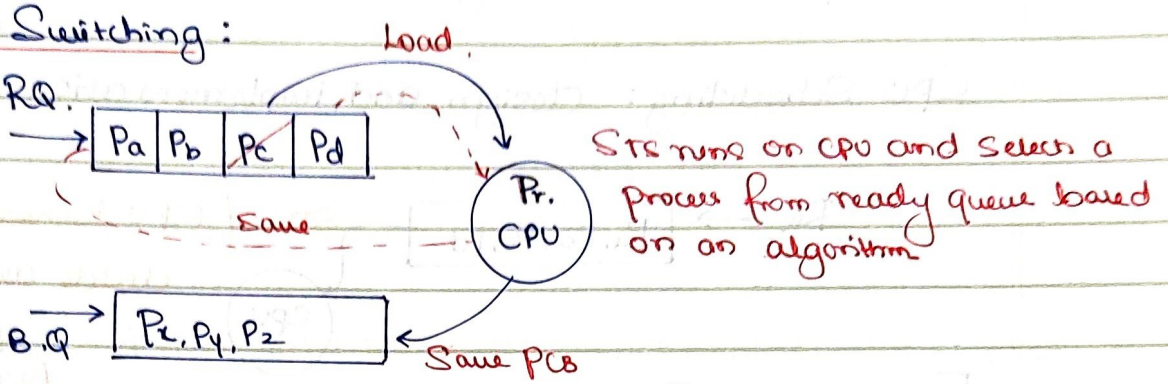
2.) Short term scheduler: operates on ready to Queue, to decide which ready process should run next.
(CPU Scheduler)

3.) Medium term Scheduler: Operates on Suspend Queue.
also called "Swapper".

Q. Which scheduler controls degree of multi programming?
Ans: Long term Scheduler (L.T.S).

* Dispatcher: Carrying out activity of context-switching.
* Content Switching: is the activity of loading and saving the PCBs of process, during a process switch on CPU.

Context Switching :



* Time taken by dispatcher to load and save the PCB's is known as "Context Switching time" or "CPU Scheduling Overhead" or "dispatch latency"

Q1. Let the time taken to switch between user and kernel modes of execution be t_1, while the time taken to switch between two user processes be t_2. Which of the following is true?

Context switching time = t_2
mode shifting time = t_1
$\because t_1 < t_2$.

* process switching involves mode shifting time

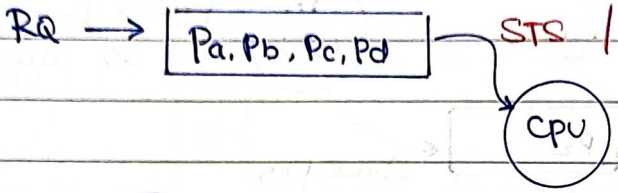
Q2 Dispatch latency is defined as:
The time to switch from one process to another on CPU

Q3. Which of the following is an invalid process state transition?

a. Ready — running ✓
b. Running — terminate ✓
c. Block — ready ✓
d. Ready — suspend ✓
e. Suspend — running ✗
f. block — terminate ✗
g. suspend — ready ✓
h. block — new ✗
i. Suspend — terminate ✗

CPU Scheduling

CPU Scheduling: design and implementation of S.T.S.



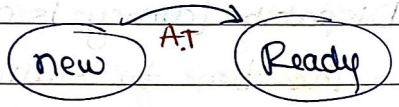
Functions: (S.T.S) Select a process from R.Q to run into CPU.

Goals (Scheduling Criteria):

* Maximize CPU utilization (throughput)
* Minimise waiting time (WT), turn around time (TAT), response time (RT).

Process Times:

1. Arrival time (AT): first time when the process comes from new to ready state is called arrival time.



2. Waiting time (WT): total time spent by the process in ready queue.

3. Burst time (B.T): time spent by the process executing in the CPU is called CPU burst time.

4. I/O Burst time (IOBT): time spent by the process in block queue performing I/O operation is called I/O burst time

5. Completion time: time at which process complete.

6. Turn Around time (TAT): the time taken by the process from its arrival time to completion time is called turn around time.

$$TAT = C.T - A.T.$$

7. Waiting time (WT): total time spent by the process in ready queue is called waiting time.

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

Note:

1. n processes $(P_1 \dots P_n)$
2. $AT(P_i) = A_i$
3. $BT(P_i) = X_i$
4. $IOBT(P_i) = Y_i$
5. $CT(P_i) = C_i$

a) $TAT(P_i) = (C_i - A_i)$

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

b) $WT(P_i) = (C_i - A_i) - (X_i + Y_i)$

$$\text{Average WT} = \frac{\sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)}{n}$$

Schedule length (L): Total time taken to complete all 'n' processes as per schedule

* No. of schedules with n-processes = n! (non preemptive)
* No. of schedules with n-processes = ∞ (pre-emptive)

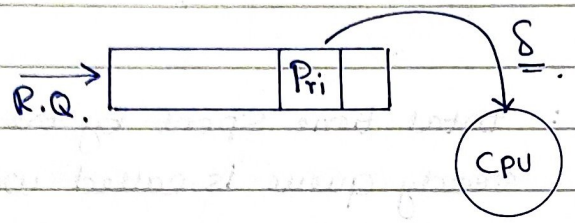
$$L = (\text{completion time of last process}) - (\text{A.T of last process})$$
$$\Rightarrow \max(C_i) - \min(A_i)$$

Throughput = no. of processes completed per unit time.

$$\eta = \frac{n}{L}$$

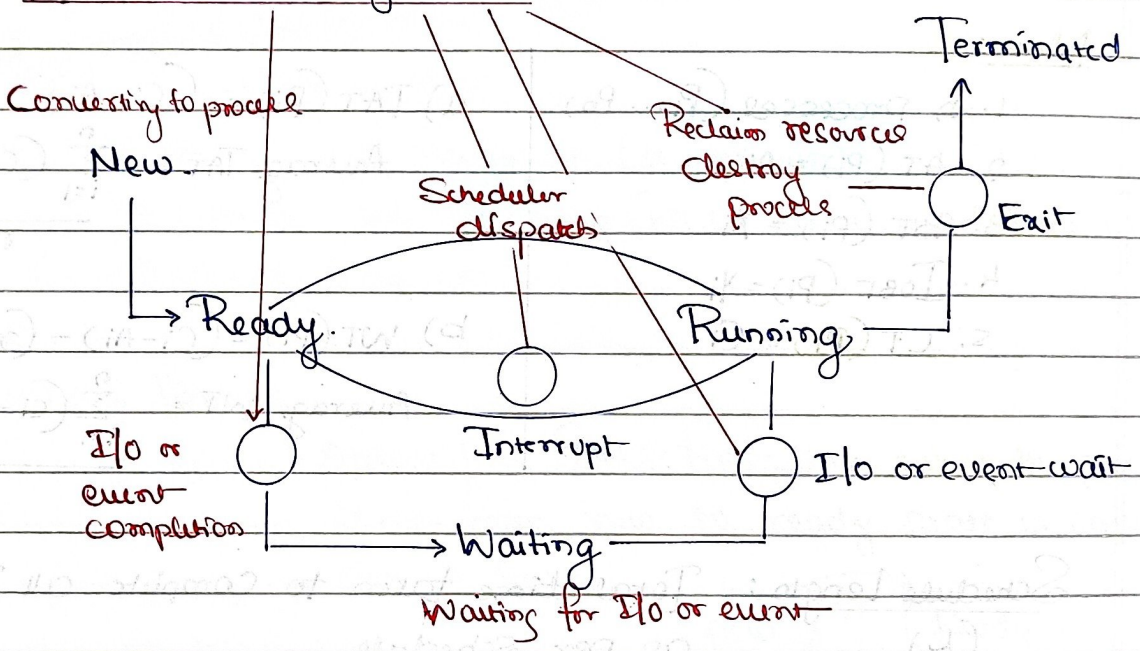
Context-Switch time : CPU Scheduling Overhead

Let CPU Scheduling overhead = "δ"



δ → Time taken to load PCB to process from Ready queue onto CPU.

CPU Scheduling Occurs



Types of CPU Scheduling :

1. Pre emptive
2. Non preemptive.

Scheduling Criteria :

Maximize	Minimize
* CPU utilization	* Turnaround time * Waiting time * Response time

CPU - I/O Bursts:

Process execution consists of a cycle of CPU execution and I/O wait:

* different processes may have distributions of bursts.
* CPU bound processes: performs lots of computations in long bursts, very little I/O.
* I/O bound process: performs lots of I/O followed by short bursts of computation.
* ideally, the system admits a mix of CPU bound I/O bound processes to maximise CPU and I/O.

Example:

:
load store
add store
read from file } CPU Burst

wait for I/O } I/O Burst

store increment index
write of file } CPU Burst

wait for I/O } I/O Burst

load store
add store
read from file } CPU Burst

wait for I/O } I/O Burst.

:

1. FCFS (First Come First Serve)

* Selection Criteria : A.T
* mode of operation : Non pre-emptive.

* Conflict resolution : Lower process id.

* Assumption :

(i) Time is in clock ticks

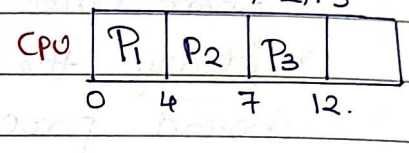
(ii) No IOBTS

(iii) Scheduling overheads (δ) = '0'.

Ex:1

P.no.	A.T.	B.T	CT.	TAT	WT.
1	0	4	4	4	0
2	0	3	7	7	4
3.	0	5	12	12	7

RQ → P_1, P_2, P_3



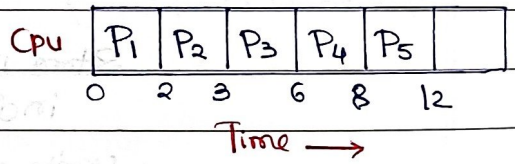
Gantt Chart →

Avg TAT = $\frac{23}{3}$. Avg. WT = $\frac{11}{3}$.

L = 12.

Ex 2

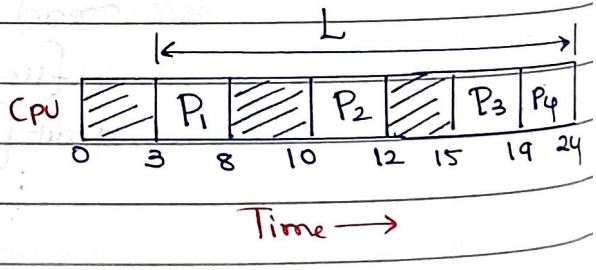
P.no	A.T	B.T	C.T.	TAT.	WT.
1	0	2	2	2	0
2	0	1	3	3	2
3	2	3	6	4	1
4	3	2	8	5	3
5	5	4	12	7	3



Avg. TAT = $\frac{21}{5}$, Avg. WT = $\frac{9}{5}$, L = 12. $\eta = \frac{\Omega}{L} = \frac{5}{12}$.

Ex 3

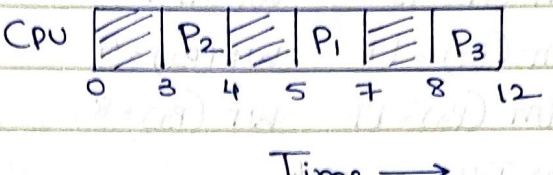
P.no	A.T	B.T	C.T	TAT	WT
1	3	5	8	5	0
2	10	2	12	2	0
3	15	4	19	4	0
4	18	5	24	6	1



% CPU idleness / L = $\frac{5}{21}$.

Ex.4.

P.no	A.T	B.T
1	5	2
2	3	1
3	8	4

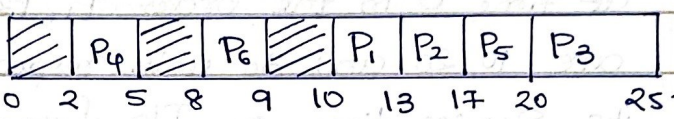


L = 12 - 3 = 9

Ex 5.

P.no	A.T	B.T
1	10	3
2	11	4
3	20	5
4	2	3
5	12	3
6	8	1.

→ RQ : P4 P6 P1 P5 P3.



L = 25 - 2 = 23.

% CPU idleness = 4/23.

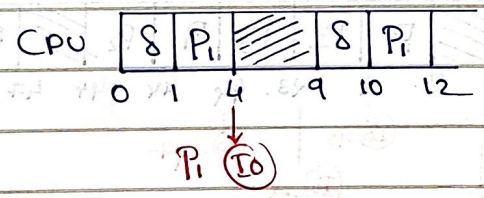
CPU Scheduling Techniques / Algorithms

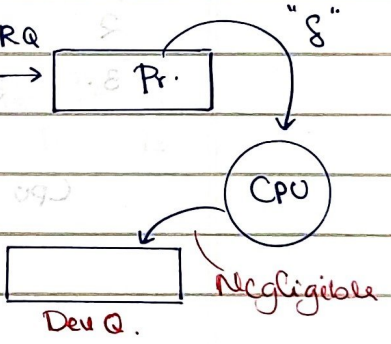
1. FCFS: Scheduling with IO-BT's and CPU Scheduling overheads.

Ex 1: δ = 1

P.no	A.T	<B.T : IOBT : BT>
1	0	<3 ; 5 ; 2 :>

RQ → P1

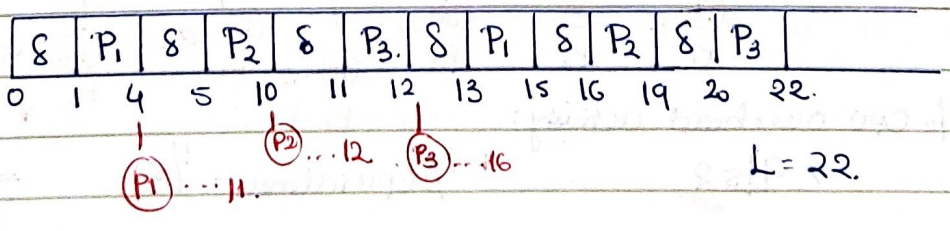




Ex 2: δ = 1.

P.no	A.T	<B.T. IOBT ; BT>
1	0	3 7 2
2	2	5 2 3
3	5	1 4 2

System has multiple IO devices.



L = 22.

TAT (P_1) = 15 WT (P_1) = 1

TAT (P_2) = 17 WT (P_2) = 5

TAT (P_3) = 17 WT (P_3) = 8

% CPU Idleness = 0%

* With 'δ' > 0 the formula for W.T becomes

$WT = TAT - (BT + IOBT + n.\delta)$

* n = no of times process get Scheduled into CPU.

Q1. Consider three processes P_1, P_2, P_3 arriving in the ready queue at time 0 in the order P_1, P_2, P_3. Their Service time requirements are 10, 20 and 30 units respectively. Each process Spends 20% of its Service time on I/O followed by 70% of its Service time on Computation at CPU and last 10% on IO before Complete. Assuming Concurrent I/O and negligible Scheduling overhead, Calculate FCFS Scheduling:

(i) Average TAT (ii) % CPU Idleness.

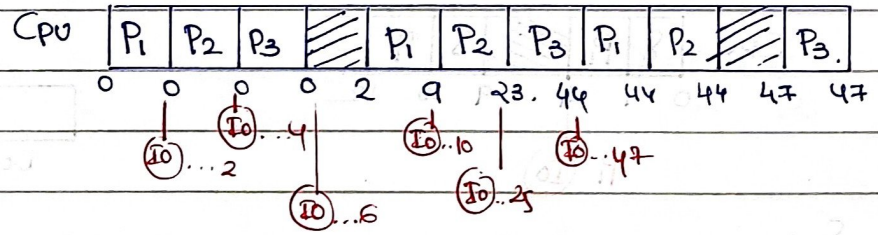
L = 47

% CPU idleness = 5/47

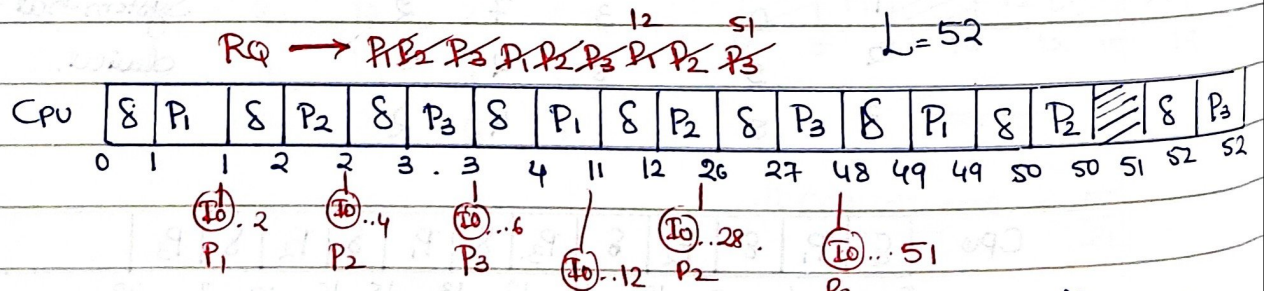
Avg TAT = (44+44+47)/3 = 44

Sol^n:

P.no	S.T	AT	<IOBT ; BT; IOBT;>
1	10	0	2 7 1
2	20	0	4 14 2
3.	30	0	6 21. 3



Above problem with δ = 1.



* % CPU overhead activity: ⇒ 9/52

* % CPU idleness = 1/52

* % CPU efficiency ⇒ $100 - (\frac{9}{52} + \frac{1}{52})$

Date 23/01/23
Page

Homework: Repeat Q 2 and 3 assuming System has only one IO-device.

1). δ=2

P.no	AT	<BT	IOBT	BT>
1	0	3	4	2
2	5	2	0	3
3	20	1	15	6

2) δ=1.

P.no	AT	IO.	BT.	IOBT
1	3	5	2	3
2	8	2	10	4
3	12	6	12	1.

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

* Selection Criteria: Burst Time
* Mode of operation: Non preemptive
* Conflict resolution: Lower process id.
  : Among the processes present in RQ Select the one having least B.T and Schedule it on CPU.

Ex.1

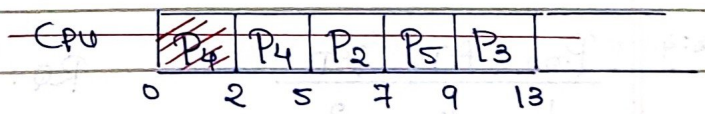
P.no	A.T	B.T
1	0	2
2	2	3
3	3	1
4	5	2
5	8	3
6	10	2

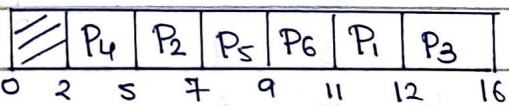
RQ: P1 P2 P3 P4 P5 P6

Ex.2

P.no	A.T	B.T
1	10	1
2	4	2
3	6	4
4	2	3
5	5	2
6	3.	2

~~RQ: P4, P2, P5, P3~~





L = 16-2 = 14

% CPU idleness = 0

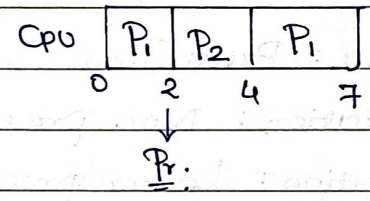
3. Shortest Remaining Time First (SRTF) / Pre emptive SJF

* Selection Criteria: BT
* mode of operation : Pre emptive
* Conflict resolution: Lower process id

→ Pre emption of running process is based on availability of Strictly Shorter process.

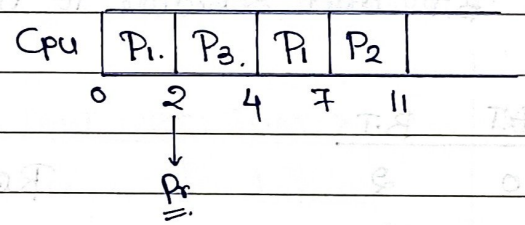
Ex: 1.

P.no	AT	BT
1	0	5
2	2	2



Ex: 2

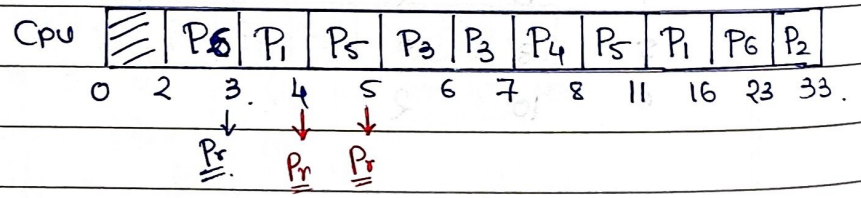
P.no.	A.T.	BT
1	0	5
2	1	4
3	2	2



Ex: 3

P.no	A.T	BT
1	3	6
2	2	10
3	5	2
4	6	1
5	4	4
6	2	8

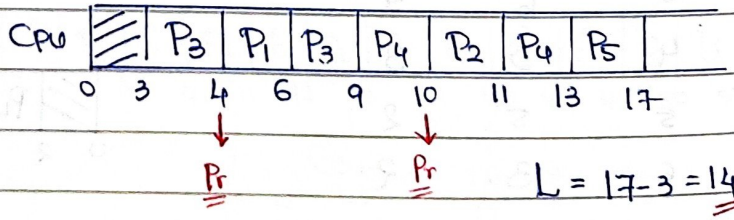
RQ: P2, P6 P1. P5 P3 P4



Ex: 4

P.no	AT.	BT
1	4	2
2	10	1
3	3	4
4	5	3
5	3	4

RQ: P3, P5, P1, P4 P2

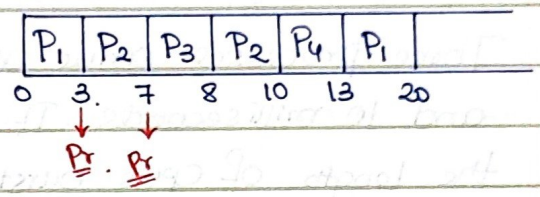


L = 17 - 3 = 14

Q1. Consider the following processes, with AT and length of the CPU burst given in ms. The scheduling algo used is premptive SRTF.

Process	AT.	B.T
P_1	0	10
P_2	3	6
P_3	7	1
P_4	8	3

The average turn around time of these processes is 8.25



Average TAT = $\frac{20+7+1+5}{4}$

$= \frac{33}{4} = 8.25$

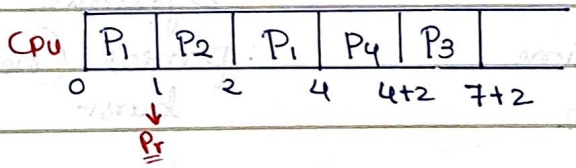
Q2.

Process	P_1	P_2	P_3	P_4
A.T	0	1	3	4
B.T	3	1	3	z (z)

The processes are run on a single processor using pre-emptive SRTF. If avg WT = 1ms, then value of z = ?

Average WT = 1 ms (SRTF)

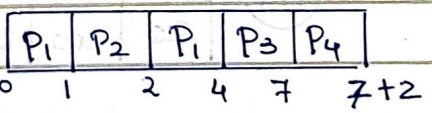
→ Case 1: $z < 3$.



Avg WT = $\frac{1+0+(2+1)+0}{4}$

$1 = \frac{z+2}{4}$ $\therefore z = 2$

Case 2: $z > 3$.



Avg WT = $\frac{1+0+1+3}{4} = \frac{5}{4} = 1.25$.

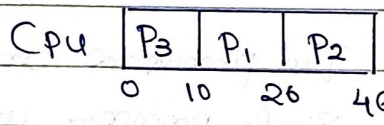
Because Avg WT = 1. So z is less than 3.

Performance of SJF/SRTF:

1. It favors shorter process:
   - → Advantages: * Complete more processes, max throughput
     * minimizing of Avg TAT, Avg WT.
   - → Drawback: * Starvation to longer processes.

Note: * SJF/SRTF is considered the "optimal algorithm"
* Since B.T's of processes are not known aprior, SJF/SRTF is "not practically implementable"!
* SJF is used as a benchmark to measure performance of other algorithms.
* SJF can be implemented with predicted burst times.

Q3. Three processes arrive at time zero with CPU bursts of 16, 20 and 10 milliseconds. If the scheduler has prior knowledge about the lengths of CPUs bursts, the min achievable average waiting time for these three processes in non-preemptive scheduler is 12 milliseconds.



Avg WT = $\frac{10+26+0}{3} = \frac{36}{3} = 12$

Prediction Techniques

- Static : Total burst time of processes.
  - (Process size) Bytes.
  - Type
- Dynamic : partial (next CPU Burst) burst

Pri

	1.			2.		
WT_1	BT.1	IOBT	WT_2	BT_2		?
RQ	CPU	IOD	R.Q	CPU	RQ	CPU

P_i 't' ; next CPU burst

Exponential Averaging Technique / Aging Algorithm: (To predict next CPU burst)

Let P_i : process
Let t_i : Completed B.T
Let T_i : predicted B.T

Let T_{n+1} denote next predicted B.T

$$T_{n+1} = \alpha t_{nn} + (1-\alpha) T_n$$
$$0 < \alpha < 1$$

Recurrence Relation:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \quad -(I)$$
$$\tau_n = (\alpha t_{n-1} + (1-\alpha)\tau_n \quad -(II)$$

} Back Substitution

$$\tau_{n+1} = \alpha t_n + (1-\alpha)[\alpha t_{n-1} + (1-\alpha)\tau_{n-1}]$$
$$= \alpha t_n + \alpha(1-\alpha)t_{n-1} + (1-\alpha)^2 \tau_{n-1} \quad -(III)$$
$$= \alpha t_n + \alpha(1-\alpha)t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + (1-\alpha)^3 \tau_{n-2} \quad -(IV)$$
$$\vdots$$

(τ_1 = initial guess)

Given the value of 'α' and τ_1 one can predict next CPU B.T.

Q1. Consider a system using exponential averaging technique to predict next CPU BT. Given $\alpha = 0.5$ and $\tau_1 = 10$. The process B.T's in previous runs are: 4, 8, 12, 10. Predict the next CPU burst time of process.

Soln:- $\tau_5 = ?$ 4, 8, 12, 10 (t_1, t_2, t_3, t_4)

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

$\tau_5 = 0.5 \times t_4 + 0.5 \times \tau_4$ (9.75)
$= \frac{1}{2} \times (t_4 + \tau_4) = \frac{1}{2}(10 + \tau_4)$

$\tau_4 = \frac{1}{2}(t_3 + \tau_3) = \frac{1}{2}(12 + \tau_3)$ (7)
$= \frac{19.5}{2} = 9.75$

$\tau_3 = \frac{1}{2}(t_2 + \tau_2) = \frac{1}{2}(8 + \tau_2) = 7.5$ (7)

$\tau_2 = \frac{1}{2}(t_1 + \tau_1) = \frac{1}{2}(4+10)$

$\tau_2 = 7.$

$\therefore \tau_5 = \frac{1}{2}(10 + 9.75) = \frac{19.75}{2} = 9.875$

4 Highest Response Ratio Next (HRRN)

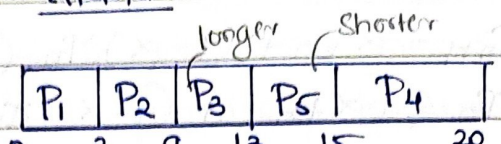
→ : HRRN not only favours shorter process but also limits the waiting time of longer process.

* Selection Criteria: Response ration : $\frac{W.T + BT}{BT}$
* Mode of operation: Non pre emp.

Ex 1.

P.no	AT	BT
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

HRRN:



t=13 R4 = 7+5/5 = 12/5

R5 = 5+2/2 = 7/2 (higher)

5. Longest Remaining Time First (LRTF)

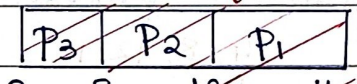
→ It is just opposite of ~~longest~~ Smallest remaining time first.

* Selection Criteria: Burst Time
* Mode of operation: Pre emptive.

Ex 1.

P.no	AT	BT
1	0	2
2	0	4
3	0	8

CPU. (it's pre emp)



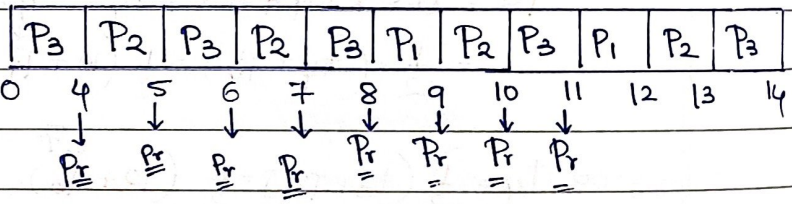
TAT:

P1 = 14-0 = 14

P2 = 12-0 = 12

P3 = 8-0 = 8.

Cpu



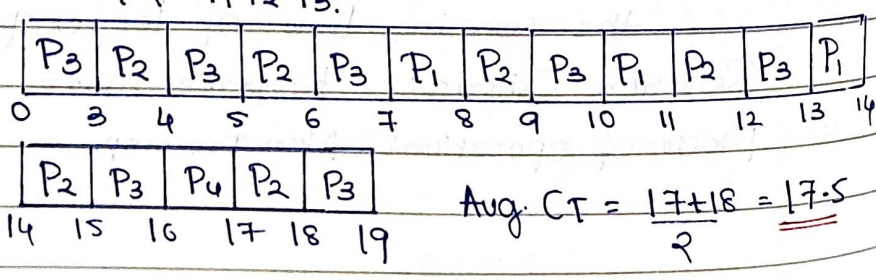
Avg T.AT = 12+13+14/3

Avg T.A.T. = 13

Q1. Consider 3 processes A, B, C with compute BT.s are 4, 6, 9 units. All the processes arrive at time zero. Consider the longest remaining time first (LRTF). In LRTF ties are broken by giving priority to processes with lowest Pid. Find the avg of completion times of A, B?

RQ: P1 P2 P3

P.no.	AT	BT
A	0	4
B	0	6
C	0	9



Avg. CT = 17+18/2 = 17.5

Q2 In a system using single processer, a new process arrives at the rate of 10 processes per minute and each such process require 5 seconds of service time. What is % CPU utilization?

Sol^n :- 60 $ → 10 processes
→ (5 Sec)

[(P1, P2 ... P10)]

60 sec cycle.

% CPU Utilization = $\frac{50}{60} \times 100$

= 83.33%

Q3. Six jobs are waiting to be run. The expected running times are 9, 7, 5, 2, 1 and x respectively. Where $5 < x < 7$ and avg CT. is 13. Find the value of x using SJF algorithm? (Assume all jobs arrive at same time = 0).

6. Priority Based Scheduling

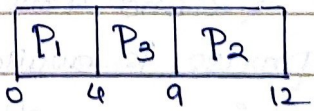
→ Priority based scheduling works exactly like SJF/SRTF except that it looks at priority instead of Burst time.

* Selection Criteria : Priority
* Mode of operation : Non preemptive / preemptive
* Tie breaking : Lower process id.

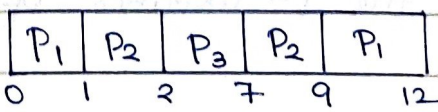
Ex 1.

Prio	P.no	AT	BT
4	1	0	4
5	2	1	3
8	3	2	5

Non pre-emp



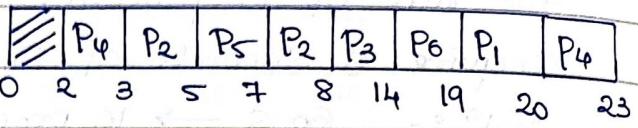
Pre emp.



Ex 2.

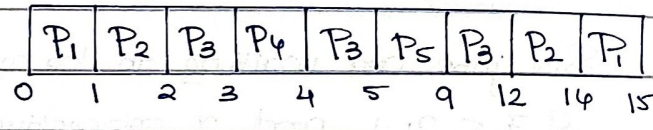
Prio.	P.no	AT	BT
4	1	4	1
6	2	3	3
8	3	8	6
3	4	2	4
7	5	5	2
5	6	3	5

$L = 23 - 2 = 21.$



Ex 3.

Prio	P.no	AT	BT
4	1	0	2 1
5	2	1	3
6	3	2	5
7	4	3	1
8	5	5	4



Avg TAT = $\frac{15+13+10+1+4}{5} = \frac{43}{5} = 8.6$

Avg WT =

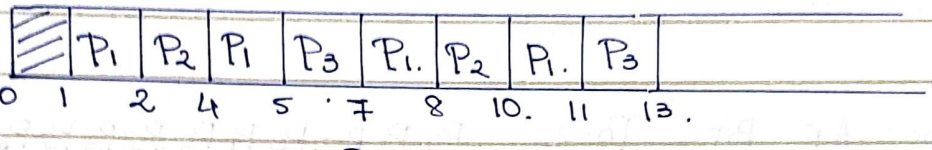
Performance of priority based scheduling:

* Causes Starvation to low priority processes, a solution exists "Dynamic priorities": It increases the priority of the processes at regular intervals of time using an algorithm called "Aging" algorithm.

Q1. Consider a System with Pre-emptive priority based scheduling with processes P_1, P_2, P_3 having infinite instances of them. The instance of these processes arrive at regular intervals of 3, 7 and 20 ms respectively. The priority of these process instances is inverse of their periods. Each of the process instance P_1, P_2, P_3 consumes 4, 2 and 4 ms CPU time respectively. The 1st instance of each process is available at 1ms. What is the completion time of the 1st instance of process P_3?

Prio.	Period	P.no	AT	BT	<Other instances>
1/3	3	1	1	1	<4, 7, 10, 13, 17...>
1/7	7	2	1	2	< 8, 15, 22, ... >
1/20	20	3	1	4	<21, 41, 61. . . . >

RQ: $P_1, P_2, P_3, P_{1_2}, P_{2_2}$



Completion time of P_3 first instance is 13 ms (at the end of 12)

Q2. Consider a System using preemptive priority based scheduling with dynamically changing priority. On its arrival a process is assigned a priority of zero and running process priority increases at the rate of 'β' and priority of the processes in the ready Q increases at the rate of 'α'. By dynamically changing values of α and β one can achieve different scheduling disciplines among the processes. What discipline will be followed for the following conditions

1. $\beta > \alpha > 0$ 2. $\alpha < \beta < 0$.

Ans: FCFS for first condition and LIFO for second condition.

7. Round Robin Scheduling

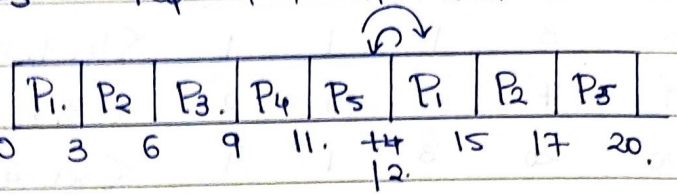
→ It is used in Pre-emptive multi programmed time shared operating systems.

* Selection Criteria: Arrival time + Time Quantum
* mode of operation: Pre emptive
* Working principle: - each process is allotted a fixed time quantum
  - Pre emption is based on completion or expiration of time quantum.

Ex. 1.

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

TQ=3 RQ= 1, 2, 3, 4, 1, 5, 3, 5.

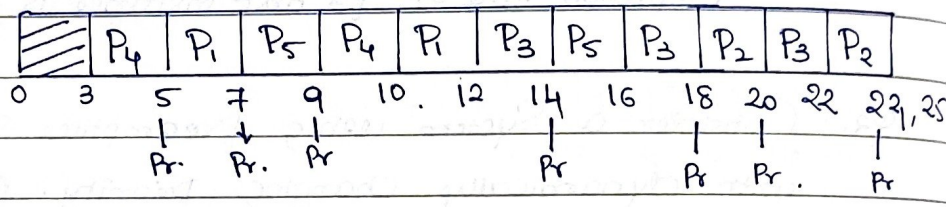


L = 20.

Ex 2.

Pno	AT	BT
1	4	4
2	15	5
3	8	6
4	3	3
5	5	4

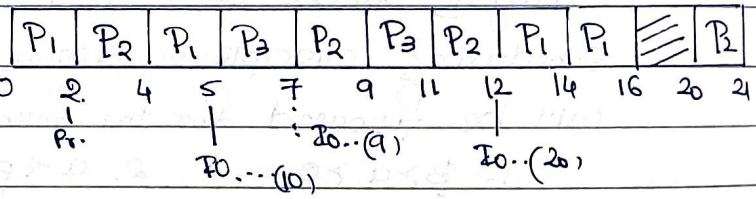
TQ=2 P4 P1 P5 P4 P1 P3 P5 P3 P2 P3 P2



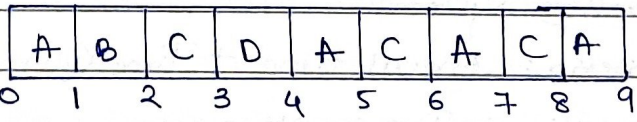
Pno	AT	BT	IOBT	BT
1	0	3	5	4
2	2	5	8	1
3	3	2	2	2

TQ=2

RQ: P1 P2 P1 P3 P2 P3 P2 P1



Q1. Consider a set of 4 processes, ABCD arriving in the order at time 0. Their burst time req are : 4, 1, 8, 1 respectively using RR scheduling with TQ : 1. The completion time of process A is 9.



Q2. Consider a system using RR Scheduling with 10 processes all arriving at time 0. Each process is associated with 20 identical request. Each process request consumes 20 ms of CPU time after which it spends 10 ms of time on I/O there after, initates subsequent request. Assuming scheduling overhead of 2 ms and

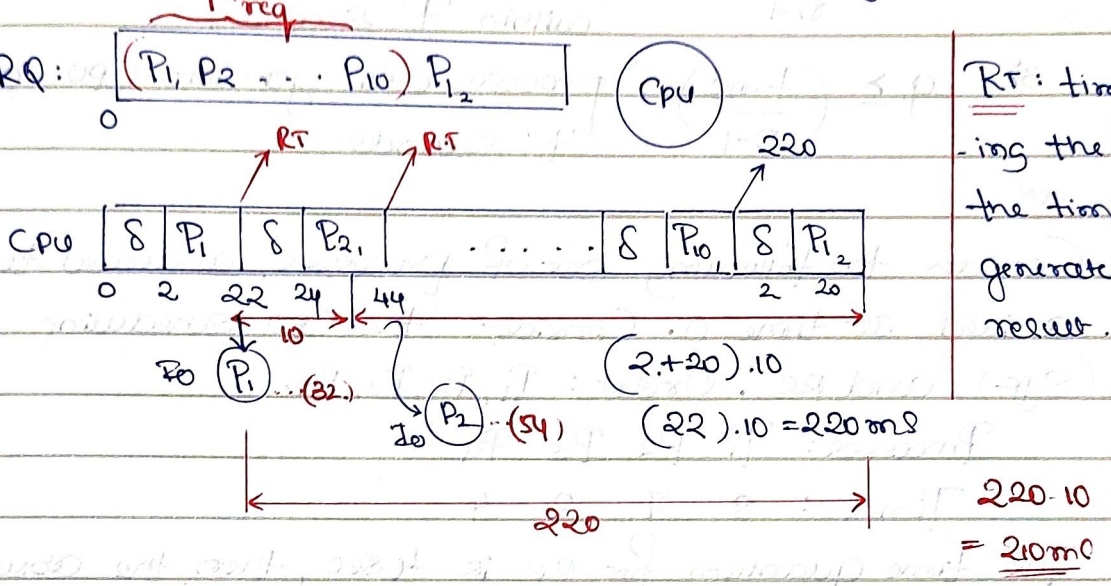
TQ of 20 ms. Calculate:

(i) Response time of 1st request of 1st process. – ans: 22 ms

(ii) Response time of 1st request of last process – Ans: 220 ms

(iii) Response time of subsequent req. of any process – Ans: 210 ms

Sol:



RT: time of admitting the request to the time it generates into result.

Performance of Round Robin

Time Quantum.

very small: Efficiency = 0

very large: same like FCFS

Small

→ More context Switching Overhead

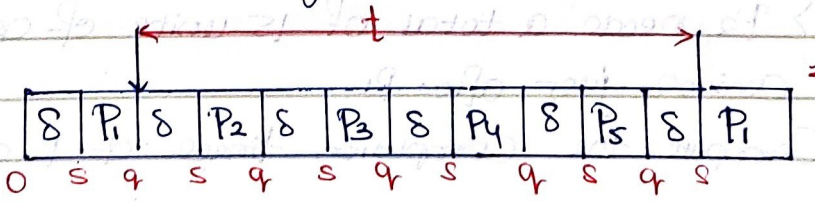
Makes the System better responsive

Large

→ Less context switching (less overhead), Poor interactiveness, responsiveness.

Q1. Consider a system with 'n' processes arriving at time 0+ with substantially large BT. The CPU scheduling overhead is 's' seconds, Time quantum is 'q' seconds. Using RR scheduling what must be value of time quantum 'q' such that each process is guaranteed to get its turn at CPU exactly after 't' seconds in its subsequent run at CPU.

→ $t - ns = (n-1)q$



$\Rightarrow q = \frac{t - ns}{n-1}$

1) $q = \left(\frac{t-ns}{n-1}\right)$: process will get into CPU exactly after 't' seconds.

2) $q < \left(\frac{t-ns}{n-1}\right)$: process will get into CPU atleast once within 't' seconds.

3) $q \geq \left(\frac{t-ns}{n-1}\right)$: process will get into CPU atleast every 't' seconds

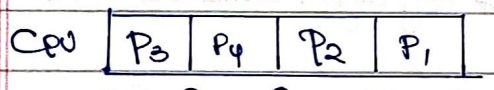
HW

Q2. Consider the following set of processes, assumed to have arrived at time 0. Consider the CPU scheduling algorithms (SJF) and RR. Order: P_1, P_2, P_3, P_4.

Processes:	P_1	P_2	P_3	P_4
Times :	8	7	2	4

The time quantum for RR is 4 sec, then the absolute value of average turnaround times in SJF and RR is

SJF



$$\text{Avg T.AT} = \frac{21+13+2+6}{4}$$

$$= \frac{42}{4} = 10.5$$

RR.

HW

Q3. Consider processes P_1 and P_2 arriving in RQ at time 0 with:

i) P_1 needs a total of 12 units of CPU time and 20 units of I/O time. After every 3 units of CPU time P_1 spends 5 units on I/O.

ii) P_2 needs a total of 15 units of CPU time and no I/O. P_2 arrived just after P_1.

Compute the completion times of P_1 and P_2 using SRTF & RR.

HW

Q4. Consider 4 processes P, Q, R, S scheduled on a CPU as per RR algorithm with TQ = 4 units. Processes arrive in order P, Q, R, S all at time t = 0. There is exactly one context-switch from S to Q, exactly one context switch from R to Q and exactly two context switches from Q to R. There is no context switch from S to P. Switching to a ready process after termination of another process is also considered a context switch.

Q5. Which of the following statements is/are correct in context of CPU Scheduling?

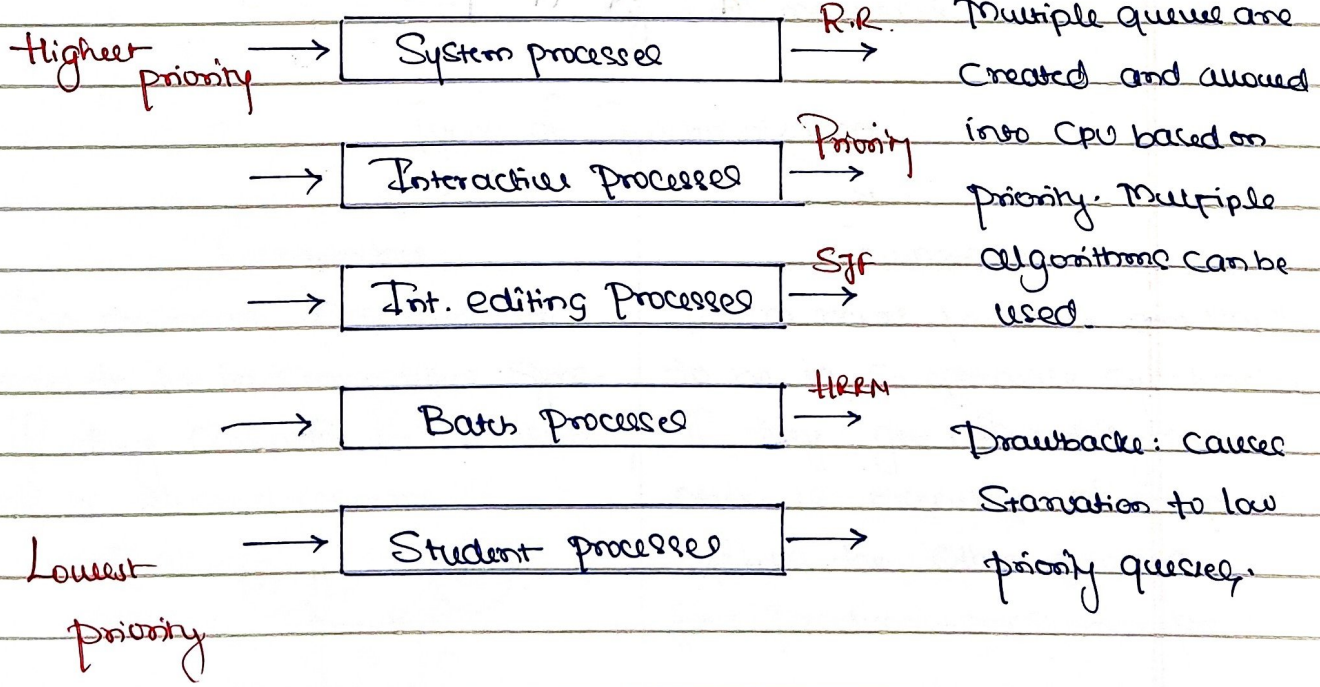
A. The goal is to only maximize CPU utilization and minimize throughput

B. Turnaround time includes waiting time

C. Implementing pre-emptive scheduling needs hardware support

D. RR policy can be used even when the CPU time required by each of the processes is not known apriori

8. Multi Level Queue Scheduling:



Multiple queues are created and allowed into CPU based on priority. Multiple algorithms can be used.

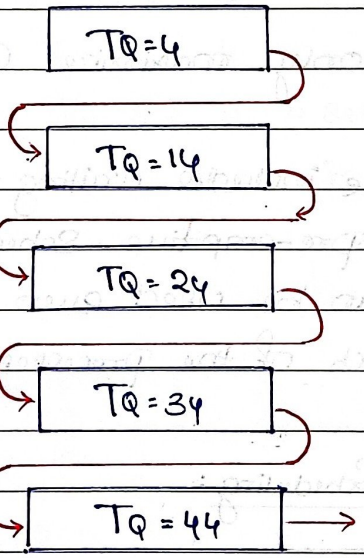
Drawbacks: causes starvation to low priority queues.

9. Multilevel feedback Queue Scheduling :

* Another way to put a reference on short lived processes, panelize processes that have been running longer.
* Pre emptive.

Q6. Consider a system which has CPU bound process, which require the burst time of 80 seconds, the multilevel feedback queue scheduling algorithm is used and the queue time quantum is '4' sec and in each level it is incremented by '10' seconds. Then how many times the process will be interrupted, and on which queue the process will terminate the execution ?

Ans : 4, 5.



B.T = 80ms.

4 + 14 + 24 + 34 = 80

Synchronization

★ **IPC**: Inter process communication is a mechanism that allows processes to communicate with each other and synchronize their actions.

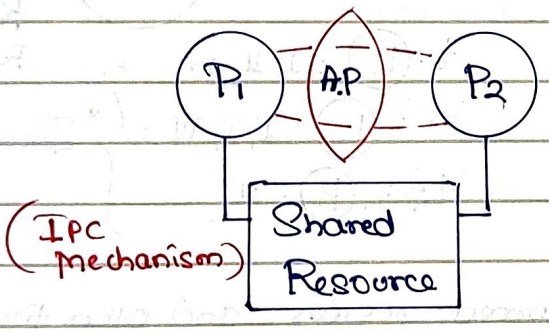
Every communication should be synchronized / co-ordinated

→ **Lack of synchronization in IPC environment leads to:**

1. Inconsistency (Incorrectness)
2. Loss of data
3. Deadlock (lockup)

Synchronization: It is agreed upon protocol in an IPC environment, to make sure that there is no inconsistency, data loss or deadlock.

IPC Environment:



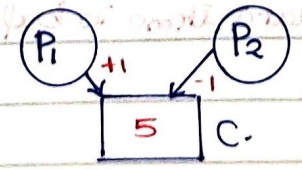
* Synchronization involves the orderly sharing of system resources by processes.
* We can think of this intersection as a system resource that is shared by two resources.

Types of Synchronization.

Competitive

Two or more processes are said to be in competitive Sync. if they compete for accessibility of a shared resource.

Ex:



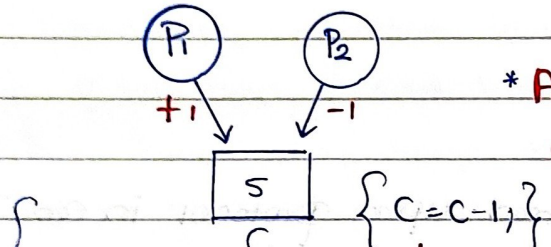
Co-operative

Two or more processes are said to be in Co-operative synchronization if they get affected by each other. ie execution of one process affects the other process.

Ex: Producer-consumer problem

Date 02/2/23

Note: * Lack of synchronization among completing processes may lead to either inconsistency, dataloss.
* Lack of synchronization among co-operating processes may lead to deadlock.
* An application of an IPC environment may involve either competition or cooperation or both type of synchronization



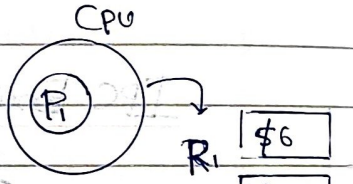
* Any process that executes in user mode can get preempted after any instruction.

I. Load R_1, C;
II. Inc R_1;
III. Store C, R_3;

Load R_2, C;
Dec R_2;
Store C, R_2;

Scenario

P_1, P_2



t: P_1 : I, II, Pr
t_1: P_2 : I, II, III ; } "Inconsistency"
t_2: P_1 : III

Note: * Sometimes we get correct results and other times it is possible to get incorrect results.
* As an end user we want a solution, that always gives correct result weather pre emption take place or not.

Implementation of producer Consumer problem

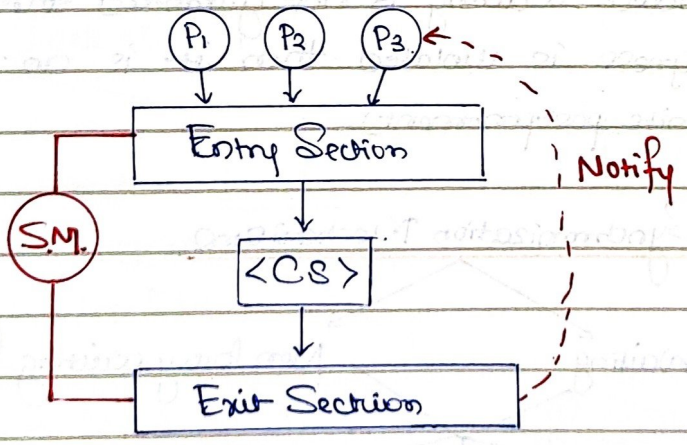
```
#define N 100
int buffer[N];
int count = 0; <No. of data items in buffer>
```

```
Void Producer (void)
{  int itemp; in = 0;
   while (1)
   {
NCS a> itemp = Produce-item();
CS  b> while (count == N);   Busy waiting
CS  c> Buffer [in] = itemp;
CS  d> in = (in+1) % N;
CS  e> count = count + 1; }
}
```

```
Void Consumer
{  int itemC, out = 0;
   while (1)                          Busy waiting
{  a> while (count == 0);
   b> itemC = Buffer [out];
   c> out = (out + 1) % N;
   d> count = count - 1;
   e> Process-item(); }
}
```

Necessary Conditions for Synchronization Problems

1. Critical Section: It is that part of the program where shared resources are accessed
2. Race conditions: Situation where in processes are trying to access critical section and final result depends on the order at which they finish their update.
3. Pre emption:




```
Do {
   Entry Section
      <Critical Section>
   Exit Section
      <Remainder Sect>
}
while (true);
```

Note: Process running in user mode can get prempted any time and any number of times (after completing any instance).

Date 7/02/23

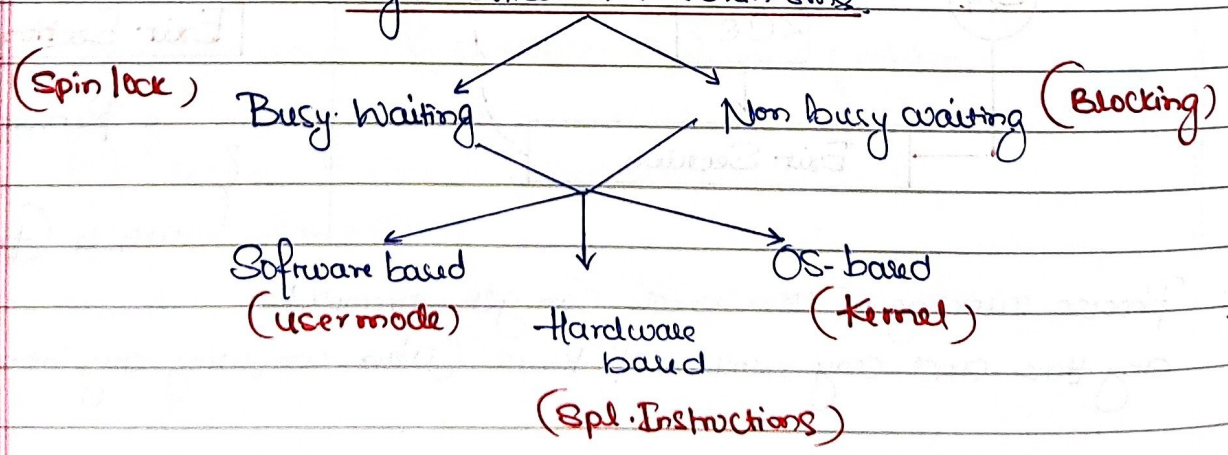
Requirements of C.S. Problems:

1. **Mutual Exclusion (M/E)**: If process P_i is executing in its Critical Section then no other processes can be executing in their Critical Sections.

2. **Progress**: If no process is executing in its critical sections and some processes wish to enter their Critical Sections, then only those processes that are not executing in their remainder Sections can participate in deciding which will enter its Critical Section next and this selection cannot be postponed indefinitely.

3. **Bounded waiting**: There exist a bound, or limit, on the number of times that other processes are allowed to enter their Critical Sections after a process has made a request to enter its Critical section and before that request is granted.

Note: * If mutual exclusion is not guaranteed then inconsistency + data loss may happen.
* If bounded waiting is not guaranteed then starvation.
* If progress is violated then it is an unfair Solution (Indefinite postponement).

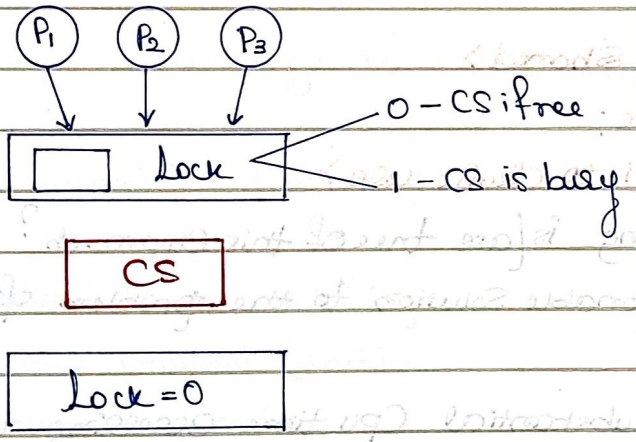


Assumptions:

* Process enters <CS> and come out of it in finite amount of time.
* When a process is in entry section then it means, it is interested in <CS>.
* A process is said to have left CS only when it has executed its Critical Section.
* A process can get pre empted from CPU while executing entry + <CS> + Exit Section.

1. Lock Variable:

- Busy waiting Solution
- Software Solution implementation at user mode
- Multi process Solution.



Implementation : High L.

```
int lock = 0;
void process (int i)
{ while (1)
  { a> NON-CS
    b> while (lock != 0);   } Entry Section
       lock = 1;
    c>  CS
    d> lock = 0; }          } Exit Section.
}
```

Implementation : Low Level.

```
integer lock = 0;
Process (int i):
a> NON-CS
b> Load Ri, lock;      }
c> Compare Ri, #0;     } Entry Section.
d> JNZ Step b;         }
e> Store lock, #1;     }
f> <CS>.
g> Store lock, #0      } Exit Section
```

* Lock variable does not always guarantee mutual exclusion along with bounded wait.

Date 8/2/23

Note: * Lock variable does not guarantee mutual exclusion always.
* Lock variable guarantees progress always
* Lock variable fails to satisfy bounded wait.

* Because a process can enter CS multiple times successfully while other processes are waiting for their turn to enter CS.
* Lock variable is a busy waiting solution leading to wastage of CPU time/cycles (in the loop).

Q1. Several concurrent processes are attempting to share an I/O device. In an attempt to achieve M/E each process is given the following structure.

```
<code unrelated to device use>
    Repeat
    until busy = false;
    Busy = true;
<code to access shared>
    Busy = false;
<code unrelated to device use>
```

Which of the following is/are true of this approach?

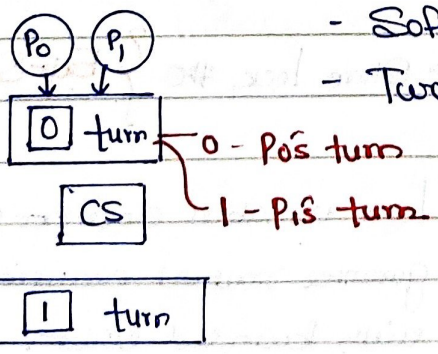
I. It provides a reasonable solution to the problem of guaranteeing M/E

II. It may consume substantial CPU time accessing the Busy variable.

III. It will fail to guarantee mutual exclusion.

2. Strict Alteration:

- Busy waiting
- Software solution implementable at user mode
- Two process solution <Pi/Pj>



"Strictly on alternate basis processes take turn to enter CS."

Implementation: High Level.

```
int turn = rand(0,1);
void process(0)
{
  while(1)
{ a> NON-CS();
  b> while(turn!=0)
  c> CS
  d> turn=1; }
}
```

```
void process(1)
{
  while(1)
{ a> NON-CS();
  b> while(turn!=1);
  c> CS
  d> turn=0; }
}
```

* Strict alternation always guarantee mutual exclusion.
* Strict alternation does not guarantee Progress.
* Strict alternation always guarantee bounded waiting.
* Like lock variable this is a busy waiting solution which leads to wastage of CPU time/cycles.

Generalised Implementation of Strict Alternation:

```
int turn = rand(i,j);
void process(int i)
{ int j= NOT(i);
  while(1)
{ a> NON-CS();
  b> while(turn!=i)
  c> CS
  d> turn=j; }
}
```

Q2 Consider the following two-process synchronization solution.

Process 0

Entry: loop while (turn == 1);

(critical section)

Exit: turn = 1;

Process 1

Entry: loop while (turn == 0);

(critical section)

Exit: turn = 0;

The shared variable turn is initialized to zero. Which of the following is true?

Ans: This solution violates progress requirement.

Q3. Code Snipped:

```
int turn = 0;
void Process(0)
{ while (1)
  { a> NON-CS();
    b> while (turn == 1);
    c> CS
    d> turn = 1; }
}
```

```
void process(1)
{ while (1)
  { a> NON-CS();
    b> while (turn == 1);
    c> CS
    d> turn = 1; }
}
```

Does the above code guarantees mutual exclusion?

Ans No.

* If turn = 1; then it will result in deadlock as no process can enter in the Critical Section.

Q4.

```
int turn = rand(i,j);
Void Process (int i)
{ int j = NOT (i);
  while (1)
  { a> NON-CS();
    b> while (turn != i);
       turn = j;
    c> CS
    d> turn = j; }
}
```

i> Does this guarantee M/E?
M/E is not guaranteed.

ii> Progress?

iii> Bounded Wait?

3. Peterson's Solution:

- Busy-waiting
- Software Solution implemented at user mode
- Two process Solution
- Combination of lock & Strict alternation.

```
#define N 2
#define True 1
#define False 0
int flag[N] = {false};
int turn;
```

```
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 and turn == i);
    e. CS
    f. flag[i] = False; }
}
```

Note: Peterson's Solution guarantees M/E always.

* It guarantees progress
* It guarantees bounded waiting.

Q5.

```
void Dekkers-Algorithm(int i)
{
  int j = !(i);
  while(1)
{ a> NON-CS();
  b> flag[i] = True;
  c> while (flag[j] == True):
    { if (turn == j)
        { flag[i] = False;
          while (turn == j);
          flag[i] = True }
    }
  d> CS
  e> flag[i] = false;
    turn = j;
}.
```

* Deckers algorithm is a variation of petrison's algorithm and is a correct solution.

Analyse the solution.

Q6 Process P_1 and P_2 hold/use Critical flag in the following routine to achieve M/E. Assume that Critical-flag is initialised to false in main program.

```
get-exclusive-access()
{  if (Critical-flag == False)
    { Critical-flag = True;
      Critical-region();
      Critical flag = False; }
}
```

Consider the following Statements

I. It is possible for both P_1 and P_2 to access Critical region Concurrently.

II This may lead to a deadlock

Ans I. is true and II is false..

Q7. Two processes P_1 and P_2 need to access a critical section of the code. Consider the following synchronization construct used by the process.

```
/P1/
while (true)
{ wants1 = true;
while (wants2 == true);
/CS/
wants2 = false; }
/Remainder Section/
```

```
/P2/
while (true)
{ wants2 = true;
while (wants1 == true);
/CS/
wants2 = false; }
/Remainder Section/
```

Here wants1 and wants2 are shared variables which are initialised to false. Which of the statements are true.

I. It does not ensure M/E.

II. It does not ensure bounded waiting.

III. It requires that processes enter critical section in strict alternation

IV. It does not prevent deadlock but ensures M/E.

Q8. Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both.

```
Process X
while (true)
{ var P = true;
while (varQ = true)
{ /CS/
varP = false; }
}
```

```
Process Y
while (true)
{ varQ = true;
while (varP == true).
{ /CS/
varQ = false; }
}
```

Here varP and varQ are shared variables and both initialised to false. Which of statements are true?

I. The proposed Solution prevents deadlock but fails to gurantee M/E.

II. It gurantees M/E but fails to prevent deadlock.

III. It gurantees M/E and prevents deadlock.

IV. It fails to prevent deadlock and M/E.

Synchronization Hardware :

Each processer supports some Support instructions (H/W) that are atomic <Executive non preemptively>

a> TSL

b> SWAP.

→ Busy waiting
→ Can be used at user mode as Spl. instructions
→ Hardware Category
→ Multi process Solution
→ Lock based Solution.

a> TSL : Test and Set Lock

TSL (&Lock); < Process executing TSL, returns the current values of lock and sets the value of lock always to True >

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

} Completly runs on Kernel mode (Atomic)

Solving CS problem through TSL

```
Bool lock = false;
void Process (int i)
{  while (1)
   {  a> NON-CS();
      b> while (TSL(&Lock) == True);
      c> <CS>
      d> Lock = F; }
   }
```

* This solution gurantee mutual exclusion.
* Does not gurantee bounded wait.

b> **SWAP Instruction <ATOMIC>**

```
SWAP (Bool *a, Bool *b)
{  Bool t;
   t = *a;
   *a = *b;
   *b = t; }
```

* Gurantees mutual exclusion
* Gurantees progress
* Gurantees bounded waiting.

User process :

```
Bool lock = False;
void Process (int i)
{ Bool key = True;
   while (1)
  { a>. NON-CS();
    b>. while (key == True)
        SWAP (&lock, &Key);
    c> CS
    d> Lock = False; }
  }
```

Q9. The enter-CS(); and leave-CS() functions to implement Critical Section of a process and are realised using test-and-set instructions as follows:

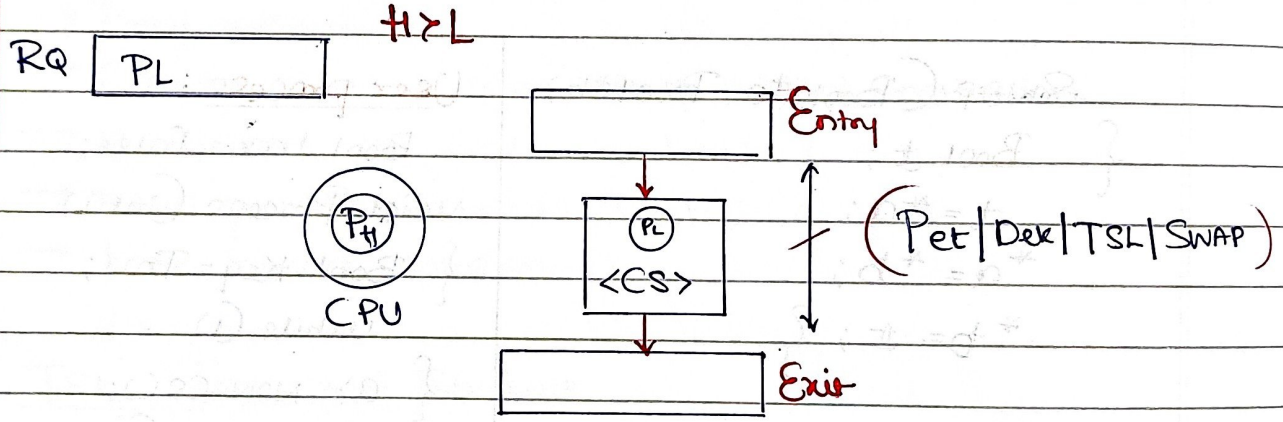
```
void enter_CS(x)
{ while (test_and_set(x));
void leave_CS(x)
  { x = 0; }
```

In the above solution, x is a memory location associated with the CS and is initialised to 0. Now consider:

I. The above solution to CS problem is deadlock free.
II. Solution is starvation free.
III. The process enters CS in FIFO order
IV. More than one process can enter CS at same time

Priority Inversion Problem

All correct busy waiting software solutions suffers from priority inversion problem. This problem happens on pre emptive priority based scheduling



* If P_H is not interested in CS then there is no problem
* If P_H is also interested in CS then it results in "Deadlock"

#Imp

Q10. Fetch-And-Add (X,i) is a atomic Read-modify-Write instruction that reads the value of memory location X, increments it by the value i and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialised to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available.

```
AcquireLock(L)
{ while (Fetch-And-Add(L,1)
    L=1; }
ReleaseLock(L)
  { L=0 }
```

This implementation:

I. fails as L can overflow

II. fails as L can take on a non-zero value when lock is actually available

III. works correctly but may starve some process.

IV. works correctly without Starvation.

Analysis:

RQ. $P_1 P_2 P_3$

ϕ X 2 Lock

P_1.
Cs

```
int lock=0

Entry  while (Fetch-And-Add(&lock,1))
         L=1;
       <CS>
Exit   L=0;
```

* If MSQ asked, I. also correct in rare cases

I:

t_1 : P_1 : F_A_A : → 0

t_2 : P_2 : F_A_A : → 01

II:

t_1 : P_1 : f_A_A : 0 : <CS> ... P_1

t_2 : P_2 : F_A_A : 1. P_2 <L=1>

t_3 : P_1 : <CS>

t_4 : P_2 : L=1, FAA=1

Blocking Mechanisms / Non busy waiting

- to avoid wastage of CPU time in the form of loops.
- all blocking mechanisms are implemented using "if-else-then".

There are three mechanisms: 1. Sleep-wakeup
2. Semaphores
3. Monitors.

1. Sleep() and Wakeup():

- Blocking construct
- Multi process solution
- Are OS primitives
- When process executes Sleep() - gets blocked till some other process wakes it up (wakeup())

Producer - Consumer using Sleep() and Wakeup()

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

void Producer(void)
    int itemp, in=0;
    while(1)
    { a>. itemp = produce-item();
      b>. if (count == N) Sleep();
      c>. Buffer[N] = 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); }
}
```

* When producer and consumer both try to update the value of count, leads to "inconsistency"

Two problems with this implementation

i. Inconsistency

ii. Deadlock.

Semaphores

* Blocking construct
* OS based <is an OS resource (S/W)>.
* General purpose utility
  { - <CS>
    - Requirements of classical IPC problems
    - Concurrency mechanisms }.
* It was proposed by E. Dijkstra
* Semaphore is implemented as an abstract data type.

Definition: Semaphore is a variable <ADT: SEM> that takes only integer values.

Operations
- down() / wait() / P()
- up() / Signal() / V()

Types.

1. Binary (mutex) <0, 1>
2. Counting (General) <-∞ to ∞>

Counting Semaphore <-∞ to ∞)

```
Typedef
    Struct
{   int values;
    QueueType L;
} CSME;
```

List of PCBs of those processes that get blocked while performing down operation unsuccessfully.

UM:

```
CSEM S;
S.value = 1;
(S=1);
DOWN(S);
<Si>
```

Next Statement.

Km:

```
DOWN(CSEM S)
{
 1. S.value = value - 1;
 2. if (S.value < 0) // unsucess
    { put this process PCB
      in the S.L(Q) and block
      the process (sleep); }
    else
      return; } // sucess.
```

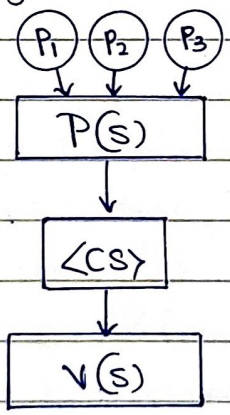
eg: CSEM S;
S = 5;

How many DOWN operations can be carried out by process sucessfully?

S = 4 3 2 1 0 -1.
1 1 1 1 1 | 1 1.

Sucessful down opn's. — after this process get blocked

* Positive value of semaphore indicates that those many down operations can be carried out sucessfully.
* After carrying out a series of down operations, if the value of semaphore becomes negative, then the magnitude of the negative value indicates the number of blocked processes.



CSEM S;
S = 1;

```
UP(CSEM S)
{
 1. S.value = S.value + 1;
 2. if (S.value < 0)
    { Select a process from
      S.L(Q) and wakeup(); }
    else
      return; }
```

Block Queue.
P2 P3.

S = 1; 0 -1 -2 -1 0 1.
blocked (down) (up)

* If the value of S is less than or equal to '0' then there are blocked processes in queue.

Q1. CSEM S=8;

Operations : 10P, 1V, 15P, 2V, 6P, 3V What is current value?
-2 -1 -16 -14 -20 -17.

Current value of S=17.

Q2. CSEM S= ?

Operations : 12P, 4V, 6P, 3V, 8P, 1V, Final value is -6.

What should be initial value of S?

Initial value of S should be 12.

Q3. Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12 V(S) operations are issued in some order. The largest initial value of S for which atleast one P(S) operation will remain blocked 7

Binary Semaphore <0,1>

UM:

```
Typedef
    Struct
{ enum value (0,1);
  QueueType L;
} BSEM;
```

List of PCBs that gets blocked while performing down operation unsuccessfully.

```
BSEM S;
S=1;
DOWN(S);
<Next Strat>
```

```
DOWN (BSEM S)
{ if (S.value == 1)
  { S.value = 0; // Sucess
    return; }
else
  { put this process (PCB)
    in S.L(Q) and
    block it (sleep();); }
}
```

* In binary semaphore we dont get to know the no. of blocked processes because no negative numbers after down operations.

```
UP(BSEM S)
{
  if (S.L(Q) is NotEmpty)
  {
    Select a process from
    S.L(Q) and wakeup(); }

  else
      S.value = 1; }.
```

Cases of Binary Semaphores

1.
```
BSEM S;
S=1;
P(S);
S=0
```
Status = Sucess.

2.
```
BSEM S;
S=0;
P(S);
S=0
```
Status = Unsuccessful.

3.
```
BSEM S;
S=0;
V(S);
```
S → 0 : if 'Q' not empty
S → 1 : if 'Q' empty.
Status = Sucess.

4.
```
BSEM S;
S=1;
V(S);
S=1
```
Status = Sucess.

5.
```
BSEM S;
S=0;
V(S); // First operation.
S=1.
```
Status = Sucessful.

Note:

As long as there is a process in "CS" & processes in block 'Q' then value of BSEM never be '0'.

Q4 BSEM S=1;

Operations : 10P, 2V, 18P, 3V, 4P, 5V;

(i) Final value of S?

(ii) Size of 'Q'[L] ?

S=X 0

10 P, 2V, 18V, 3V, 4P, 5V;

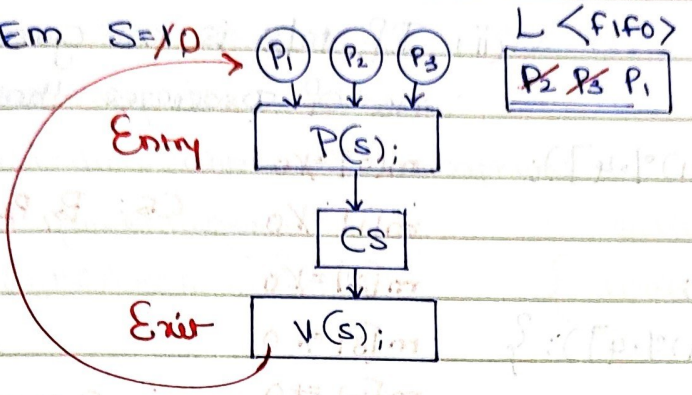
L → 9 7 2 5 2 2 2 6 21

Final value of S=0

and 21 blocked processes.

Both binary and counting semaphore can be used to solve the problem of CS.

eg: BSEM S=1/0



L <fifo>
~~P2~~ ~~P3~~ P1

1) Mutual Exclusion ✓
2) Progress ✓
3) Bounded wait ✓

Q1. BSEM S=1, T=1;

```
Pr(i)                    Pr(j)
{ while(1)               { while(1)
  { P(S);                  { P(T);
    P(T);  <- To prevent deadlock
    <CS>                     P(S);
    V(T);                    <CS>
    V(S); }                  V(S);
  }                          V(T); }
                           }
```

Does it gurantee ME? and deadlock?

Yes it gurantees ME but it is not deadlock free.

To prevent it from deadlock Swap down operations in one process. Both processes down opn. should be same.

Q2. BSEM S=1, T=0

```
Pr(i)                    Pr(j)
{ while(1)               { while(1)
  { P(T);                  { P(S);
    Print('1');              Print('0');
    Print('1');              Print('0');
    V(S); }                  V(T); }
}                        }
```

What will be output?

00110011....

Does it gurantee
1. ME ✓
2. Progress ✗
3. Bounded wait ✓

This is a implementation of "strict alternation" using semaphores

Q3. BSEM m[0...4] = {1};

```
Pr(i) i=0,4
{
  while(1)
  { P(m[i]);
    P(m[(i+1)%4]);
    <CS>
    V(m[i]);
    V(m[(i+1)%4]); }
}
```

(i) Does it guarantee M/E?

No

(ii) If M/E is not guaranteed then max no. of processes that can be in CS?

m[0] = X0
m[1] = X0
m[2] = X0
m[3] = X0
m[4] = X0

CS: P0, P2.

P1, P3, P4 } blocked

∴ 2 processes in CS

* Deadlock can occur.

Q4. BSEM = 1, T=0, Z=0

```
Pr(i)
{ while(1)
  { P(S);
    Print(*);
    V(T);
    V(Z); }
}
```

```
Pr(j)
{ P(T);
  V(S); }
```

```
Pr(k)
{ P(Z);
  V(S); }
```

What is min and max no of * that get printed?

Case 1: Pri, Prj, Prk, Pri blked

I: <* *> : minimum → 2

Case II: Pri, Prj, Pri, Prk, Pri

II: <* * *> maximum → 3.

Variation: BSEM S=1, T=0, Z=0;

```
Pri
{ while(1)
  { P(S);
    Print('*');
    V(T); }
}
```

```
Prj
{ P(T);
  V(Z); }
```

```
Prk
{ P(Z);
  V(S); }
```

In this variation the min and max no of Stars to be printed = 2;

Q5. Consider the following processes T_1, T_2, T_3 executing on a single processor, synchronized using three binary semaphore variables S_1, S_2 and S_3, operated upon using standard wait() and signal(). The threads can be context switched in any order and at any time. Which.

Which initialization of semaphores would print sequences

BCABCABCA... ?

T_1

```
while (true)
{ wait (S3);
  Print ("C");
  Signal (S2); }
```

T_2

```
while (true)
{ wait (S1);
  Print ("B");
  Signal (S3); }
```

T_3

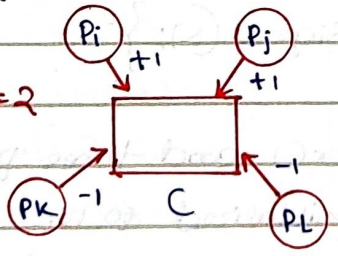
```
while (true)
{ wait (S2);
  Print ("A");
  Signal (S1); }
```

Ans: $S_1 = 1$, $S_2 = 0$, $S_3 = 0$

$T_2 \rightarrow T_1 \rightarrow T_3$

HW

Q6 Consider a counting semaphore initialized to 2, there are 4 concurrent processes P_i, P_j, P_k and P_L, the processes P_i and P_j desire to increment the current value of variable C by 1 whereas P_K and P_L desire to decrement the value of C by 1 All processes perform their update on C under semaphore control. What can be the min and max value of C after all processes finish their update?



Assuming initial value of C != 0

```
Process (x)
{ P(S);
  <C = C ± 1>
  V(S); }
```

x = i, j, k, l

+1 −1

Q7. Consider three processes using four binary semaphores a, b, c, d in the order shown below.
Which is sequence is a deadlock free sequence?

X : P(b); P(a); P(c);
Y : P(b); P(c); P(d);
Z : P(a); P(c); P(d);

a = b = c = d = 1. } Initialized.

Q8. Each of a set of n processes executes the following code using two semaphores a and b initialized to 1 and 0 resp. Assume that count is a shared variable initialized to 0 and not used in code section P.

```
<Code Section P>
    wait(a);
    count = count + 1;
    if (count == n) signal(b);
    signal(a); wait(b); signal(b);
<Code Section Q>
```

What does code achieve?
It ensures that no process executes code section Q before every process has finished code section P.

HW

Q9. Consider two functions Incr() and Decr() GATE 2023

```
Incr()                    Decr()
{ wait(s);                { wait(s);
  x = x + 1;   } 5          x = x - 1;   } 3.
  signal(s); }              signal(s); }
```

Five processes invoke Incr() and three processes invoke Decr()
x is a shared variable initialized to 10. (10+5-3) = 12

I_1 : S value is 1 (Bin. Sem.) max/min = 12
I_2 : S value is 2 (counting. Sem.)

Let v_1 and v_2 be min possible values of implementation of I_1 and I_2 then choose the values of x for v_1 and v_2.

Q1. Which of the following may result in a process getting blocked?

a) DOWN
b) UP
c) System-call
d) Scheduler dispatch.

Classical IPC Problems

1. Producer Consumer Problem:

```
#define N 100
int Buffer[N];
CSEM Empty = N;
<No of empty slots>
CSEM full = 0;
<No. of full slots (data-items)>
BSEM mutex = 1;
<used b/w P and C to
ensure M/E on buffer>
```

```
void Producer (void)
{ int temp, in = 0;
  while (1)
  { a). itemp = produce-itemC();
Entry { b). DOWN (Empty);
      { c). DOWN (mutex);
    < d). Buffer[in] = itemp;
      e). in = (in+1) % N; >
Exit { f). UP (mutex);
     { g). UP (full);
```

```
void Consumer (void)
{ int itemC, out = 0;
  while (1)
  { a). DOWN (full);
    b). DOWN (mutex);
    c). itemC = Buffer[out];
    d). out = (out+1) % N;
    e). UP (mutex);
    f). UP (empty);
    g). Process-item (itemC); }
  }
```

Q1. Consider the following solution to the P-C sync. problem. Shared buffer size is N. Three semaphores empty, full and mutex are declared with respective initial values of 0, N and 1. Semaphore empty denotes the no. of available slots in the buffer, for Consumer to read from. Semaphore full denotes no of available slots in the buffer, for producer to write to. The placeholder variables denoted by P, Q, R, and S in code below can be assigned either empty or full.

```
Producer:                          Consumer:
  do{                                do{
     wait(P);                           wait(R);
     wait(mutex);                       wait(mutex);
    // add item - buffer               // Consume item - buffer
     signal(mutex);                     signal(mutex);
     signal(Q); }                       signal(S); }
  while(1);                          while(1);
```

Which of the assignment of P, Q, R, and S yeild correct solution?

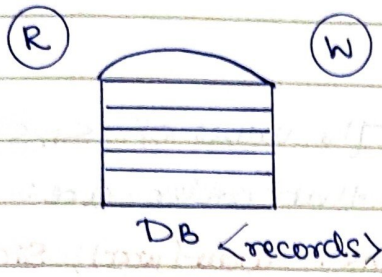
Ans P: full, Q: empty, R: empty, S: full

Q2. Consider the procedure below for P-C problem use semaphores:

```
Semaphore n=0;
Semaphore s=1;
void Producer()                    void Consumer()
{  while(true)                     {  while(true)
   {  produce();                      {
      SemWait(s);                        SemWait(s);
      addToBuffer();                     SemWait(n);
      SemSignal(s);                   removeFromBuffer();
      SemSignal(n); }                    SemSignal(s);
   }                                     consume(); }
}                                  }
```

Ans. Deadlock occurs when/if the Consumer succeeds in acquiring Semaphores when buffer is empty.

2. Reader Writer Problem.



BSEM mutex=1;
{
P(mutex)
<DB>
V(mutex); }

First Reader Writer Problem.

t_1 : r_1 and w_1 : Anyone
t_2 : r_1 ; ✓ w_1 (wait)
t_3 : r_2 ; ✓
t_4 : r_3 ; ✓

r_1 r_2 r_3

t_1 : r_1 and w_1 : Anyone
t_2 : w_1 ; ✓ r_1 (wait)
t_3 : w_2 ; ✗ (wait)

w_1

Problem: Starvation to writers.

Implementation of First R-W using Semaphore

```
int rc=0;
BSEM mutex=1;
<used by Rs to update>
BSEM db=1;
<used by Rs and Ws to
    access <DB> as CS>

void writer (void)
{
  while (1)
  { a>. DOWN(db);
    b>. <DB-write>
    c>. UP (db); }
}
```

```
void Reader (void)
{ while (1)
  {
    a>. DOWN(mutex);
    b>. rc = rc+1;
    c>. if (rc==1) DOWN(db);
    d>. UP (mutex);
    e>. <DB-Read>
    f>. DOWN(mutex);
    g>. rc = rc-1;
    h>. if (rc==0) up(db);
    i>. UP(mutex); }
  }
```

Q3. Synchronization in the classical readers and writers problem can be achieved through use of semaphores. In the following incomplete code for readers-writers problem, two binary semaphores

mutex and wrt are used to obtain sync.

```
Wait (wrt)
// waiting is performed
Signal (wrt)
 Wait (mutex)
readCount = readCount +1
if readCount == 1, then "S1"
  "S2"
// reading is performed
  "S3"
 readCount = readCount -1
if readCount = 0 then "S4"
 Signal (mutex)
```

The values of S1, S2, S3, S4 in that order are:

Ans: Wait(wrt), Signal(mutex), Wait(mutex), Signal(wrt).

3. Dining Philosophers Problem

'N' - Philosophers ($N \geq 2$)

If all become hungry
Causes "Deadlock"

```
#define N 5
 void Philosopher (int i)
{  while (1)
   {  a> think (i);
      b> take-fork (i);
      c> take-fork ((i+1)%N);
      d> eat (i);
      e> put-fork (i);
      f. put fork ((i+1)%N); }
}
```

Without deadlock, for N=5 what is the max no. of philosophers that can be eating?

P_0 : f_0, f_1 ✓
P_1 : ×
P_2 : f_2, f_3 ✓
P_3 : f_4
P_4 : f_4, f_0 ×

Ans: 2

Prevent/Avoid Deadlock in Dining Philosophers

Semaphore

→ Put semaphore control on taking and releasing the forks.

Non Semaphore based

→ Out of 'N' philosophers,
Let (N-1) → L-R
and 1 → R-L

→ Even numbered : L-R
Odd numbered : R-L

Q4. A solution to the dining philosophers problem which avoids deadlock is:

Ensure that one particular philosopher picks up the left fork before right fork, and all other philosophers pick up right before left fork.

Q5. Let m[0...4] be mutexes and $P_0 - P_4$ be processes. Suppose each process P[i] executes the following

```
wait(m[i]); wait(m[i+1]%4)
<CS>
release(m[i]; release(m[i-1]%4)
```

Which situation could occur?

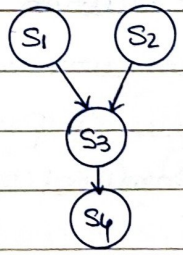
Ans: Deadlock.

Concurrency vs Sequentiality

```
{ S1 : a = b+c;
  S2 : d = e*f;
  S3 : k = a+d;
  S4 : l = k*10; }
```

I_1 : Load R_1, b
I_2 : Load R_2, c
I_3 : Add R_1 R_2
I_4 : Store a, R_1

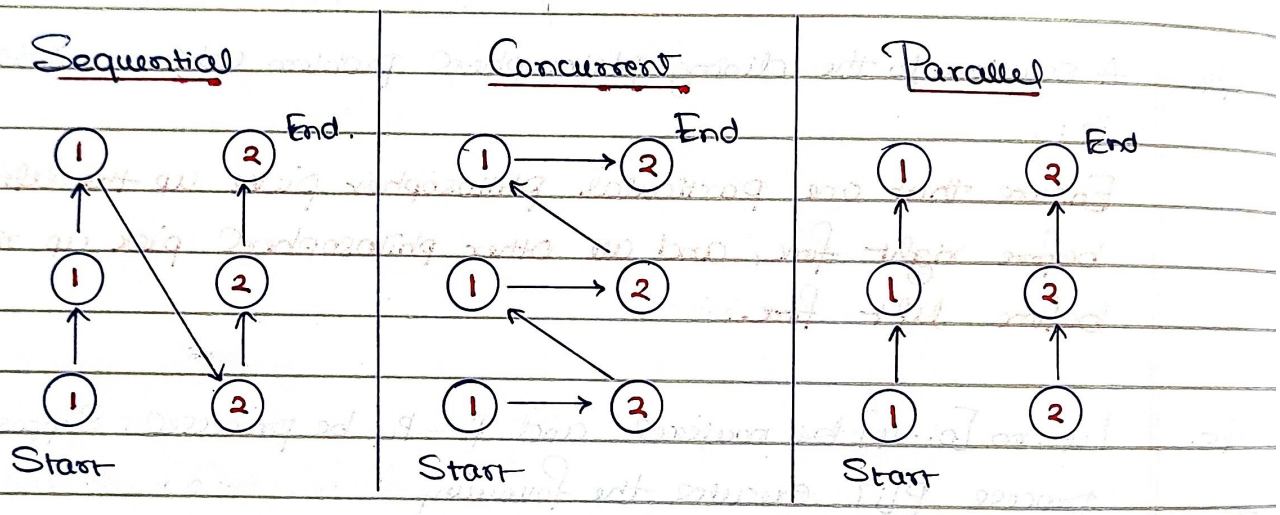
* Only S1 and S2 can perform concurrently (no dependency)



Precedence Graph (directed)

Si : High level language statement.

Concurrency	Parallelism
* A system is said to be concurrent if it can support two or more actions in progress at the same time	A system is said to be parallel if it can support two or more actions executing simultaneously
* Concurrency is about dealing with lots of things at once.	Parallelism is about doing lots of things at once.



An Analogy:

Concurrent : Two queues and one coffee machine

Parallel : Two queues and two coffee machines

Note:

* In Concurrency process will execute with preemptions.
* Parallelism is possible with multiple CPUs/Cores.
* Concurrency is about structure, parallelism is about execution.
* Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

* The modern world parallel it has:
  1. Multi Cores
  2. Networks.
  3. Cloud of CPUs
  4. Loads of users.
* Concurrency makes parallelism easy.

Types of Concurrency

1. Pseudo (one CPU) → Interleaved execution among processes/applications.
2. Real/Physical (Parallelism) → Multiple processing units

Concurrency Conditions

→ Si = a = b + c; Sj = d = e * f;

* Output of one statement should not serve as input to another

S — R(S) : Read set
  — W(S) : Write set

eg: (I) S : a = b + c
R(S) → b, c; W(S) → a.

(II) S : a++ = ++b * --c;
R(S) and W(S) → a, b, c.

(III) S : scanf("%d", &x);
R(S) → ϕ
W(S) → x

(IV) printf(x);
R(S) → x
W(S) → ϕ

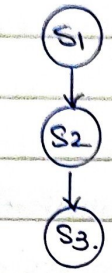
Bernstein's Concurrency Conditions:

I. $R(S_i) \cap W(S_j) = \phi$
II. $W(S_i) \cap R(S_j) = \phi$
III. $W(S_i) \cap W(S_j) = \phi$
IV. $R(S_i) \cap R(S_j) =$ may/may not ϕ

Concurrency mechanisms

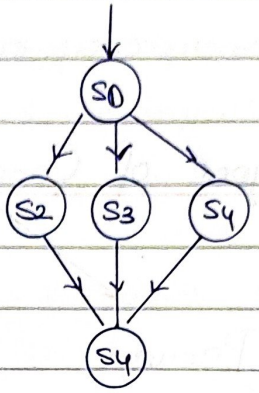
1. Parbegin - Parend / Cobegin - Coend :

```
{ begin
    S1;
    S2;
    S3;
  end }
```



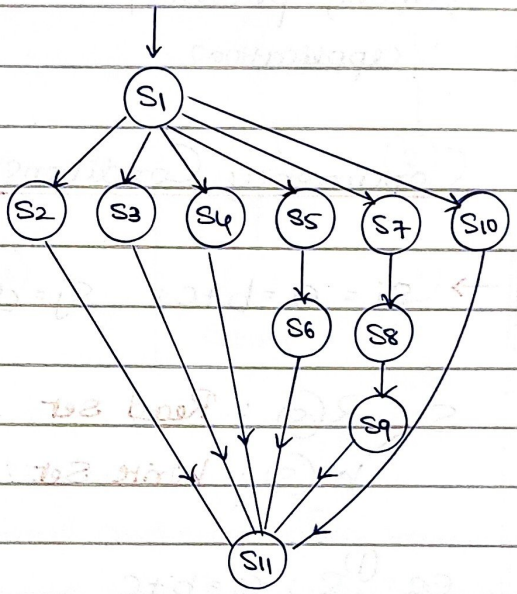
Sequential

```
S0;
Parbegin
    S1;
    S2;
    S3;
Parend
    S4;
```

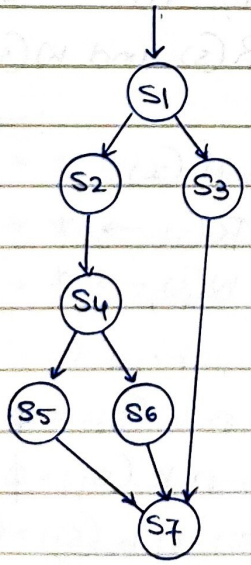


eg:

```
S1;
Parbegin
    S2; S3;
    S4;
    begin S5; S6; end
    begin S7; S8; S9; end
    S10;
Parend
    S11;
```



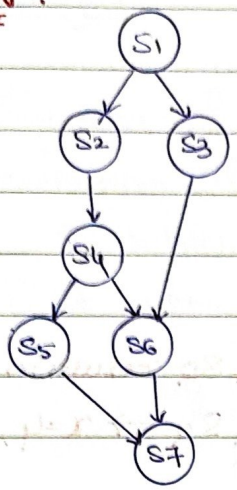
eg:




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

How many times parallel construct to be used?

HW :

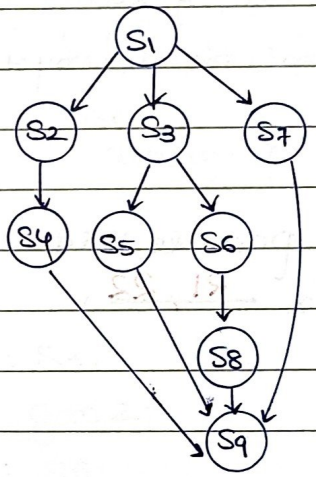


S1; BSEM a,b,c,d,e,f,g = {0}

```
Cobegin
  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
Coend
```

Q1. Draw the precedence graph for the Concurrent Program.

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



Q2.

```
void P(void)        void Q(void)        void main()
{ A;                { D;                { Parbegin
  B;                  E; }                  P(); Q();
  C; }                                    Parend; }
```

Valid output Sequences: ABCDE
DEABC, ADBEC.

Q3. int x=0; y=0;

Cobegin

begin

S1: x=1;

S2: y=y+x;

end

begin

S3: y=2;

S4: x=x+3;

end

Coend.

Final values of x and y

I. x=1, y=2

II. x=1, y=3

III. x=4, y=6

Statements S3, S4, S1, S2 results in x=1, y=3.

Statements S1, S3, S4, S2 → x=4, y=6

Q4. int x=0, y=20;

Bsem mx=1; my=1;

Final possible values of x {21, 22}

Cobegin

begin

P(mx);

x=x+1;

V(mx);

end;

begin

P(mx);

x=y+1;

V(mx);

end

Coend

HW

Q5. integer B=2;

P1()

{ C=B-1;

B=2*C; }

P2()

{ D=2*B;

B=D-1; }

main()

{ Parbegin

P1()

P2() }

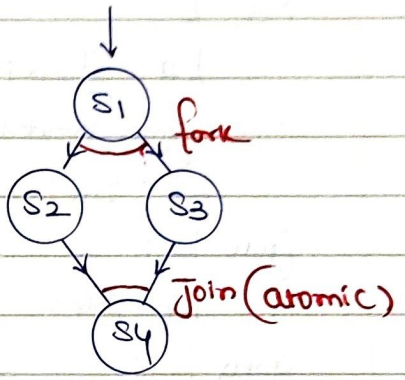
Parend.

The no. of distinct values of B are _____

2. Fork join

Syntax of fork : fork L;

→ Execution of fork results in starting 2 computation concurrently
I. Which is immediately after fork
II. Which is at Label 'L'.

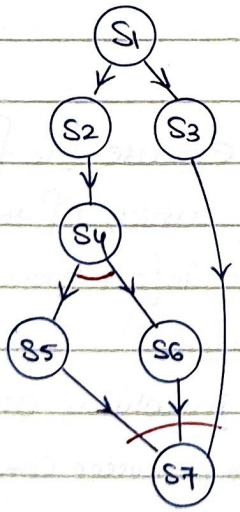


eg:

```
integer count=2;
S1;
fork L;
S2;
goto X;
L: S3;
X: Join (count);
S4;
```

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

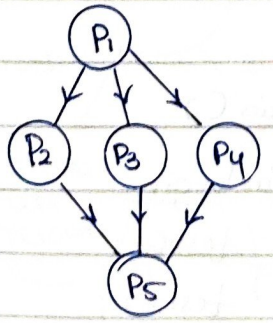
eg:




```
int count=3;
S1;
fork L;
S2; S4;
fork X;
S5;
goto Z;
```

```
L: S3;
   goto Z;
X: S6;
Z: Join (count)
   S7;
```

eg

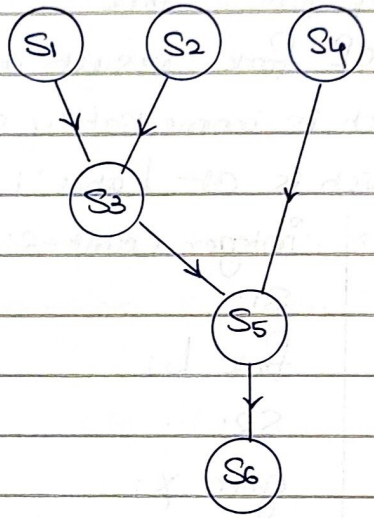



```
P1;
fork L;
P2;
L: fork X
   P3;
   goto Z;
```

```
X: P4
Z: Join (count);
   P5;
```

eg. N=2, M=2 Draw the precedence graph.

```
fork L3, L4 :
S1 ;
    L1 : Join N
S3; &
    L2 : Join M
S5 : goto next;
    L3 : S2;
goto L1;
    L4 : S4;
goto L2;
next : S6;
```



Q1. Consider the following pseudo code where S is a semaphore initialized to five and counter is a shared variable initialized to 0 in line 1. Assume that increment operation in line #7 is not atomic.

```
1  int Counter = 0
   Semaphore S = init(5);
   void parop(void)
   {  wait(S);
      wait(S);
7     Counter ++;
      Signal(S);
      Signal(S);
   }
```

If five processes execute the function parop concurrently, which of the following program behaviour is/are possible?

~~I.~~ there is a deadlock involving all processes.
~~II.~~ Counter = 5 after all processes comp- parop
~~III.~~ counter = 1 " " "
IV. counter = 0 " " "

Q2.

```
int count=0; void test()
        {int i, n=5;
         for(int j=1; j<=n; ++i)
             count=count+1; }
```

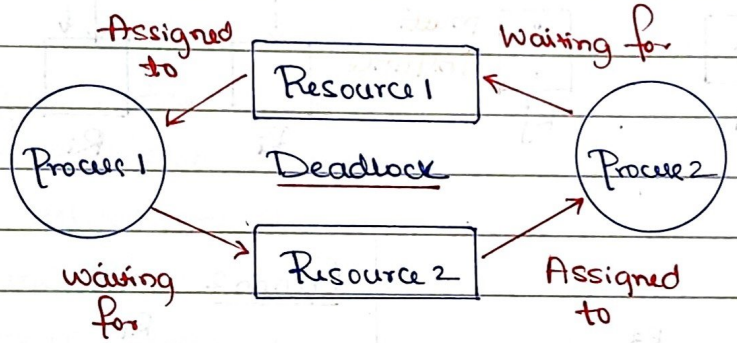
```
main()
{ Parbegin
    test();
    test();
  Parend }
```

Date 16/2/23
Page

Deadlock

Deadlock: Two or more processes are said to be in deadlock if they wait for the happening of an event which will never happen.

Consequences:
* Throughput/Efficiency drops
* Ineffective utilization of resources

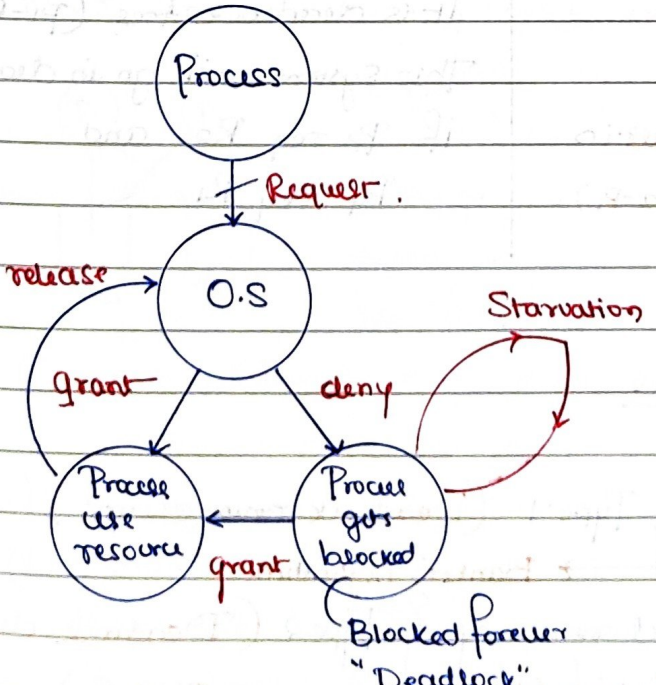


System Model:

→ n: no of processes $\langle P_1 \dots P_n \rangle$

→ 'm': no. of resources $\langle R_1 \dots R_m \rangle$

- Single instance
- Multi Instance

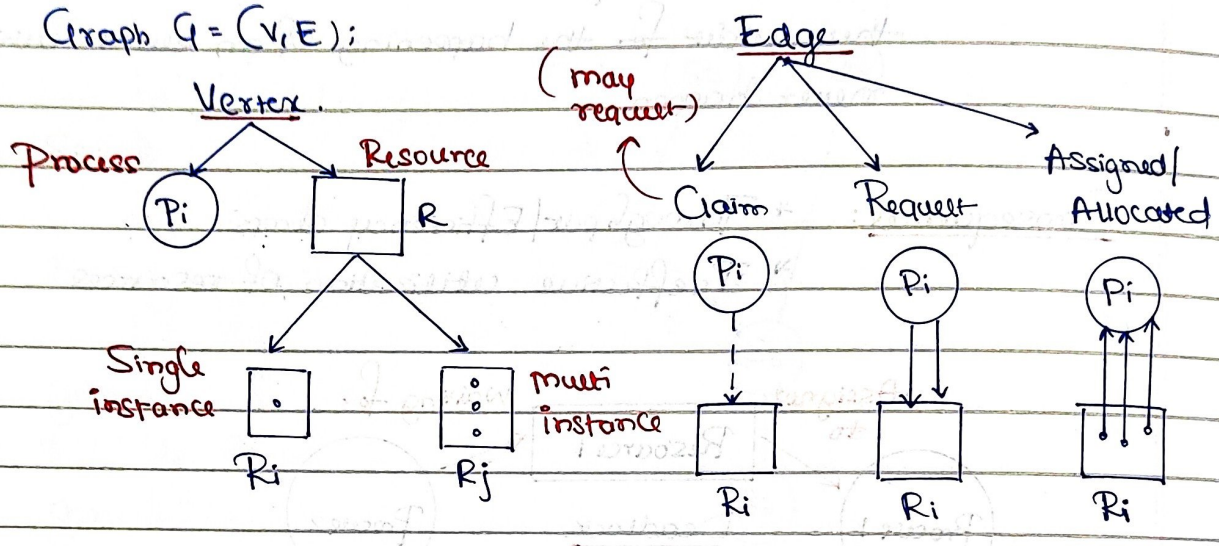


Necessary Conditions for Deadlock:

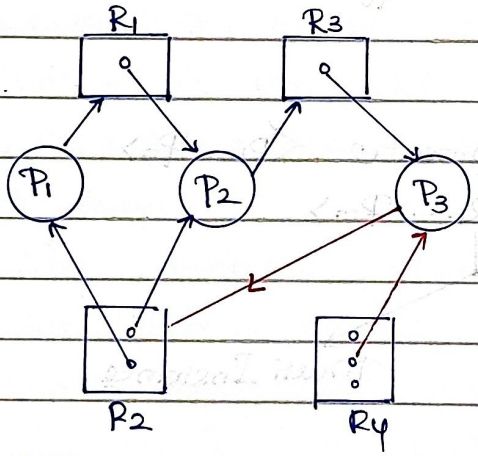
1. Mutual Exclusion
   → Shared resource <CS>
2. Hold and wait
   Every H&W is not deadlock but every deadlock will have H&W.
3. No pre-emption of resources
4. Circular wait

Resource Allocation Graph (RAG)

Graph G = (V, E);

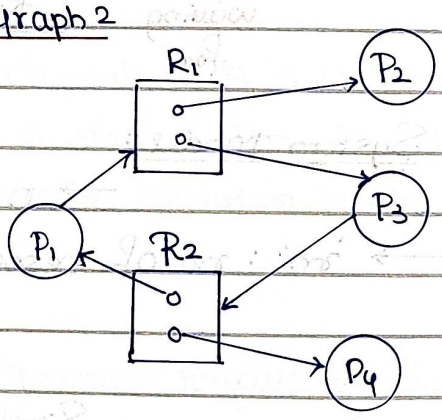


Graph 1.



It is deadlock free.
After changes it will result in deadlock (P_1-R_1-P_2-R_3-P_3-R_2-P_1) cycle.

Graph 2



It is deadlock free (P_1-R_1-P_3-R_2-P_1)
This system will go in deadlock if P_3 req R_2 and P_4 req R_1

Deadlock Handling Strategies

1. Deadlock Prevention } Type 1 (Deadlock never occurs)
2. Deadlock avoidance → Banker's Algorithm.
3. Deadlock detection and recovery } Type 2 (Deadlock definitely occurs)
4. Deadlock Ignorance → Ostrich Algo

I. Deadlock Ignorance (Ostrich Algorithm)

* Pretend there is no problem (Ignorant Strategy of ostrich)
* Reasonable if : (i) deadlock occur very rarely
  (ii) cost of prevention is high.
* UNIX and Windows take this approach
* It is a trade-off between convinience and correctness.

II. Deadlock Prevention

Deadlock Prevention can be done by dissatisfying or negating the necessary conditions of deadlock.

a) mutual exclusion :→ Since every multi programmed Operating System has atleast one shared resource ∴ M/E is non dissatisfyable.

b) Hold and Wait → Hold or Wait implementation

(i) Process must request and be allocated all resources prior to its start.

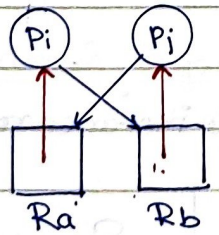
Drawbacks: Causes/leads to Starvation, inefficiency

(ii) Process must release all resources before making a fresh/new request.

Drawback: Suffer from Starvation

(iii)

c) No preemption: → pre emption of resources from processes.



forced (Running process should not block)

Self (allows other processes to Complete first)

Drawbacks: Causes Starvation.

d) Circular-wait :→ is prevented by a total order relation among all processes and resources.

(i) Assign unique numbers/Ids to each resource

(ii) Never allow a process to request a lower numbered resource than the last one allocated.

Drawback: Starvation possible

classmate
Date 19/2/23
Page

Note :

(1.) If the R.A.G. has resources of only multi instance type, then cycle is only a necessary condition

(2.) If R.A.G has resource of both Single and multi instance, then presence of cycle is only a necessary condition for deadlock.

(3.) If the R.A.G has resources of only Single instance type, then cycle in a necessary and Sufficient condition for deadlock.

III. Deadlock Avoidance

1. Single instance resource
<Resource allocation Graph algorithm>

2. Multi instance of Resource (SI+M.I)
Banker's Algorithm
(i) Safety algorithm
(ii) Resource request algorithm

Note: Both the algorithms are based on Apriori Knowledge/ information.

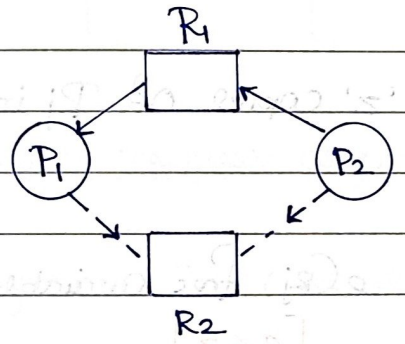
1. Resource-Allocation Graph Algorithm <Single instance resource>

* Resources are claimed a priori in the system.
* If process Pi starts executing then all claim edges must appear in R.A.G.
* If Pi's request resource Rj, then the request is granted only if, converting the request edge to ~~Claim~~ assigned edge does not lead to a cycle in R.A.G Otherwise

System State
Safe → No deadlock
Unsafe → Danger of deadlock

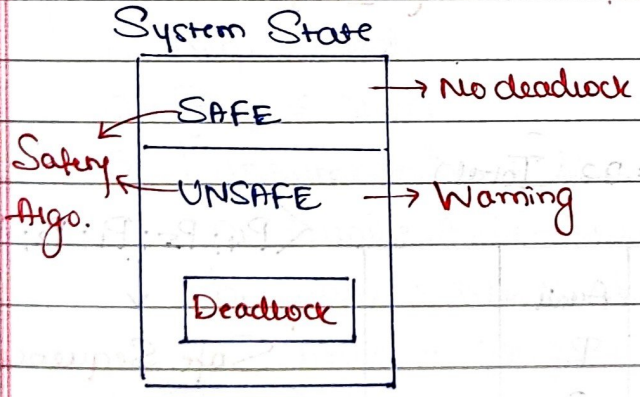
the process is blocked
<No cycle implies Safe State and formation of cycle implies Unsafe state>

* The basic objective of R.A.G algorithm (S.I) is to always operate the system is in SAFE State.
* System is said to be safe, if the conversion of request edge to assigned edge does not lead to cycle in R.A.G. Otherwise if it is leading cycle then it is unsafe mode.



→ Current State: Safe (no cycle)

→ What if P2 also request R2?



Banker's Algorithm (m. Instances)

(i) Safety algorithm (ii) Recource request algorithm

Data Structures (System state)

1. n : no of processes (P1 ... Pn)
2. m : no. of resources (R1 ... Rm)
3. maximum [1...n, 1...m]nxm :

max [i,j] = k

Pi ⟶ k(Rj)

Process Pi demands (apriori) 'k' copies of Rj'.

4. Allocation $[1..n, 1..m]_{n \times m}$: $Alloc[i,j] = a$
$P_i \leftarrow a(R_j)$ $[a \leq k]$

5. Need $[1..n, 1..m]_{n \times m}$: Need $[i,j] = b \rightarrow$ Need = max − allocation
$P_i \leftrightarrow b(R_j)$ $[b = k - a]$

6. Request $[1..n, 1..m]_{n \times m}$: $P_i \longrightarrow c(R_j)$ @ time 't'.
$Req[i,j] = c$. $[c \leq b]$

7. Total $[1...m]$
Total $[j]$ = 'z' → There are 'z' copies of R_j in the system.

8. Available $[1...m]$
Avail $[j]$ = 'e' There are e(R_j) free available @ time 't'.
Avail = Total − $\sum_{i=1}^{n}$ Alloc$_i$ $[e \leq z]$

Example : n = 5, m = 1, R = 22 (Total)

Pid.	Max R	Alloc R	Need R	Avail R	
P_1	10	5	5	3	
P_2	8	4	4	5	
P_3	12	5	7	8	
P_4	5	2	3	13	
P_5	6	3	3	17	

<22>

<P_4; P_5; P_1; P_2; P_3;>
↓
"Safe Sequence"
(We can have multiple safe sequences)

Need = max − Alloc

Safety Algorithm:

System is said to be "safe" if the need of all processes can be satisfied with available resources in some order, otherwise it is "unsafe."

Example 2: $\langle A, B, C \rangle = \langle 10, 5, 7 \rangle$ $\langle P_1; P_3; P_4; P_0; P_2 \rangle$

	Allocation A B C	Max A B C	Available A B C	Need A B C	
P_0	0 1 0	7 5 3	3 3 2	7 4 3	"System is Safe."
P_1	2 0 0	3 2 2	$\langle 5\ 3\ 2 \rangle$	1 2 2	
P_2	3 0 2	9 0 2	$\langle 7\ 4\ 3 \rangle$	6 0 0	
P_3	2 1 1	2 2 2	$\langle 7\ 4\ 5 \rangle$	0 1 1	
P_4	0 0 2	4 3 3	$\langle 7\ 5\ 5 \rangle$	4 3 1	
			$\langle 10\ 5\ 7 \rangle$		

(ii) Resource request algorithm

Algorithm Res-request(Pi, Reqi, Alloci, Needi, Avail)

{

1. Reqi ≤ Needi
2. Reqi ≤ Alloci
3. [Assume to have satisfied the Reqi]
   a. Avail = Avail - Reqi
   b. Needi = Needi - Reqi
   c. Alloci = Alloci + Reqi
4. Run Safety algorithm
5. If System is SAFE then grant the Reqi
   else deny the Reqi and block the process Pi; }

HW

Processes	Allocation A B C D	Max A B C D	Available A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Q1. An OS uses the banker's algorithm for deadlock avoidance when managing the allocation of three resources types x, y and z to the processes P_0, P_1 and P_2. The table given below presents the current system state. Here the allocation matrix shows the current no. of resources of each type allocated to each process and max matrix shows the maximum number of resources of each type required by each process during its execution.

	Allocation			Max		
	X	Y	Z	X	Y	Z
P_0	0	0	1	8	4	3
P_1	3	2	0	6	2	0
P_2	2	1	1	3	3	3

There are 3 units of type x, 2 units of type y and 2 units of z still available. System is currently in safe state. Consider the following independent requests for additional resources in current state.

Req 1: P_0 requests 0 units of x, 0 units of y and 2 units of z

Req 2: P_1 requests 2 units of x, 0 units of y and 0 of z.

	Max	Alloc	Need	Avail
	x y z	x y z	x y z	x y z
P_0	8 4 3	0 0 1	8 4 2	3 2 2
P_1	6 2 0	3 2 0	3 0 0	$\langle 1\ 2\ 2 \rangle$
P_2	3 3 3	2 1 1	1 2 2	$\langle 6\ 4\ 2 \rangle$
				$\langle 8\ 5\ 3 \rangle$
				$\langle 8\ 5\ 4 \rangle$

T_0: P_0 : $\rightarrow \langle 0,0,2 \rangle$

$\langle P_1, \ldots \rangle$

System is unsafe & Req 1 is not granted.

T_1: P_1 : $\langle 2,0,0 \rangle$

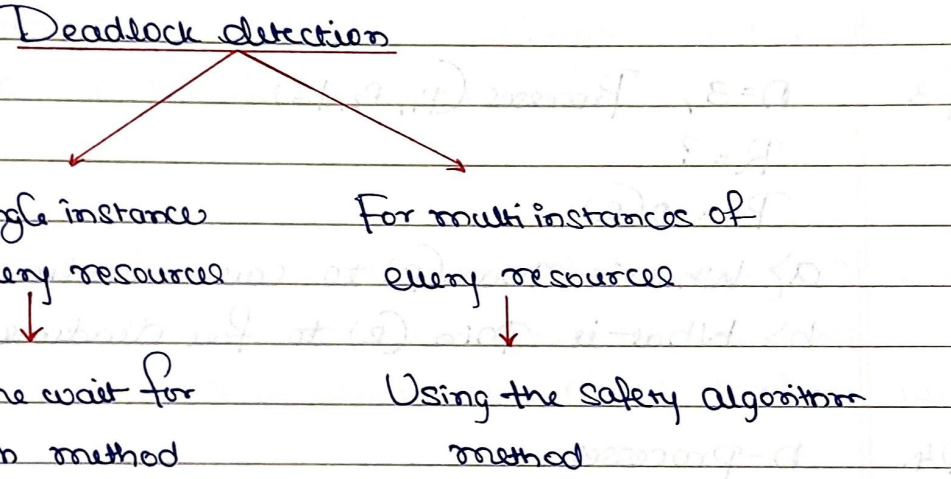
Request 2 : $\langle P_1, P_2, P_0 \rangle$

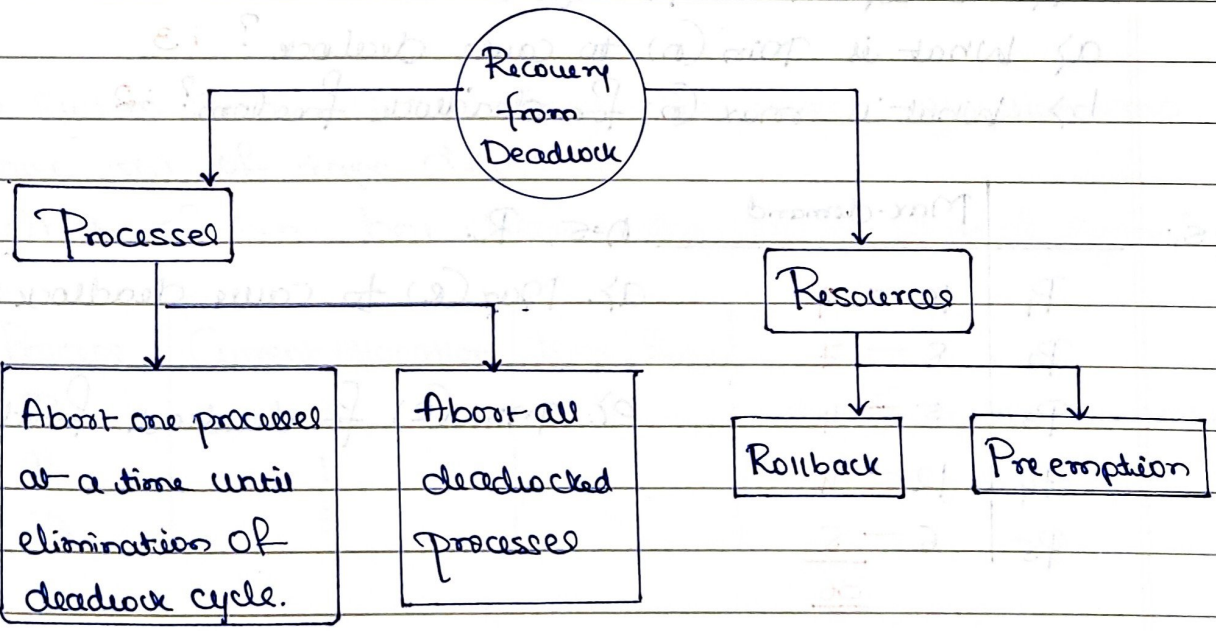
$\therefore$ System is safe ~~but~~ and request is granted

Only Request 2 is granted and Request 1 is not granted

(iv) Deadlock detection and recovery:

When to activity/activate (apply detection algo.)?
→ Under utilization of cpu
— majority of processes are blocked.





1. Prevention } Class 1
2. Avoidance }
3. Detection and Recovery } Class 2
4. Ignorance. } Class 3.

Conceptual Problems.

Q2. Consider a system having 'n' processes and a single 'R' having '6' copies. Each process need 2 copies of R to complete.

a) What is min (n) to cause deadlock? :→ 6

b) What is max (n) to deadlock freedom? :→ 5

Q3 n=3, Processes (P_1, P_2, P_3)

R = ?

$P_i = 2(R)$

a) What is Max(R) to cause deadlock? →3

b) What is Min (R) to free deadlock? →4

Q4. n-processes

R = 6 copies. P_i requires : 3(R)

a) What is Min (n) to cause deadlock? : 3

b) What is max (n) for deadlock freedom? : 2

Q5.

	Max-demand	n=5, R.
P_1	10 — 9	a). Max(R) to cause deadlock : 36
P_2	8 — 7	
P_3	5 — 4	b). Min (R) for deadlock freedom : 37
P_4	12 — 11	
P_5	6 — 5	
	36	

Q6. Which of them is not a valid deadlock prevention scheme?

Ans Never request a resource after releasing any resource

Q7. Which is not true of deadlock prevention and deadlock avoidance?

Ans In deadlock prevention the request for resources is always granted if the resulting state is safe.

Q8. An OS implements a policy that requires a process to release all resources before making a request for another resource. Select true statement.

Ans. Starvation can occur but deadlock cannot occur.

Q9. A computer has 6 tape drives, with n processes competing for them. Each process may need two drives. What is the max value of 'n' for deadlock freedom?

Ans 5

Q10 Consider a System having m resources of same type. These resources are shared by 3 processes A, B, C which have a peak demand 3, 4, 6 resp. What value of 'm' deadlock will not occur?

Ans Value should be greater than 10.

Q11. A System shares 9 tape drives. The current allocation and max req. of tape drives for:
Which of them best describes the current State of System?

Process	Current Allocation	Max. Req.
P_1	3	7
P_2	1	6
P_3	3	5

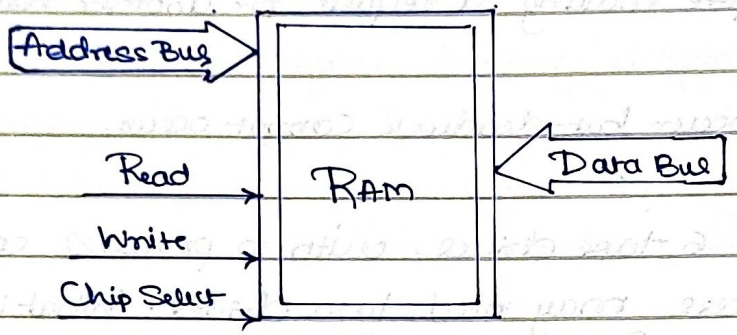
Ans: Safe, Not deadlocked

Q12 Which of following Statements is/are true w.r.t deadlocks?

Ans. a. Circular wait is a necessary condition for formation of deadlock.

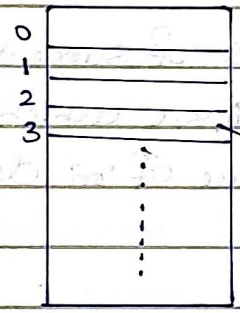
d. In the resource allocation graph of a System, if every edge is an assignment edge, then System is not in deadlock State.

Memory Management

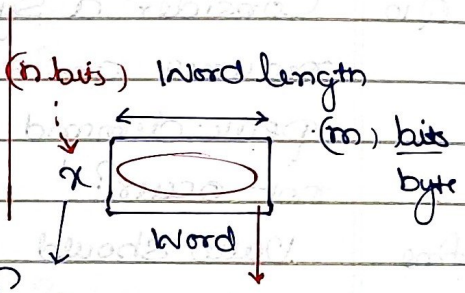


Block diagram - Primary Memory

Linear dimensional view of Memory
(Abstract view)



Words / Locations / Cells



Total no. of words
= 'N' Memory Capacity

fixed length address

You can store instructions + data

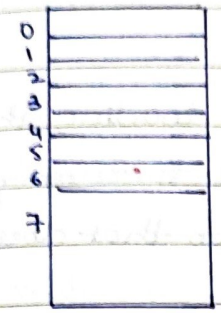
Memory: Memory is specified as number of words × width of words
($N \times m$)

n: Size of address in bit
m: word length / width of word
N: total number of words

I. **Relation b/w N & n:**

→ Binary bits: (0, 1)
→ Using 3 binary bit 8 combinations are possible
→ 'n' bits :→ 2^n combinations possible

eg:



$N = 8$ words

$n = 3$ bits

$2^3 = 8W$

$N = 2^n$ words

$n = \log_2 N$ bits

Conversions:

$2^{10} = 1024 = 1K$ words

$2^{20} = 2^{0}$ 1m. words

$2^{30} = 1G$ words

$2^{40} = 1T$ words.

$2^{50} = 1P$ words

eg. * $N = 256$

$n = \log_2 256 = 8$ bits

* $n = 6$ bits

$N = 2^6 = 64$ words.

* $n = \log_2 z$ bit

$N = z^{(\log_2 z)}$

$= z^{(\log_2 z)}$ ∴ $N = z$ Bytes

II. $N \propto n \propto m$.

1. $n = 13$ bits
$m = 8$ bits
$N = 8KW$
$= 8KB$.

2. $n = 18$ bits
$m = 16$ bits
$N = 256KW$ Word view
$= 512KB$ Byte view

3. $n = 33$ bits
$m = 64$ bits
$N = 8GW$
$= 64GB$

4. $N = 64KB$
$m = 4B$
$n = \frac{64KB}{4B} = 16K$ Words.

5. $n = 23$ bits
$m = 32$ bits
$N = 2^{23} = 2^3 \times 2^{20} = 8m$ words
$N = 8m \times 4 = 32m$ Bytes

6. $N = 256MB$
$m = 128B$
$N = \frac{256MB}{128B} = 2m$ words.

$n = 21$ bits (W)
$n = 28$ bits (B)

7. $N = 8G$ bits (Twist)
$m = 64$ bits
$N = 128m$ words
$N = 1G$ Bytes.

Loading vs Linking

Loading: Loading refers to the activity of loading the program or bringing the program from disk to memory. The component of the operating system that does this activity is called "Loader".

The two approaches of loading are: 1. Static loading
2. Dynamic loading

1. **Static loading**: The whole program is loaded in memory before starting the execution.
   * Advantage: faster execution (time efficient)
   * Drawback: possible wastage of space (ineffective utilization of memory)

2. **Dynamic loading**: The process of loading the program on demand at runtime is called Dynamic loading.
   * Advantage: no wastage of memory
   * Drawback: slow execution (time inefficient)

Linking: Linking is the process of resolving external references in the program

```
#include <stdio.h>
   :
main()
{  :
   fun();   BSA
   :
BSA
   scanf();
}
```

```
fun()
{ ...
  ...
  ... }
```

Note: Compilation always starts with 1 from first line of code

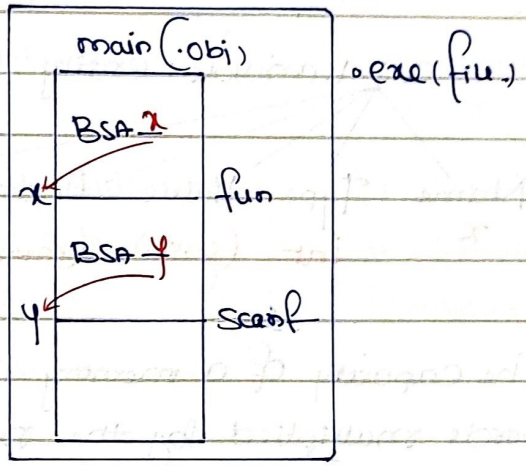
* Execution of program always start with main()

BSA: Branch and Save address

The two ways of linking are: 1. Static linking
2. Dynamic linking

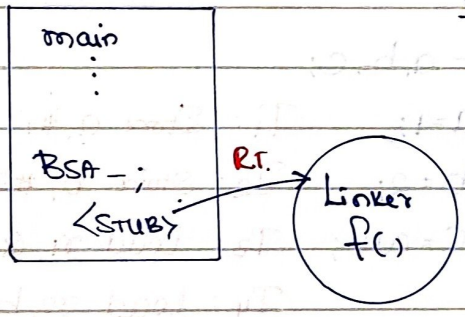
Static Linking: Linking of program before execution. Space inefficiency is the drawback of static linking.

```
main()          | fun()
{ ....          | { :
  ....          |
BSA_ fun();     |   Scanf();
     :          |     :
   }            |   }
```



Dynamic Linking: Linking at runtime is called dynamic linking. It is space efficient but time inefficient.

```
main()          | f()
{ ....          | { :
  ....          |
BSA_ f();       |   Scanf();
<STUB> :        |     BSA_
     :          |     :
   }            |   }
```



STUB: a small piece of code that gives the address to BSA.

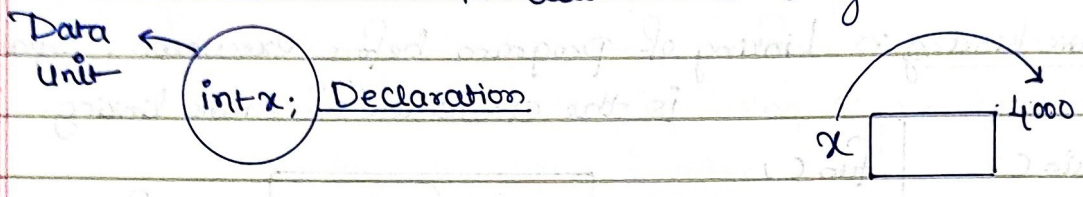
.dll: (Dynamic link library). The libraries linked at runtime using principle of dynamic linking is called DLL

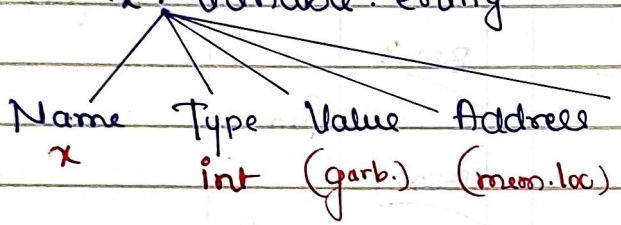
Advantages:
* Space efficiency
* Reusability
* Modification (flexibility) is easy.

Drawbacks:
* Time inefficiency
* Less security (Insecure)

Note: Static linking is more secure

Address Binding: Association of program instructions and data units to memory locations (address) is address binding.

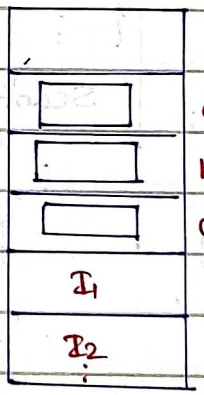




Q1. The capacity of a memory unit is defined by the number of words multiplied by the number of bits/word. How many seperate address and data lines are needed for a memory of 4K x 16?

Ans $n = \log_2 4K = 12$ bits, $m = 16$ bits $= 2B$ = Dataline/bus

```
int a, b, c;
a=1;        I1: Store a, #1
b=2;        I2: Store b, #2
c=a+b;      I3: Load r1, a
            I4: load r2, b
            I5: Add r1, r2
            I6: Store c, r1
```



Binding time: Time at which the binding takes place is called as binding time.

```
int (x);
x=1;
```

Type : at compile time (static binding)
Address : at load time (St. b)
Value : run time (address binding)
Size : Compile time (St. b)

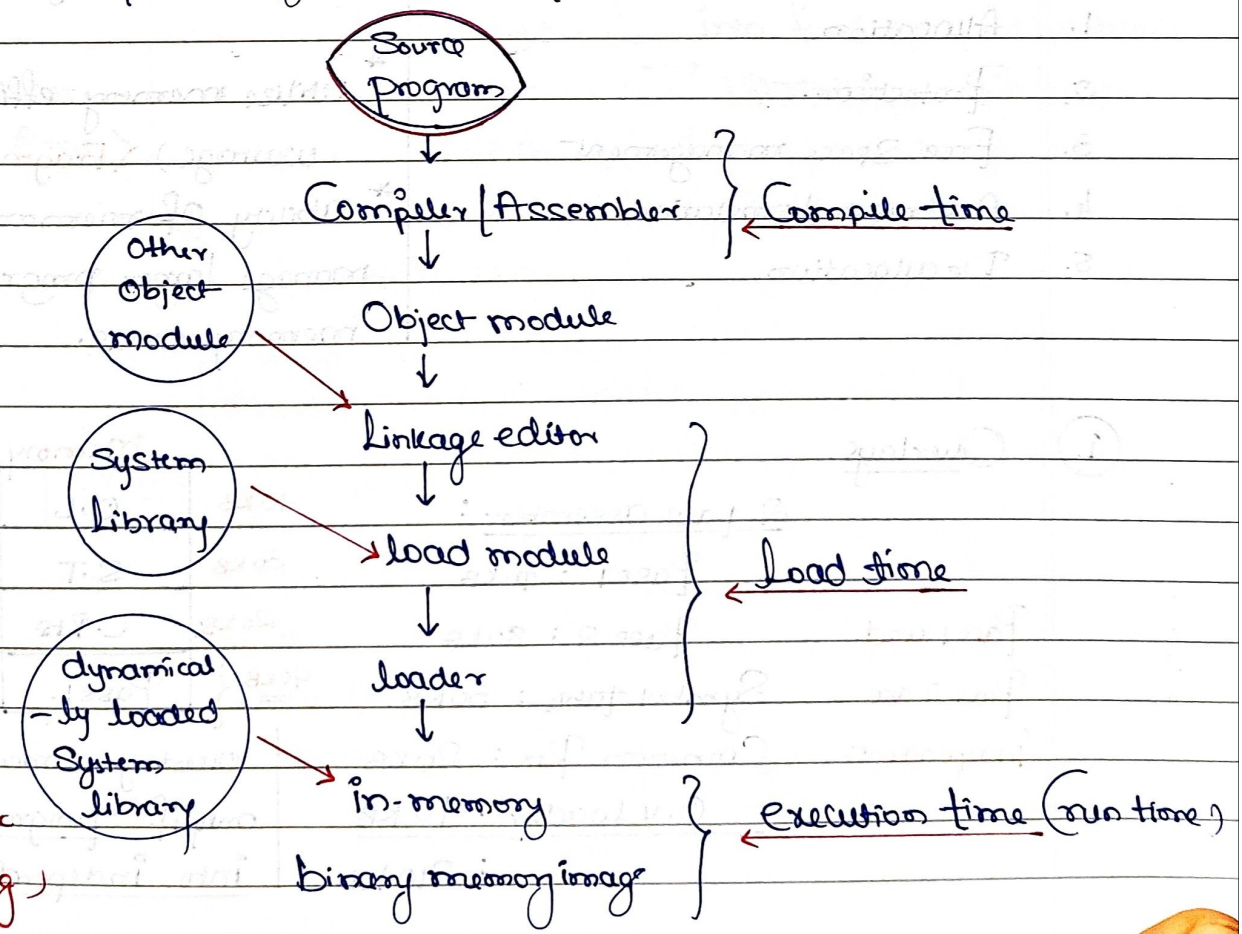
Types of binding: 1. Static binding (cannot change)
2. Dynamic binding (can change)

Address Binding

(Static) Compile time	(Static) Load time	(dynamic) Runtime address binding
	Loader.	Loader.
Compiler: Address	→ Compiler will not generate/associate address	→ Relocation happens in runtime (relocatable)
I1: 4000		→ Flexibility is advantage
I2: 4004		
I3: 4008	→ Loader will assoc.	

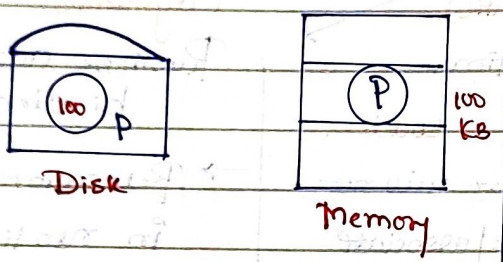
I1: 0, I2: 4, I3: 8 } offset. 6000 Base address (0.5)

Multistep processing of a user program.



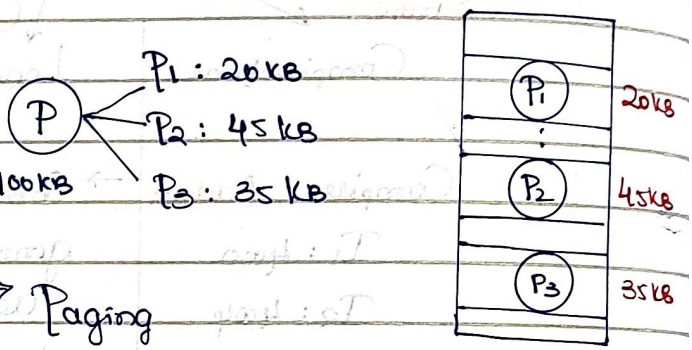
Memory Management Techniques

Contigous <centralized>



Techniques →
- Overlaye
- partitions → fixed, variable
- buddy system

Non-Contigous <decentralized>



- Paging
- Segmentation
- Paging and Segm.
- Demand paging.

Function of Memory Manager:

1. Allocation
2. Protection.
3. Free Space management
4. Address translation
5. Deallocation

Goals of memory manager:

* Utilize memory efficiently (min wastage) <Fragmentation>
* ability of memory manager to manage larger program in small memory areas.

(1.) Overlays.

2 pass assembler :
- Pass 1 : 70 KB
- Pass 2 : 80 KB
- Symbol table : 30 KB
- Common Rts : 20 KB
- O.V loader : 10 KB

: 210 KB.

Pass 1 and Pass 2 are independent.

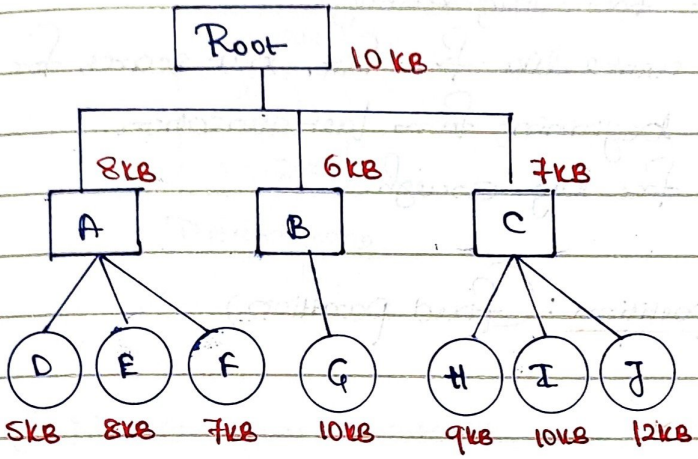
Memory : 150 KB

10 KB	O.L
30 KB	S.T
20 KB	C.Rts
90 KB =	Pass 1.

{ overlay = replace }

Overlaye concept is possible only if "program divisible into independent modules."

Consider the following program expressed as an overlay.



Prog-Size : 92KB (on disk)

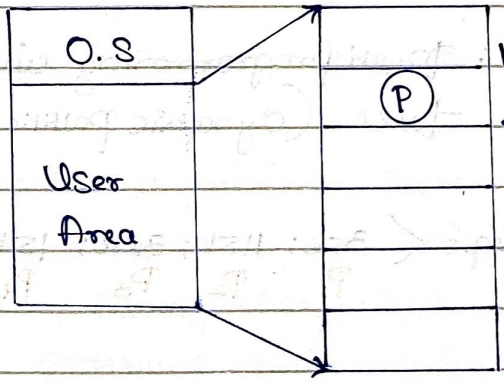
* What is minimum memory required to successfully execute this program using Overlay?

Min. Memory = max { Path-length Required from root to leaf }.

= 29KB

(2.) Partitions:

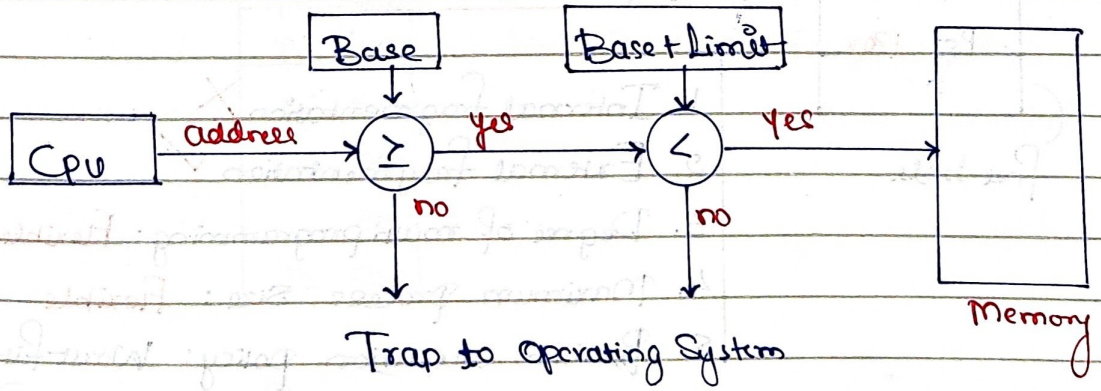
(i) Fixed partitions (MFT): Multiprogramming with fixed tasks.



→ User area will be divided into fixed no. of partitions

→ 1 partition = 1 partition

→ partitions maybe of different sizes.



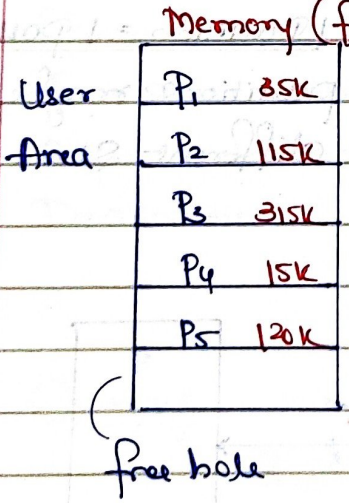
Partition allocation policies:

1. First fit: first free big enough (Internal fragmentation)
2. Best fit: smallest free big enough
3. Next fit: next fit works like first fit, but search for free partition beginning from last allocation.
4. Worst fit: largest free big enough.

Performance of fixed partition: (fixed partition)

1. Internal fragmentation ✓
2. External fragmentation ✗
3. Degree of multiprogramming: Limited
4. Maximum process size: Limited
5. Partition allocation policy: Best fit (less internal fragment.)

(ii) Variable partitions (MVT): multiprogramming with variable tasks (dynamic partitioning).

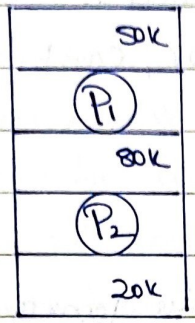


Process Req.'s < 35K; 115K; 315K; 15K; 120K; ....
P1 P2 P3 P4 P5

Performance issues:

1. Internal fragmentation ✗
2. External fragmentation ✓
3. Degree of multiprogramming: Flexible
4. Maximum process size: Flexible
5. Partition allocation policy: Worst fit is better.

External fragmentation:

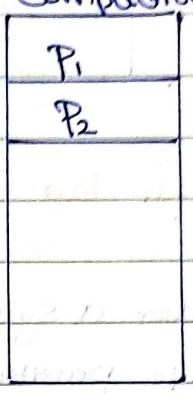


Memory map.

Free space: 150K
Process: P3 - 85K
(External frag of 150K)

⟹ relocate

(i) Compaction (time consuming operation)



* Compaction is not possible if there is no runtime address binding.

(ii) Non Contigous allocation:

* One of the reason of having non contigous allocation is to avoid the problem of external fragmentation
* N.C.G allocation divides the process into smaller parts and distributed to free holes.

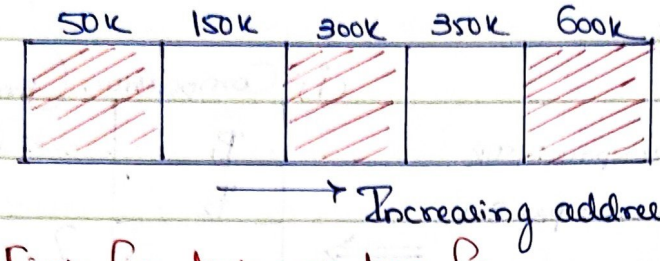
Note: Adjacent free holes are automatically merged to one free hole.

Q1 Consider a memory system having 6 partitions of sizes 200K, 400K, 600K, 500K, 300K, 250K. There are 4 processes of size: 357K, 210K, 468K, 491K. Using Best fit allocation policy, what are partitions that remain Unallocated?

P4	200
P1	400
	600
P3	500
	300
P2	250

600K, 300K remain unallocated

Q2 Consider the following memory map in which blank regions are not in use and hatched regions are in use. Using variable partitions with no compaction.



The sequence of requests for blocks of size 300K, 25K, 125K, 50K, can be satisfied if we use:

Ans First fit but not best fit.

Q3. Consider a system with memory of size 1000KBytes. It uses variable partitions with no Compaction. Presently there are 2 partitions of sizes 200K and 260K respectively.

(i) What is the allocation request of the process which could be always denied?

Ans: 541K

(ii) What is the smallest allocation request which could be possibly denied?

Ans: 181K

Q4. Consider a system having memory of size 2^{46} Bytes, uses fixed partitioning. It is divided into fixed size partitions each of size 2^{24} Bytes. The OS maintains a process table with one entry per process. Each entry has 2 fields: first, a pointer pointing to partitioning in which process is loaded and second, field is Process Id. The size of PID is 4 Bytes.

Calculate:

(a) The size of pointer to the nearest Byte :- 3B

No. of partitions (N) = $\frac{2^{46}}{2^{24}} = 2^{22}$, Partition address = 22 bits = 3Byte

(b) Size of process table in Byte is if system has 500 process.

Ptr Size = 3B

Process id = 4B

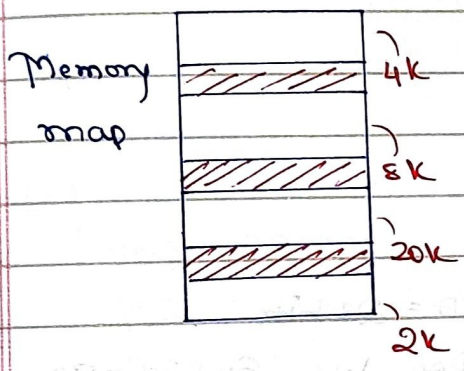
7B × 500 → 3500 B

Qs. Consider a system with Variable Partition with no compaction

HW

Free holes: 4K, 8K, 20K, 2K.
Program size: 2K, 14K, 3K, 6K, 10K, 20K, 2K.
Time for execution: 4, 10, 2, 1, 4, 1, 8.

Using best fit allocation policy and FCFS CPU Scheduling, find the time of loading and time of completion of each program. The fault time in seconds are.



CPU

P_1	P_2	P_3	P_4	P_7	P_5	P_6

0 4 14 16 17 25 29 30

Q6. Consider the allocation of memory to a new process. Assume that none of the existing holes in the memory will exactly fit the Processes memory requirement. Hence, a new hole of Smaller Size will be created if allocation is made in any of the existing holes. Which of the following is true?

Ans- The hole created by best fit is never larger than hole created by first fit.

Non Contigous Allocation: (to prevent external fragmentation)

Address Space: A set of space/words associated with addresses

Logical Address Space

Virtual (A.S)

Physical Address Space.

Physical. A.S

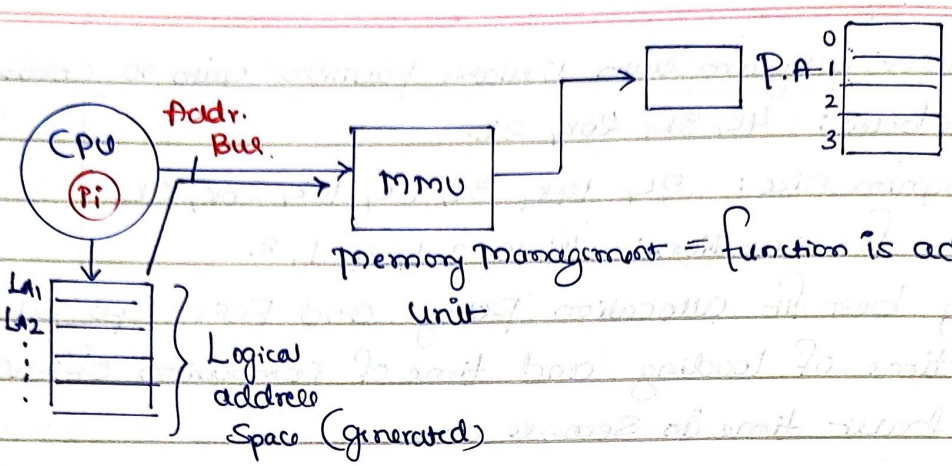
0
1
2
3
:

→ Words (Cells) locations

$PAS = 2^{P.A}$

$P.A = \log_2 PAS.$

RAM/memory



Memory Management = function is address translation.
Unit

Eg: ① L.A.S = 64 MB ; PAS = 4 MB
LA = 26 bits PA = 22 bits

② LA = 33 bits
LAS in bytes, if word size is 64 bits = 8B.
$LAS(B) = 2^{33} \times 8B = 8G \times 8B$
$= 64\,GB$

PA = 23 bits
PAS, W.S = 64 bits = 8B
$8M \times 8B = 64MB$

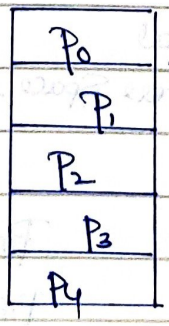
* Logical address: address generated by CPU to access instruction/data unit of program in execution.

* Physical Address (Real/Absolute): address needed to access needed to access the program instructions/data unit in physical memory ie address in MAR.

1. Simple Paging:

* Organising of LAS/VAS:

LAS: 5KB. → LAS is divided into equal size units known as pages.



→ Page size is generally in the power of 2.