Software : set of tested programs + documentation

Software

- System Software
  (low-level software)
  [runs in background]
  - operating S.S.
  - Device Drivers
  - Translators
    - Compiler
    - Interpreter
    - Assembler
  - Utility
    - antivirus
    - cleanup tools
    - defragmenters

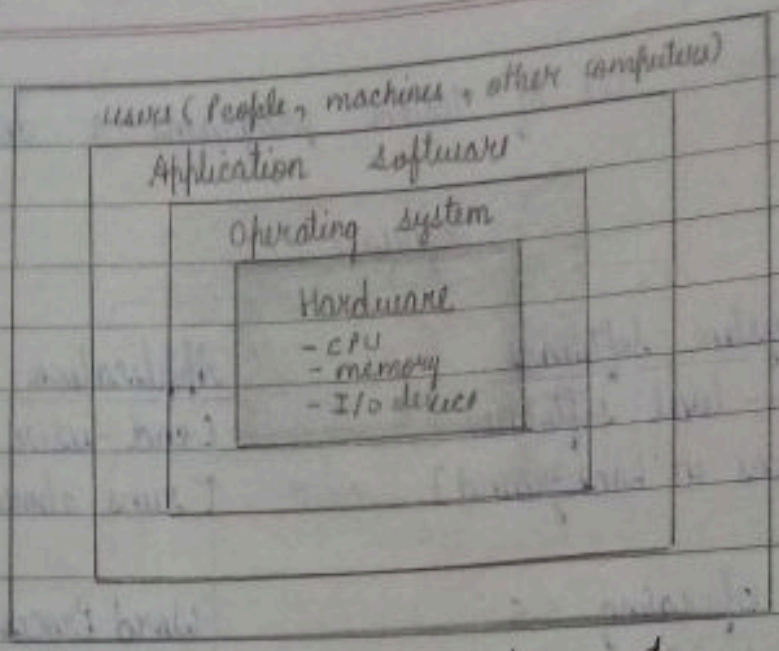
- Application software
  (end-user software)
  [runs above system software]
  - Word Processors
    Notepad, MS Word
  - Database software
    MySQL, MS Access
  - Multimedia software
    VLC Media Player, Windows M.P.
  - Graphics software
    Adobe Photoshop, Blender
  - Web Browsers
    Google chrome, Opera, UC

OPERATING SYSTEM SOFTWARE

[kernel - core program] + [system prog]

User's View :
It is an interface between the user and the hardware [CPU, I/O devices, memory]

System's View : Control program
It is a resource Manager



components of computer system

Interface :

- CUI [Character User Interface]
  eg : DOS
- GUI [Graphical User Interface]
  eg : Windows

B+

Dt.
Pg. B+

FUNCTIONS

- Resource Management
- CPU Scheduling / Process Management
- Memory Management (Mainly RAM)
- I/O device Management drivers drivers
- Storage Management [disk / file system]
- Security & Protection passwords etc
- Process synchronization
- Coordination between other software & users
- deadlock handling

GOALS

- Efficient utilization of resources
- Convenience (user-friendly)

Dt _____ B+
Pg _____

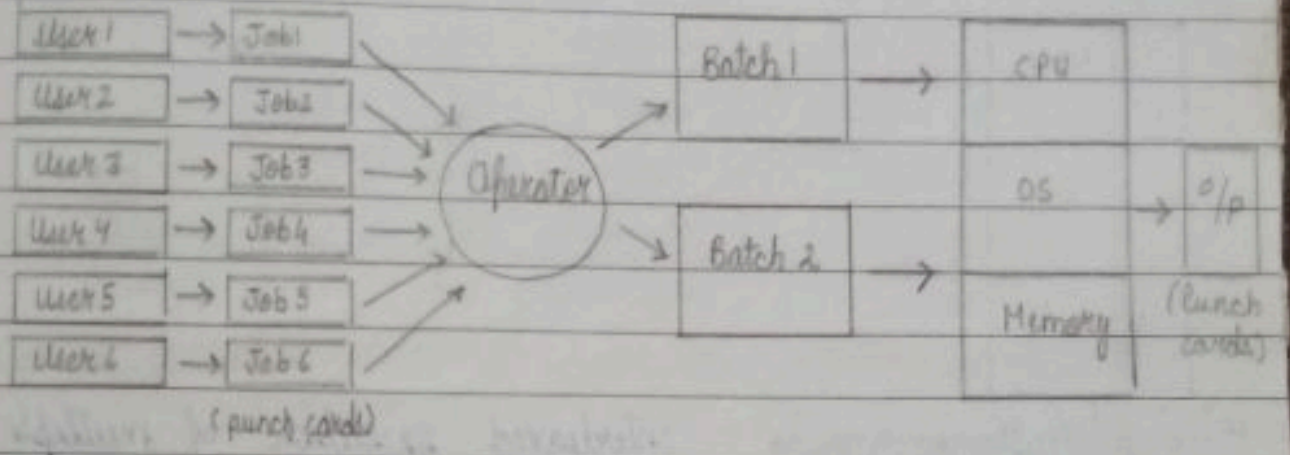
TYPES

eg: job

Calculator Program	
Input: 12 x 2	instruction division

12/2

1) Batch Operating System

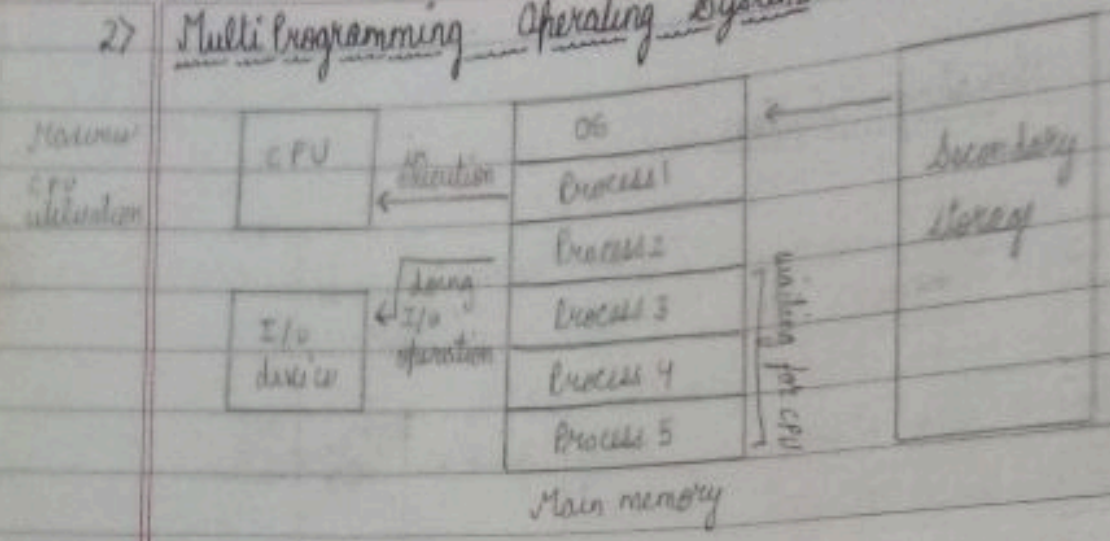


Job: Program + input data + control instructions
Batch: group of similar kind of jobs

→ Each user prepares its job on an off-line device (punch cards and submits it to the computer operator.
→ Computer operator then groups similar kind of jobs to speed up the processing [All the jobs in a batch required similar resources] → time of allocation/deallocation reduced
→ Batches are fed to the computer one by one. One job is executed at a time. First come first served
→ Then output is obtained (in the form of punch cards) and returned to the user.

Advantages	Disadvantages
→ Saves time that was being wasted earlier (context switching)	→ CPU remain idle for long time
→ No manual intervention required	→ May lead to starvation
→ Multiple users can share batch system	→ Priority can't be set for jobs
	→ lack of interaction b/w user & job

2) Multiprogramming Operating System



Multiprogramming : Interleaved execution of multiple processes by the same computer

Note : Only one process is executed by a processor at a time

→ In multiprogramming, multiple processes are loaded into the main memory which are ready to execute
→ If running process performs I/O operations etc, CPU doesn't sit idle rather picks another process for execution
→ Therefore CPU never sits idle [except at the time of context switching or if no process is ready for execution]

Advantages	Disadvantages
• High CPU utilization	• Scheduling is required
• Waiting time is less	• Main memory managment is required [Memory fragmentation, paging]
• Increased throughput	• CPU is allocated to another process only if it is idle
• priority can be assigned	• Context switching

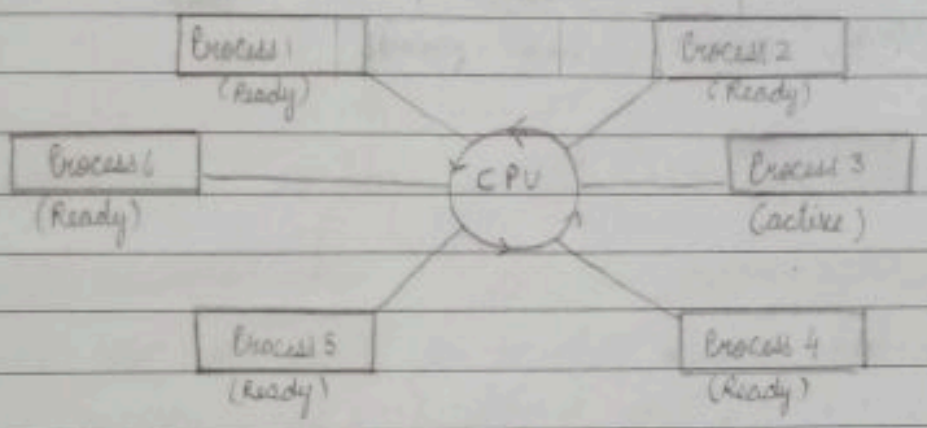
Multi user
pre emptive
CP util
WT
RT
throughput

Dt _____ B+
Pg _____

3) Multi Tasking Operating System

Minimal Response time

(Time Sharing / fair share / Multiprogramming with Round - Robin)



Multitasking : execution of multiple tasks by sharing processor's time

Time quatum / slice / slot : short period of time during which a task gets ~~user~~ CPU attention. (10 - 100 millisec.)

→ Multiple processes are loaded into the memory and CPU switches between them in fixed time intervals (time quantums)
→ Time slot is decided by the OS
→ Context switching is frequent enough to give an expression that all tasks are being executed simultaneously
→ Reduces response time (main goal)

Advantages

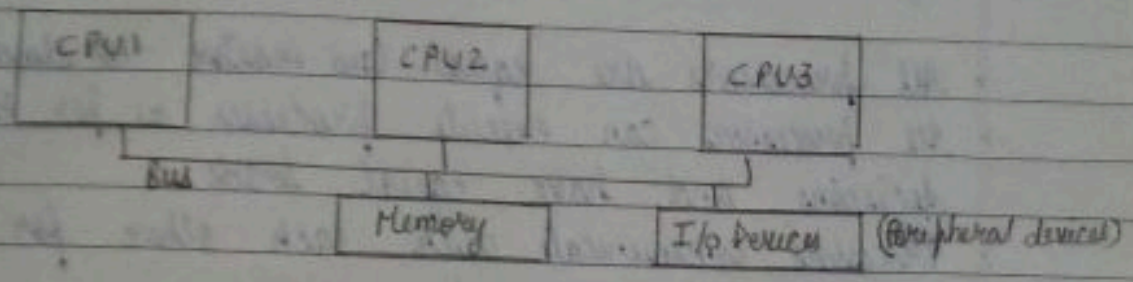
- response time ↓
- High CPU efficiency
- Throughput ↑
- Multi - user & user friendly

Disadvantages

- context switching overhead
- data communication problem
- system must have higher
- specification

Tutorial

Multi Processing Operating Systems



Multiprocessing: executing multiple processes simultaneously using multiple processors.

→ Multiple processors work in parallel sharing the resources like memory, bus, peripheral devices, system clock etc (tightly coupled systems)
→ Applications designed for the use in multiprocessing are threaded, so they can be broken into smaller routines ~~and~~ that are executed on multiple processors independently
→ Each CPU contains a copy of the OS and these copies communicate with one another to coordinat operations.
→ Used when large data is to be processed at high speed
eg: satellite control, weather forcasting etc

Types:
- Symmetric Multiprocessing OS (SMP OS)
- Asymmetric Multiprocessing OS (ASMP OS)

Adv.	Disadv.
- high speed, time ↓ throughput↑	- Process sync required
- if one CPU fails, others handle the execution	- Memory management "
- cost effective [shared resources]	- CPU scheduling "

Tutorial

i) Symmetric Multiprocessing OS

→ All processors are equal (no master - no slave)
→ All processors can execute processes as per their decisions and have equal loads.
→ Processors communicate with each other for the execution if required.

advantages	Disadvantages
- If one processor fails, other handles the execution - No dependency.	- difficult to design

ii) Assymmetric Multiprocessing OS

→ It uses the concept of Master-Slave architecture
→ Only one processor work as a primary or master processor which controls the other slave processors.
→ Some processors are assigned maintenance task while others perform application tasks (asymmetry)

advantages	Disadvantages
- easy to design	- If master processor fails, execution stops.

Dt _____
Pg _____ B+

PROCESSES

program : set of instructions that perform a specific task.
(passive unit)

Process : a program in execution
(active unit)
- It is an instance of a computer program that is being executed by one or many threads.

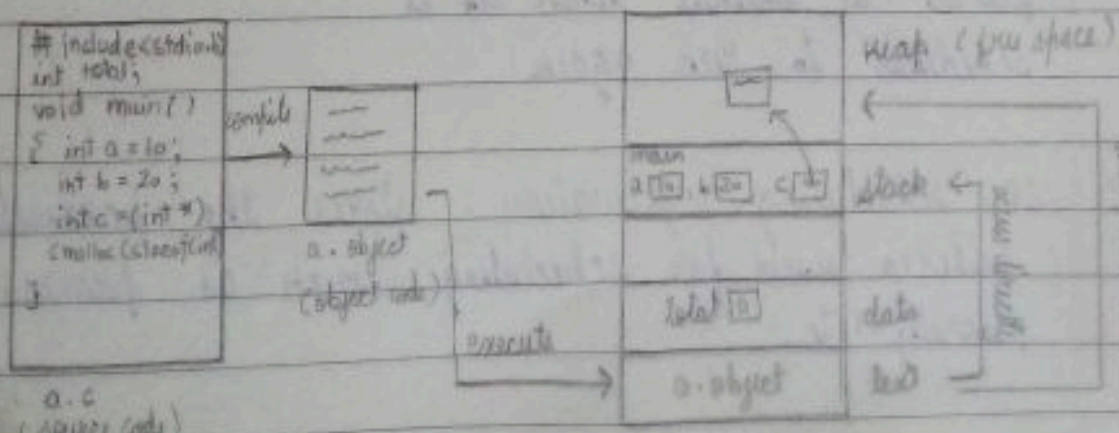
- **Stack** : contains the temporary data such as function parameters, local variables, return addresses etc
[fixed size] (can be accessed directly)
size is determined during compilation & allocated when process begins to execute
implemented using stack DS

- **Heap** : Dynamically allocated memory to a process during runtime [via new, malloc etc.]
[dynamic size] (accessed using pointers)
size is determined during runtime & allocated during execution
NOT implemented using Heap DS

Stack
↓
↑
Heap
Data
Text

- **Data** : global and static variables

- **Text** : compiled program code [current activity represented by the value of program counter & contents of processor's registers]



B+

PCB (Process Control Block / Task Control Block)

PCB is a data structure used by OS to maintain all the information about a process
[Brain of a Process]

- Process ID : unique Id / number provided to each process by the OS
- Pointer : points to parent process
- Process state represents current state of the process : new, ready, running etc.
- Process counter : represents the address of next instruction to be executed
- CPU registers : contains state info of the process which allows the process to continue when its is scheduled to run again (accumulators, index registers, stack pointers etc)
- CPU scheduling information : stores the parameters of criteria used for scheduling such as process priority etc.

Process ID
Pointer
Process state
Process Counter
CPU registers
CPU scheduling
Memory managment info
Accounting info
I/O status info
Process privileges

- Memory Management info: stores limit register info, page and segment tables etc.

- Accounting info keeps bookkeeping data such as CPU time used, time limits, job numbers etc.

- I/O status info: list of I/O devices allocated to the process, list of open files.

- process priveleges: access-permissions for resources of the system.

Dt. _____ B+
Pg. _____

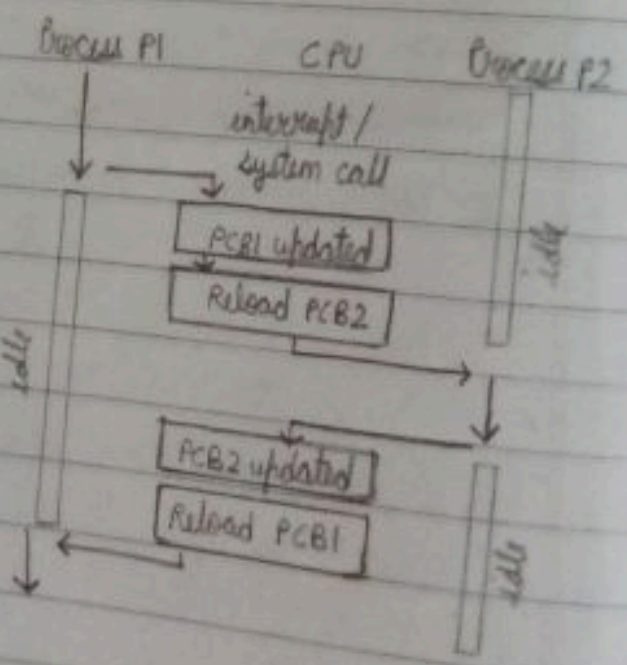
Context switching remove PCB of running process & place PCB of next process in queue

- It is a process in which CPU switches from one process or task to another
- The state of currently running process is saved (so that it can resume execution later) and another process is alloted to the CPU
- It is used for multitasking
- Context switching occurs so fastly that it gives an illusion that all processes are being executed at the same time

(state of running process is switched)

Steps involved:

- Context of the process P1 (running currently) is saved in the PCB of process 1
- PCB1 is moved to relevant queue (ready queue, I/O queue, waiting queue etc.)
- New process that is to be executed is selected from ready queue
- PCB of process 2 is updated.



→ Triggers for context switching
- Multitasking
- Interrupt Handling
- User and kernel Mode switching

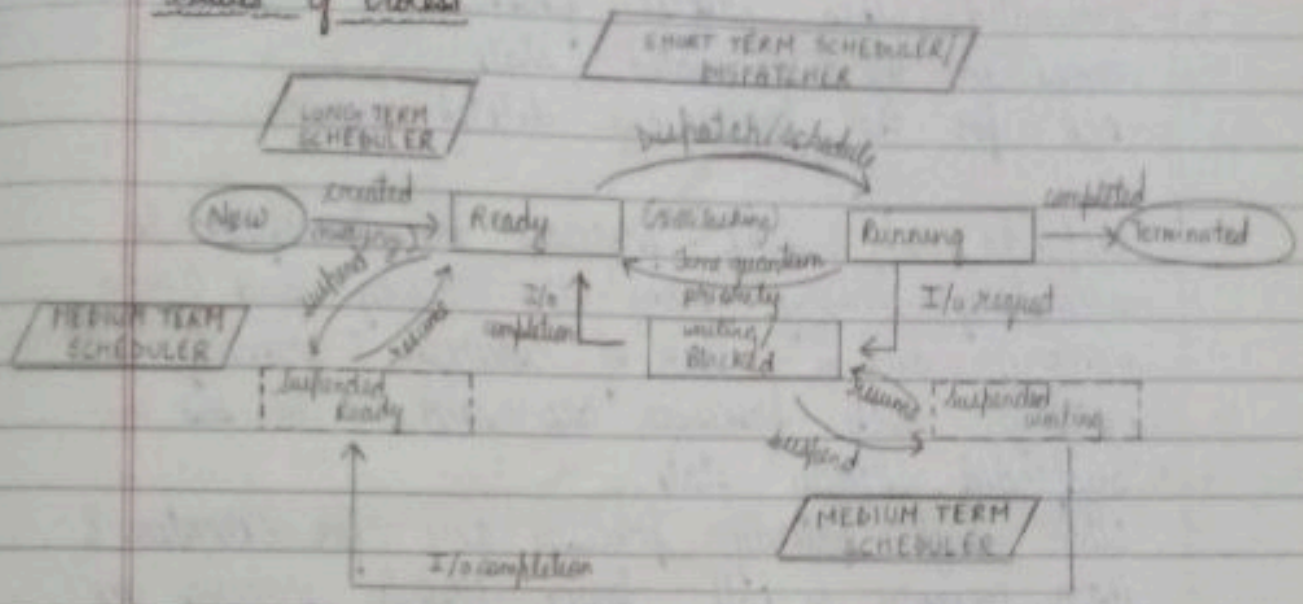
Advantages
- used for multitasking / multiprograming.
Only 1 CPU is required to run multiple processes.

Disadvantage
- context-switch time is pure overhead (No useful task is performed during context switching)
(takes few multi sec. (switch)

Job Queue
Ready Queue
Waiting Queue

Dt _____
Pg _____ B+

States of Process



- New: The program which is to be picked up by the OS for execution (from the secondary memory) is said to be in 'new' state
  At a time - multiple processes can be in new state
- Ready: After creation, process is transfered to main memory in ready queue & is said to be in 'ready' state
  A process in ready state waits for the CPU to be assigned
  At a time - multiple processes can be in ready state
- Running: Once a process is chosen by OS for execution from the ready queue, it enters into 'running' state
  At a time - only 1 process can be in running state (for single processor OS)
  - n processes — (for n processors)
- Waiting / Blocked: Whenever a process requires other resources (I/O devices etc), it enters in 'waiting' or 'blocked' state & waits for the resources (in waiting queue)
  At a time - multiple processes can be in waiting state

- Terminated: When process finishes its execution, it comes in the terminated state. deallocation of resources PCB for the process is deleted by OS.

- Suspended ready: Some processes are transferred to secondary memory from the ready queue (main memory) due to lack of resources (primary memory). Such processes are said to be ~~sts~~ in 'suspended ready' state.
  - When high priority processes come for execution & main memory is full, lower priority processes are transfered to sec. memory.
  - Suspended ready processes wait in the sec memory until space gets available in the main memory & are then transfered to the main memory again.

- Suspended waiting: When processes are transfered to sec. memory from the waiting or blocked state, they are said to be in 'Suspended waiting' state (due to lack of space in main memory)
  A process in suspended waiting state still executes & is transfered to suspended ready state in case I/O op. is completed [& if still not completed & space gets available in waiting queue → returns to the waiting queue]

Throughput - Turnaround time
waiting time Response time

Dt _____ B+
Pg _____

→ CPU bound processes : processes which spends most of their time on executing the code i.e take more CPU time.
eg : • complex mathematical algorithms
• searching / sorting algorithms
• encryption decryption algorithms

→ I/O - Bound processes : processes which spend most of their time submitting I/O requests and waiting for the response i.e takes more I/O time.
eg : word processor
[more time is spent in waiting for input & less on CPU - spell checking, auto suggestion etc]

→ Throughput : Number of processes completed per unit time

point of time → Arrival time : The point of time at which process enters the ready queue / state.

duration → Burst time : time required by a process to get executed on CPU
(execution time / running time / CPU time)

point of time → Completion time : The time at which process completes its execution (exit time)

duration → Turnaround time : Total time spent by a process in the system
TA time = Completion / exit time - arrival time
= Burst time + waiting time

→ Waiting time : time spent by process in ready queue (waiting queue) waiting for the CPU.
waiting time = TA time - Burst time

→ Response time : Time between a process enters ready queue and gets scheduled on the CPU for the first time.
RT = time at which process gets CPU for the first time - arrival time.

→ Starvation (indefinite blocking)
Phenomenon in which a process is ready to run but has to wait indefinitely due to lower priority [when CPU is biased]

Dt
Pg B+

Schedulers special system software used to handle process scheduling

1) Long Term Scheduler
Selects processes from secondary memory & loads them into main memory (in ready queue) for execution
- controls the degree of multiprogramming [no of processes that can fit in ready queue]
- LT scheduler must select a careful mixture of I/O bound & CPU bound processes
  - if CPU bound processes ↑ → I/O devices remain idle
  - if I/O bound processes ↑ → CPU remains idle.

2) Short Term Scheduler
selects a process from the ready queue for execution based on some scheduling algorithm. CPU scheduling
- executes most frequently.
- ensures that, there is no starvation due to high burst time process

3) Medium Term Scheduler
swaps out a process from main memory [from ready queue / waiting queue state] & places into sec memory [suspended ready / suspended waiting ~~queue~~ state] & later swaps in
- responsible for suspending & resuming a process
for maintaining balance b/w I/O & CPU bound processes
dec degree of multiprog.

Dispatcher responsible for loading the process selected by ST scheduler on the CPU from ready queue
- context switching

Tutorial

Dt _____ B+
Pg _____

frequency : ST > MT > LT

Dt
Pg B+

PROCESS - SCHEDULING

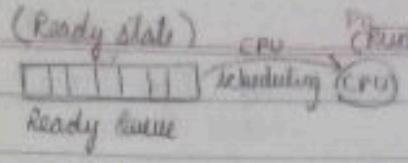
It is an OS task which involves allocation of resources and time to the processes in multiprogramming systems.

- handles the removal of the running process from the CPU & selection of another process on the basis of a particular algorithm.
- schedules processes in different states - ready, running and waiting

- OS maintains separate queue for each of the process state:

  - Job Queue: keeps all the processes ~~int~~ in system
  - Ready Queue: keeps all the processes residing in main memory, ready to execute
  - Device Queue: keeps all the processes which are waiting for I/O devices.

CPU Scheduling



Process of determinging which process will use the CPU while another process is on hold (for maximum CPU utilisation) based on different algorithms
[using ST scheduler]

Criteria

- CPU utilization should be maximum $\left[\frac{\text{expected time}}{\text{actual time}} \leq 1\right]$
  ideally - 100% ; practically - 40% - 90%
- Throughput should be high
  1 of second
- Avg. Turnaround time should be minimum
- Avg. waiting time minimum
- Avg. response time minimum

Dt
Pg

B+

CPU Scheduling Algorithms

1> Preemptive Scheduling	2> Non-Preemptive Scheduling
• CPU can be allocated to another process of high priority (or due to time quantum) in between the execution of currently running process.	• CPU can't be allocated to another process until current process completes its execution
• efficient CPU utilization	• Poor CPU utilization
• WT & RT is less	• WT & RT is high
• overhead of switching the process from ready state to running state & vice versa.	• No such overhead
• CPU is allocated to a process for specific time (may be less than its BT)	• CPU is allocated to a process until it terminates (equal to its BT)
• Round Robin, SRTF, preemptive priority. low priority processes may face starvation	• FCFS, non-preemptive SJF, non-preemptive priority may lead to convoy effect

CPU Sched. Algo → FCFS, SJF, LJF, Multilevel Queue
SRTF, LRTF, Round Robin, Priority

Ready queue
→ CPU ticked ✗
→ ——— ✗
→ ——— ✓

Dt _____ B+
Pg _____

FIRST COME FIRST SERVE (FCFS)

[available for batch OS]

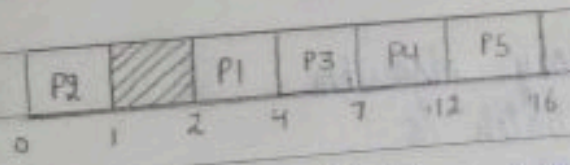
(simplest algorithm)
assigns CPU to the process which arrives first into the ready queue
- Non-premptive scheduling (WT = RT)
[as soon as a process gets response - doesn't have to wait anymore - completes & exits]

Example:
(Assumption : I/O time is not required for any process)

Process	Arrival time (AT)	Burst time (BT)	completion time (CT)	CT - AT Turnaround time (TAT)	TAT - BT Waiting time (WT)	Response time (RT)
P1	2	2	4	4-2 = 2	2-2=0	0
P2	0	1	1	1-0 = 1	1-1=0	0
P3	2	3	7	7-2 = 5	5-3=2	2
P4	3	5	12	12-3 = 9	9-5=4	4
P5	4	4	16	16-4=12	12-4=8	8
				29	14	14

*[if more than 1 process arrive at same time - any of them can be taken]

Gantt chart :



→ average TAT : 29/5 = 5.8
→ average WT : 14/5 = 2.8
→ average RT : 14/5 = 2.8
→ throughput : 5/16

Tutorial

Convey Effect

It is a phenomenon associated with FCFS in which processes of higher burst time comes before the processes with lower burst time resulting in high avg waiting time.

a CPU bound process followed by I/O bound processes

Case: 1

Process	AT	BT	TAT	WT
P_1	0	50	50	0
P_2	1	1	50	49
P_3	1	2	52	50
				99

starvation: until due to priority

gantt chart:

P1	P2	P3

0 50 51 53

avg waiting time = 99/3 = 33

Process	AT	BT	TAT	WT
P_1	1	50	52	2
P_2	0	1	1	0
P_3	0	2	3	1
				3

gantt chart:

P_2	P_3	P_1

0 1 3 53

avg waiting time = 3/3 = 1

Case 1 suffers from convey effect.

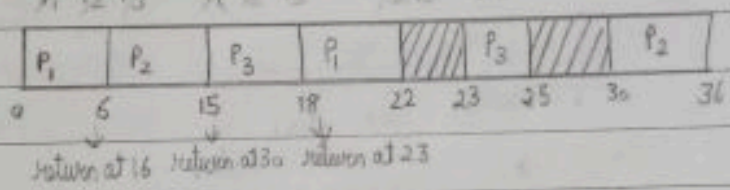
Adv.: - simple & easy algorithm
- suitable for batch OS

Disadv.: - convey effect
- non-preemptive nature - CPU utilisation is low.
- doesn't respond according to priority of process

→ Three processes P_1, P_2, P_3 with process time 20, 30, 10 respectively. Each process ~~#~~ uses first 30% of its process time on CPU then 50% ~~on~~ in I/O & last 20% on CPU.
Find avg WT, TAT, RT if FCFS scheduling is followed

Process	PT	AT	BT	I/OT	BT	CT	TAT	WT	RT
P_1	20	0	6	10	4	22	22	12	0
P_2	30	0	9	15	6	36	36	21	6
P_3	10	0	3	5	2	25	25	20	15
							83	53	21

gantt chart:



avg. RT = 21/3 = 7
avg. WT = 53/3 = $17.6\bar{6}$
avg. TAT = 83/3 = $27.\bar{6}$
throughput = 3/36 = 1/12
CPU utilization factor = $\frac{30}{36}$ = 0.833 = 83.3%

CPU idle time factor = $\frac{6}{36}$ = $\frac{1}{6}$ = 16.66%

SHORTEST JOB FIRST (SJF)

assigns CPU to the process which has smallest burst time out of all the processes present in ready queue
- effectively reduces the average waiting time.

- Preemptive SJF (SRTF)
- non-preemptive SJF (normal)

Non-preemptive SJF

Once CPU is assigned to a process, it can't be snatched by another process, even if ~~it~~ its burst time is smaller.

Example:
* I/O time ×

Process	AT	BT	CT	TAT	WT	RT
P_1	2	1	7	5	4	4
P_2	1	5	16	15	10	10
P_3	4	1	8	4	3	3
P_4	0	6	6	6	0	0
P_5	2	3	11	9	6	6
				39	23	23

* [if more than one process have same BT → follow FCFS]

gantt chart:

P_4	P_1	P_3	P_5	P_2

0 6 7 8 11 16

throughput : 5/16
avg waiting time : 23/5 = 4.6
avg response time : 4.6
avg TAT : 39/5 = 7.8

Preemptive SJF
(SRTF Shortest Remaining Time First)

CPU can be allotted to another process if its burst time is smaller than the remaining time of currently running process

Example:

Process	AT	BT	CT	TAT	WT	RT
P_1	2	1	3	1	0	0
P_2	1	5	16	15	10	10
P_3	4	1	5	1	0	0
P_4	0	~~6~~ 5 4	11	11	5	0
P_5	2	~~3~~ 2	7	5	2	1
				33	17	11

P_4, P_2, P_4, P_5, P_3

gantt chart:

P_4	P_4	P_1	P_5	P_3	P_5	P_4	P_2

0 1 2 3 4 5 7 11 16

avg. TAT = 33/5 = 6.6
avg. WT = 17/5 = 3.4
avg. RT = 11/5 = 2.2
throughput = 5/16

If arrival time is same for all processes
non preemptive SJF = preemptive SJF

Dt _____
Pg _____ B+

Adv. :
- min. avg WT & TAT (SRTF) than all other algos
- better avg response time
- max. throughput (SRTF)
- provides a standard for other algo [for WT]

used in batch OS

optimal when arrival time is same for all processes.
(no convoy effect)

Disadv. :
- can not be implemented bcz it depends on BT of the processes which is usually not known beforehand. (approx BT can be determined)
- may lead to starvation of the processes with larger burst time. [Longer BT process may have to wait for ∞ time]
- convoy effect in non-preemptive SJF
- priorities can't be set.
- poor response time for processes with larger BT

Tutorial

Prediction of burst time

To implement SJF algo, BT needs to be known before. Following techniques are used to predict BT :

→ Static Techniques

- Process size
  If P_{old} (100 kB) - 5 units
  ∴ P_{new} (102 kB) ≃ 5 units
  As their size is approximately same, their BT is predicted to be equal.

- Process type
  - OS process (system process)
  [schedulers, compilers, program managers etc]
  lower BT - 3 to 5 units

  - User process
    - interactive process I/O bound
      lower BT
    - foreground process I/O bound + CPU bound
      slightly higher BT MS office, editors
    - background process CPU bound
      higher BT key logger

→ Dynamic Techniques

- Simple Averaging

$$P_6 = \frac{\overset{P_1}{t_1} + \overset{P_2}{t_2} + \overset{P_3}{t_3} + \overset{P_4}{t_4} + \overset{P_5}{t_5}}{5}$$

predicted burst time of $(n+1)$th process $= \frac{1}{n}\sum_{i=1}^{n} T(i)$

- Exponential Averaging

$$\boxed{\varepsilon_{i+1} = \alpha A_i + (1-\alpha)\varepsilon_i}$$

ε_{i+1} = expected time for process $i+1$

ε_i = expected time for process i

A_i = Actual burst time of process i

α = smoothering factor

$0 \le \alpha \le 1$

- If $\alpha = 0$ $\quad \varepsilon_{i+1} = \varepsilon_i$ expected BT of previous process
- If $\alpha = 1$ $\quad \varepsilon_{i+1} = A_i$ Actual BT of previous process
- If $\alpha = 1/2$ equally weighted - ε_i & A_i

eg: $\alpha = 0.5$ $\quad \varepsilon_1 = 5$ $\quad \varepsilon_5 = ?$

Process	BT	ε_i
P_1	4	5
P_2	8	4.5
P_3	5	6.25
P_4	6	5.625

$\varepsilon_2 = 0.5(4) + 0.5(5)$
$= 2.0 + 2.5 = 4.5$

$\varepsilon_3 = 0.5(8) + 0.5(4.5)$
$= 4.0 + 2.25 = 6.25$

$\varepsilon_4 = 0.5(5) + 0.5(6.25)$
$2.5 + 3.125 = 5.625$

$\varepsilon_5 = 0.5(6) + 0.5(5.625)$
$3.0 + 2.8125 = \underline{5.8125}$

→ Three processes P_1, P_2, P_3 with process time 20, 30, 10 respectively. Each process uses its first 30% of the process time on CPU, then 50% in I/O & last 20% on CPU. Find avg WT, TAT, RT, CPU utilisation % if system follows SJF scheduling non-preemptive by default

Process	PT	AT	BT	I/O T	BT	TAT	WT (TAT - BT)	RT
P_1	20	0	6	10	4	24	14	3
P_2	30	0	9	15	6	41	26	11
P_3	10	0	3	5	2	11	6	0
						76	46	14

gantt chart:

P_3	P_1	P_3	P_2	P_1		P_2

0, 3, 9, 11, 20, 24, 35, 41

(return at 3) (return at 14) (return at 35)

avg WT = 46/3 = 15.3

avg TAT = 76/3 = 25.3

avg RT = 14/3 = 4.6

CPU utilization factor = $\frac{\text{expected CPU time}}{\text{actual CPU time}} = \frac{30}{41} = 0.7317 \approx 73\%$

CPU idle time factor = $\frac{\text{CPU idle time}}{\text{total CPU time}} = \frac{11}{41} = 0.2683 \approx 27\%$

throughput = 3/41

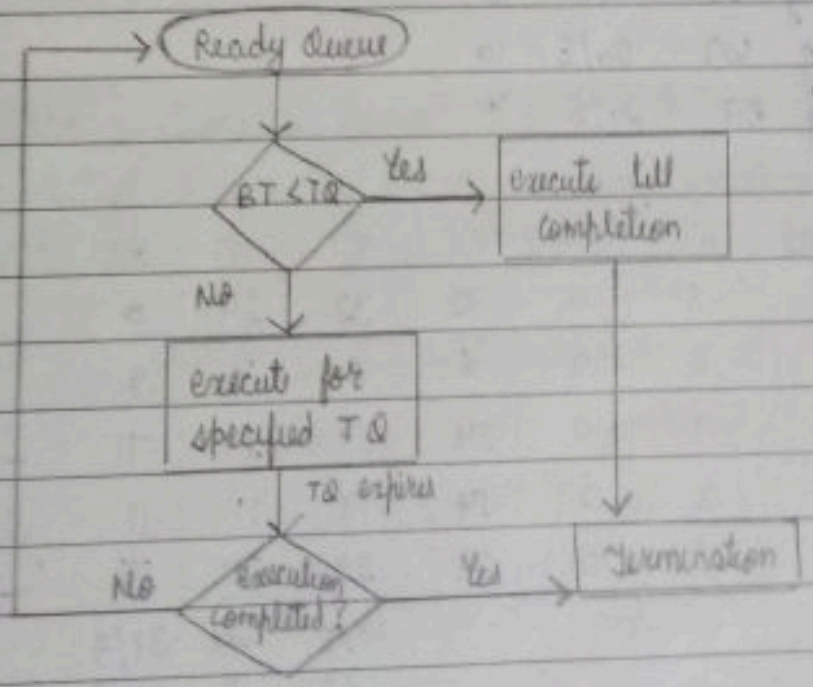
ROUND ROBIN [FCFS + time quantum]

assigns time slices to each process in equal portions (in circular order) [starvation free] [response T ↓]

- no priorities
- suitable for time sharing OS
- preemptive

Time quantum: period of time for which process is allowed to execute uninterruptedly in preemptive multitasking (10 – 100 ms)

- Time quantum must be selected carefully
  - if too small – overhead of context switching
    – waiting time ↑
  - if too large – ends up as FCFS scheduling



Example:

Process	AT	BT	CT	TT	WT	RT
P_1	0	8 ~~5~~ 2	22	22	14	0
P_2	5	2	11	6	4	4
P_3	1	7 ~~4~~ 1	23	22	15	2
P_4	6	3	14	8	5	5
P_5	8	5 ~~2~~	25	17	12	9
				75	50	20

TQ = 3

P_1	P_3	P_1	P_2	P_4	P_3	P_5	P_1	P_3	P_5

Ready Queue

gantt chart:

P_1	P_3	P_1	P_2	P_4	P_3	P_5	P_1	P_3	P_5

0 3 6 9 11 14 17 20 22 23 25

avg TT = 75/5 = 15
avg WT = 50/5 = 10
avg RT = 20/5 = 4

Process	BT	AT	CT	TT	WT	RT
P_1	8 ~~5~~ 2	0	32	32	24	0
P_2	2	0	6	6	4	4
P_3	7 ~~4~~ 1	0	34	34	27	7
P_4	3	0	14	14	11	11
P_5	5 ~~2~~	0	29	29	24	15
				115/5 = 23	90/5 = 18	37/5 = 7.4

- context switch time = 1 unit
- TQ = 3 units
- no. of context switches = 9
- CPU utilization = $\frac{25}{34}$ = 0.735 = 73.5 %

Ready Queue:

P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_5	P_1	P_3

gantt chart:

P_1		P_2		P_3		P_4		P_5		P_1		P_3		P_5		P_1		P_3

0 3 4 6 7 10 11 14 15 18 19 22 23 26 27 29 30 32 33 34

Dt. _____ Pg. _____ B+

Adv. : - easy & simple to implement BT not required as in SJF
- each process gets a fair share of CPU (same priority for all processes)
- starvation free
- no convoy effect
- diminishes response time
- suitable for multitasking

Disadv. : - deciding size of TQ is very tough
(if not decided properly may lead to WT ↑, TAT ↑, RT ↑, throughput ↓, CPU utilization ↓)
- overhead of context switching
- priorities can't be set.

→ Let there are n processes in a system. Context switch time is S seconds. What should be the max time quantum so that any process has to wait for atmost T seconds to enter in CPU again

$$[t_q * (n-1) + n * S] \leq T$$

$$t_q \leq \frac{T - nS}{n-1}$$

Tutorial

Dt. _____ B+
Pg. _____

PRIORITY SCHEDULING

A priority is associated with each process when loaded in ready queue. CPU is assigned to the process having higher priority.

- Priority is decided based on memory requirements, time requirements etc.
- In case of tie, FCFS is followed.
- Priorities can be
  - static : doesn't change throughout the execution
  - dynamic : changes after some fixed interval of time

Preemptive Priority Scheduling:

CPU can be assigned to a process with higher priority in the middle of execution of a process with lower priority.

Example:

Lower the no, higher the priority

Process	Priority	AT	BT	CT	TT	WT	RT
P_1	3	0	8 3	15	15	7	0
P_2	4	1	2	17	16	14	14
P_3	4	3	4	21	18	14	14
P_4	5	4	1	22	18	17	17
P_5	2	5	6 1	12	7	1	0
P_6	6	6	5	27	21	16	16
P_7	1	10	1	11	1	0	0
					96/7 =13.7	69/7 =9.8	61/7 =8.7

gantt chart :

P_1	P_1	P_1	P_1	P_5	P_5	P_7	P_5	P_1	P_6	P_3	P_4	P_2

0 1 3 4 5 6 10 11 12 15 17 21 22 27

Tutorial

Non-Preemptive Priority Scheduling:

lower no → higher priority

Process	Priority	AT	BT	CT	TT	WT	RT
✓ P_1	3	0	8	8	8	0	0
P_2	4	1	2	17	16	14	14
✓ P_3	4	3	4	21	18	14	14
✓ P_4	5	4	1	22	18	17	17
✓ P_5	2	5	6	14	9	3	3
P_6	6	6	5	27	21	16	16
✓ P_7	1	10	1	15	5	4	4
			27		95/7	68/7	68/7
					13.5	9.7	9.7

P_1 P_5 P_7 P_2 P_3 P_4 P_6

gantt chart:

P_1	P_5	P_7	P_2	P_3	P_4	P_6

0 8 14 15 17 21 22 27

Adv. priorities can be set

Disadv. - starvation of processes with lower priority
- deciding priorities is tough

once in 1967, IBM 7094 used priority sch. - 2 process waited till 1973

AGING

method to prevent starvation
Gradually priority of processes is inc. after fixed intervals of time.

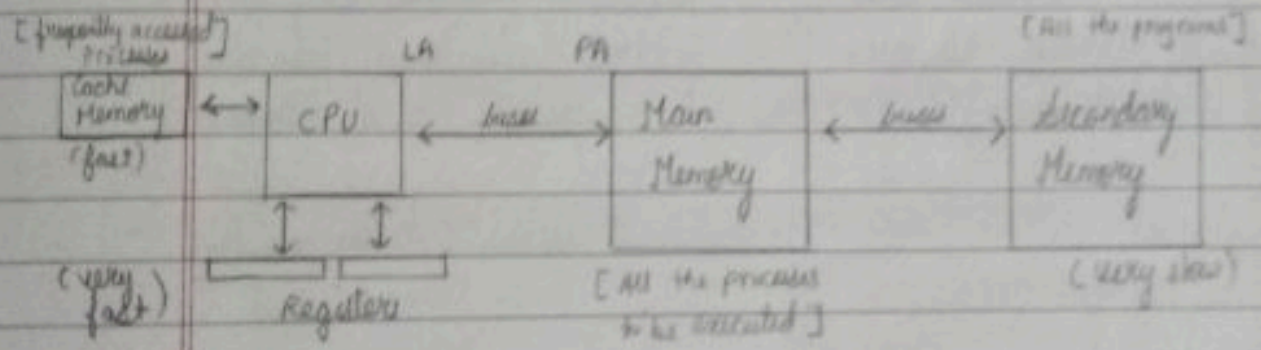
B+

MEMORY MANAGEMENT

- It is a functionality provided by operating system which manages the primary memory of computer system

Tasks
- Moves processes back & forth between main memory & disk
- keeps track of memory allocation / deallocation
- Security: prohibits user program to enter in OS area & other user area.

goals
- maximise memory utilization
- maximise CPU utilization
- security to user programs & OS



degree of multiprogramming
no. of processes that can be accommodated in main memory at a time for execution.

- degree of multiprog. ↑ , CPU utilization ↑
  depends on size of RAM

Example:

RAM size = 4MB Process size = 1MB
degree = 1 idle time = k (30%)
CPU utilization = $1-k$ (30%)

RAM size = 8MB Process size = 4MB
degree = 2 idle time = k^2
CPU utilization = $1-k^2$

RAM size = 16MB Process size = 4MB
degree = 4 idle time k^4
CPU utilization = $1-k^4$ (99%)

Tutorial

Dt _____ B+
Pg _____

Memory Allocation Techniques

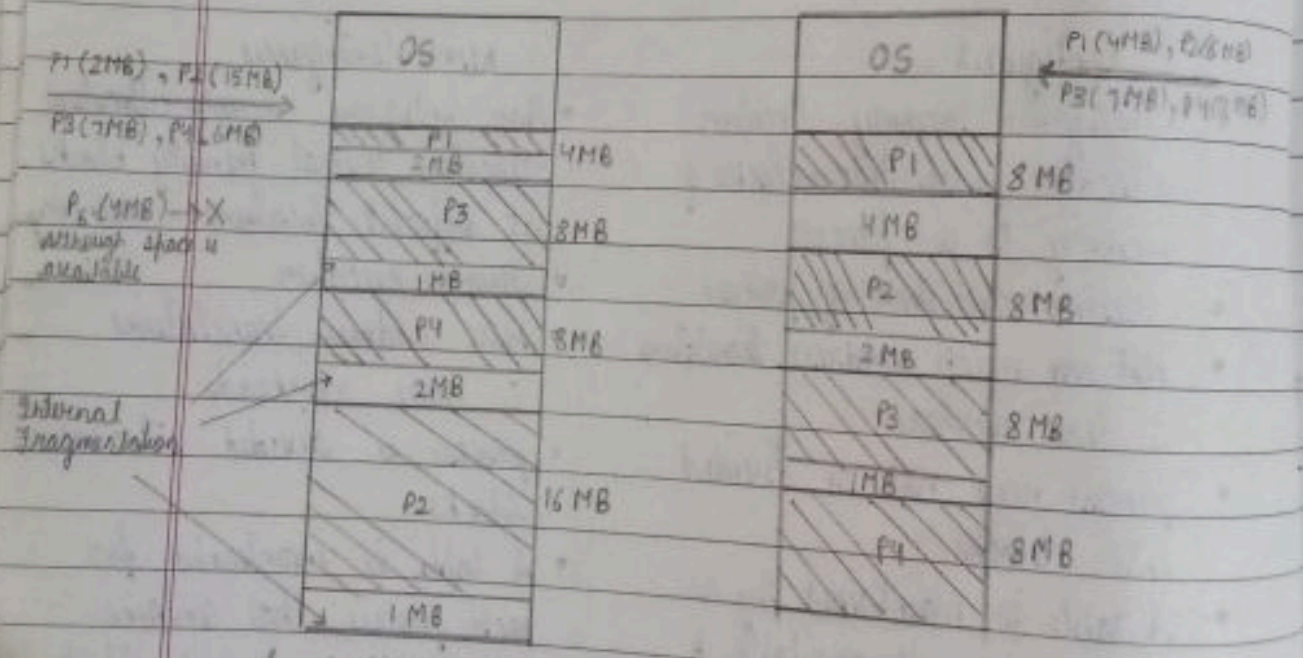
- Contiguous
  - Fixed partitioning
    - equal sized
    - variable sized
  - Dynamic partitioning
- Non Contiguous
  - Paging
  - Multilevel Paging
  - Inverted Paging
  - Segmentation
  - Segmented Paging

Contiguous	Non - Contiguous
• Contiguous memory allocation allocates consecutive blocks of memory to a process	• Non Contiguous memory allocation allocates several memory blocks at different locations in memory
• Faster execution of process	• slower execution
• Not ~~one~~ much address translations ∴ less overhead	• More address translations ∴ more overheads
• process need not be divided into blocks	• process is divided into various blocks
• A table is maintained by OS that contains all available & occupied partitions in memory.	• A table is maintained for each process that contains base addresses of each block acquired by the process in memory
• Wastage of memory	• less wastage

Fixed Partioning

no. of size of partitions is decided in advance before the process is loaded

- oldest & simplest techique
- Memory is divided into fixed - size partitions.
- No of partitions are fixed.
- Size of partition may or may not be same
- spanning is not allowed.
- A partition is allocated to a single process only
- No of partitions = degree of multiprogramming



Equal Size Partitioning

Unequal Size Partitioning

Internal Fragmentation (memory wastage)

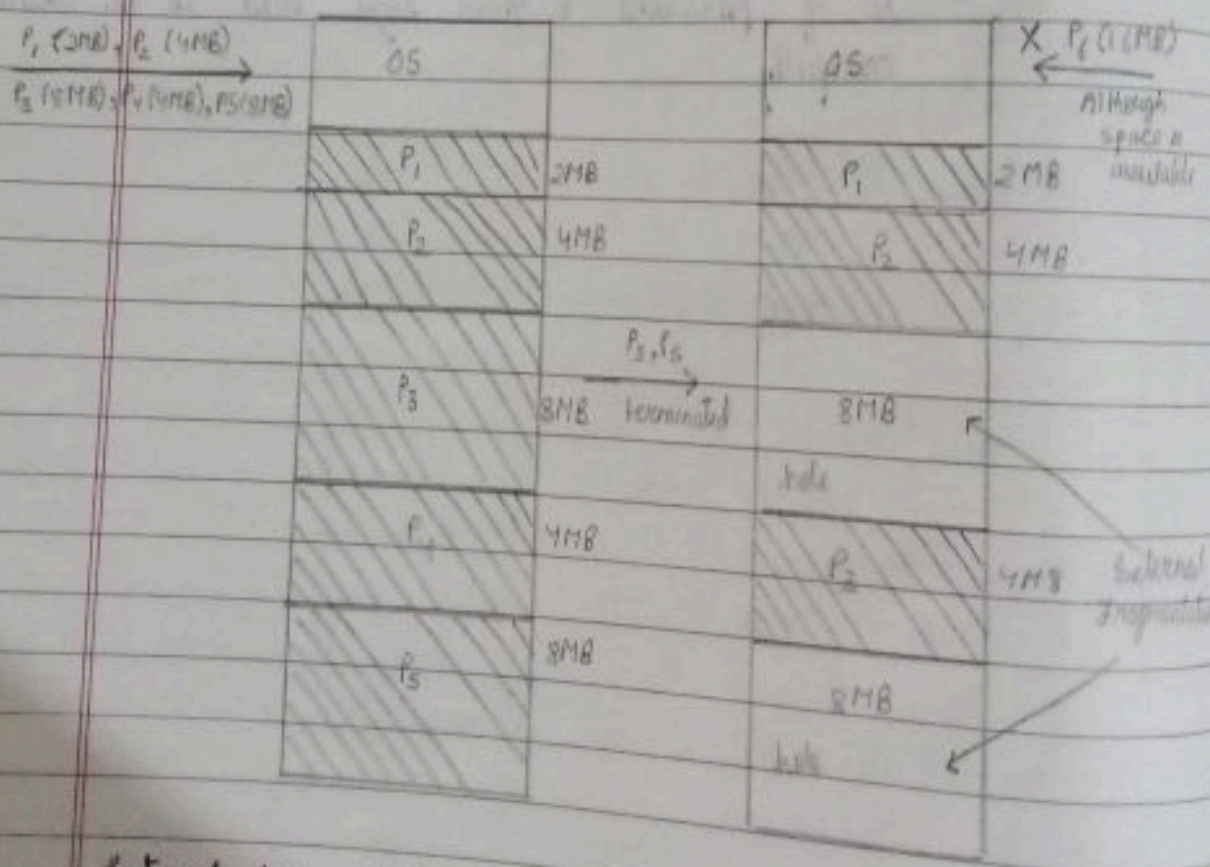
When the process is allocated to a memory block whose size is more than that of process & due to which some part of the memory is left unused.

Adv • Easy to implement

Disadv • Internal Fragmentation → external fragmentation
• Limitation to size of the process
(processes which are larger than the size of partitions are not allocated the memory)
• limitation to the degree of multiprogramming
(= no of partition → fixed)
• no of partitions & their sizes need to be decided carefully.

Dynamic Partitioning

- Initially RAM is not partitioned.
- Partitions are created at the time of process loading
- Size of partition is equal to the incoming process (this avoids internal fragmentation)
- No ~~of processes~~ or size of partitions is not fixed & depends on the size of incoming processes & size of main memory.



External fragmentation

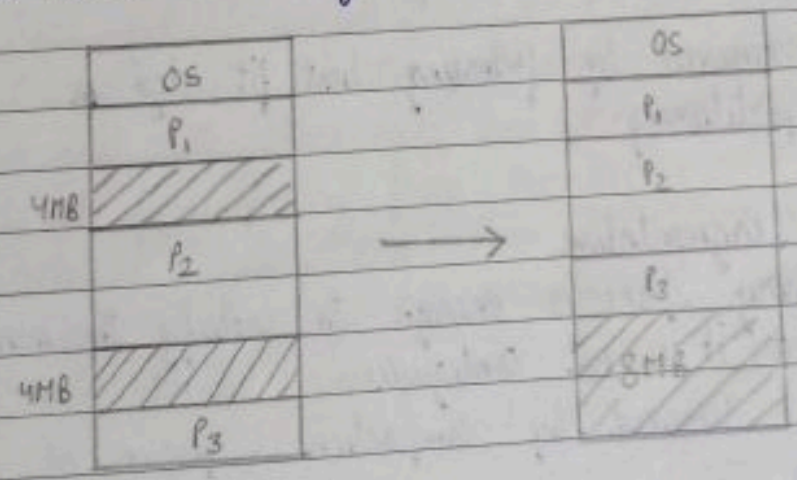
When the memory space is sufficient but can't be utilised because it is not contiguous.

holes are not in memory & their size ↓ over the time

Adv. • No internal fragmentation
(bcz partition size = process size)
• No limitation on size of processes
(bcz partitions are made dynamically in accordance with the size of process)
• No limitation on degree of multiprogramming
(bcz no of processes aren't fixed)

Disadv. • External fragmentation
• Allocation & deallocation of memory is complex

Compaction (defragmentation)
It is a process of combining all the free spaces together to make it contiguous & avoid external fragmentation



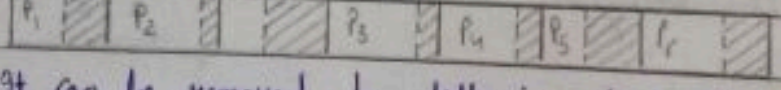
drawback: efficiency of the system is decreased. Huge amount of time is invested & CPU remains idle for this time.

Fragmentation

When processes are loaded & removed from the memory, the free memory space is broken into smaller pieces. Due to the small size, these memory fragments can't be allocated which leads to memory wastage. This is called fragmentation

→ Internal Fragmentation allocated space is wasted

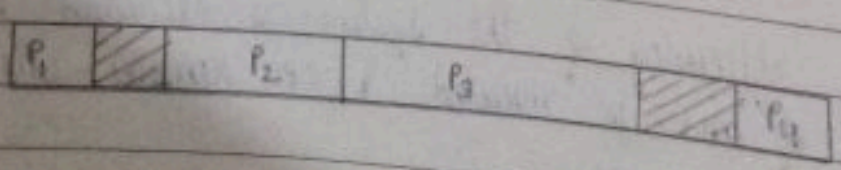
- When memory block assigned to a process is bigger than the process size, left over portion of the block remains unused.
- It usually happens when a block / partition is allocated to a single process & sharing is not allowed.



- It can be removed by following best fit algo or dynamic partitioning.

→ External Fragmentation unallocated space is wasted

- Total memory space is enough to satisfy the memory request but it is not contiguous.
- It can be removed by compaction, paging or segmentation



Dt _____ B+
Pg _____

Algorithms (for unequal fixed size / dynamic partitioning)

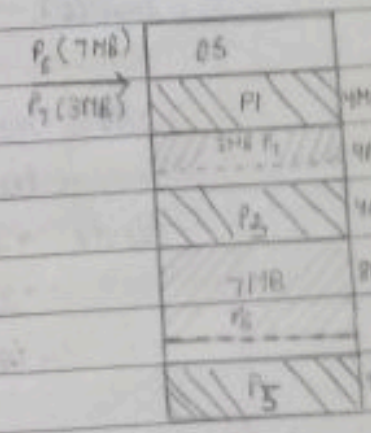
In fixed & dynamic partitioning, free partitions and holes are allocated processes based on some algorithms

1) First - Fit

Process is allocated the first available memory ~~with~~ (unallocated partition / hole) with space more than or equal to it's size

adv: fast

disadv: memory wastage (not efficient)

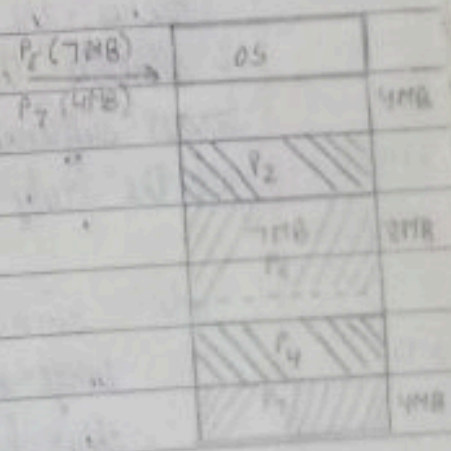


2) Next - Fit (modified - first - fit)

Process is allocated the first available memory with space more than or equal to its size, starting from the last allocated memory space

- uses a moving pointer to track
- last allocated memory space.

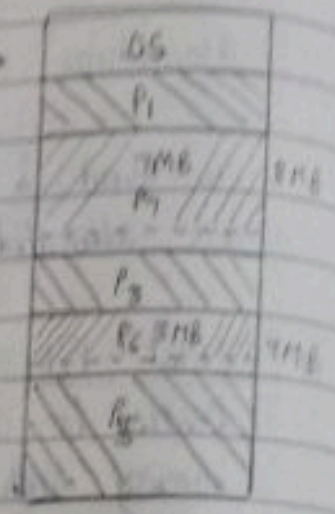
faster.



3> Best Fit

searches the whole memory acc to the size of process & allocates it to the closest-fitting free partition / hole in the memory

- keeps a list of free memory (smallest - largest)

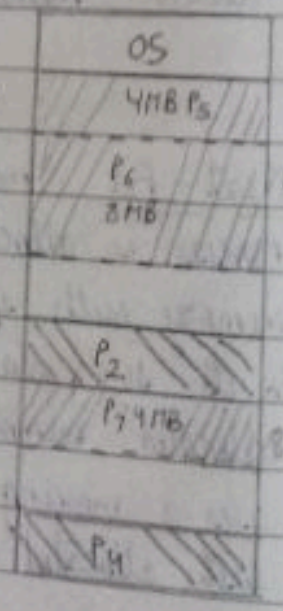


adv: • memory efficient
(less memory is wasted)

disadv: • slow
• reduces holes to much smaller size which can are very rarely occupied afterwards.

4> Worst Fit

searches the whole memory & allocates the process to largest sufficient partition among all the freely available partitions



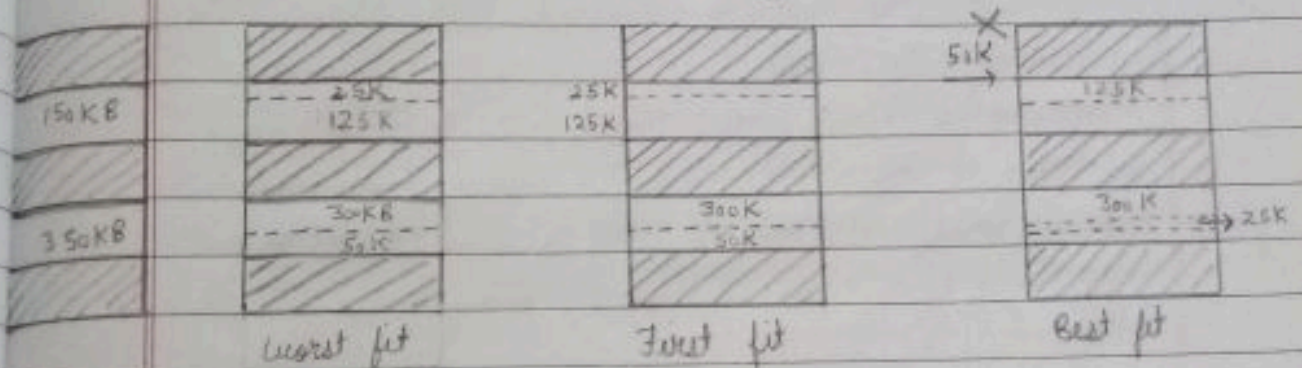
adv: remaining holes are large enough to accomodate other processes.

disadv: if large process comes later, it can't be accommodated.

Dt _____ Pg _____ B+

Questions :

→ 300K , 25K , 125K , 50K are requested respectively
Above requirement could be satisfied with
a) Best Fit only b) First fit only c) both d) None



Worst fit First fit Best fit

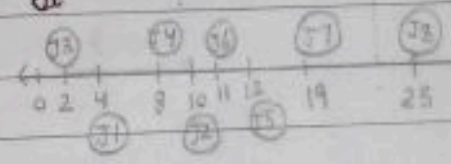
→

enter time	0	0	0	0	8	10	11	
Request No	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8
Request Size	2K	14K	3K	6K	6K	10K	7K	20K
Usage Time	4	10	2	8	4	1	8	6

3K-J_3	4K
6K-J_4 6K-J_5	8K
14K-J_2 10K-J_6 7K-J_7	20K
2K-J_1	2K

" best fit " overall time = 0

Calculate the time at which J_7 will be completed a) 17 b) 20 c) 19 d) 37



Tutorial

Dt _____
Pg _____ B+

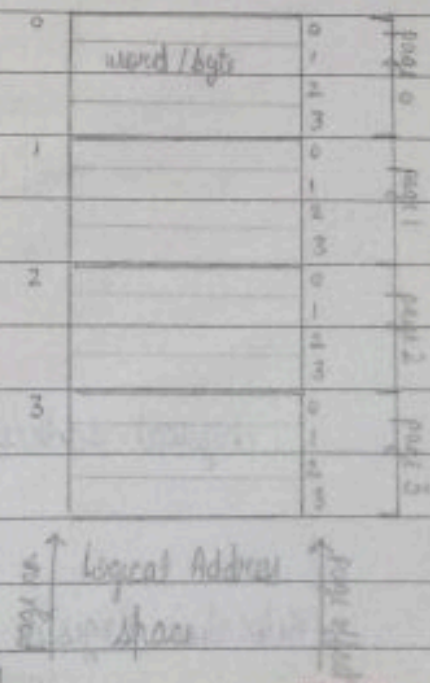
PAGING

- It is a non-contiguous memory allocation technique
- In this technique,
  - logical address space is divided into pages (fixed-size)
  - physical address space is divided into frames (fixed-size)
  - size of page = size of frame (equal sized)
    (for max memory utilisation & avoid fragmentation)
- A process is divided into pages & stored in frames in main memory
  (Pages can be stored at different locations in non-contiguous frames)

→ Logical Address space (in reference to the secondary memory)
- set of all the logical addresses generated by the CPU
- It is a virtual / Conceptual space which doesn't exist in memory

Logical address: It is a virtual address generated by CPU during program execution.
It is used as a reference to access the physical memory

- Logical address: | Page no. (p) | page offset (d) |
- For n bit page no., $n = 2$
  no. of pages = 2^n 4 pages
- For m bit page offset, $m = 2$
  no. of words / page = 2^m 4 words / page
- no. of bits in logical address =
  no. of bits in page no. + no. of bits in pg offset
  $= n + m$
  size of logical address space $= 2^{n+m}$ $2^{2+2} = 16$ words
  (total words)



p = LA/page size
d = LA % page size

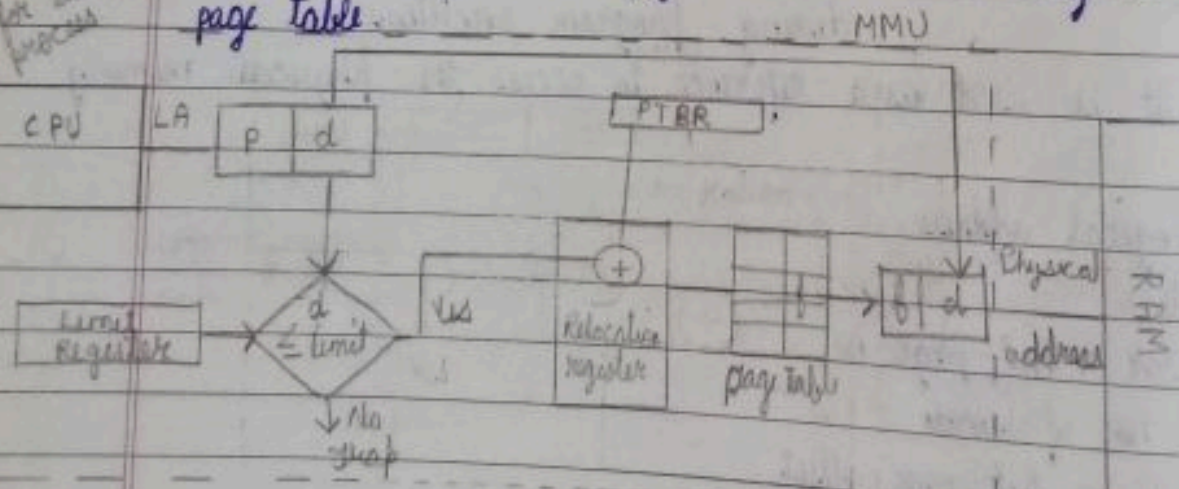
→ Physical Address Space (in reference to the main memory)
set of all the physical addresses corresponding to logical addresses in logical space
It actually exists in the memory

physical address | frame no. (f) | frame offset (d) |

Note: The user never directly deals with the physical address. CPU generates the logical address & program appears to runs on the logical addresses. But actually logical addresses are mapped to the corresponding physical addresses by MMU

MMU (Memory Management Unit)
It is a hardware device responsible for converting logical addresses into physical addresses using a page table.

A page table is maintained for each process



physical address = frame no * size of frame + offset
= f * n + d

Relocation Register: special register in CPU used to map logical addresses to physical addresses

Example

→ Logical Memory Space

0	a	0
	b	1
	c	2
	d	3
1	e	0
	f	1
	g	2
	h	3
2	i	0
	j	1
	k	2
	l	3

Page Table

Page No	Frame No
0	0
1	2
2	4

Physical Memory Space

0	0	a	0
	1	b	1
	2	c	2
	3	d	3
1	4		0
	5		1
	6		2
	7		3
2	8	e	0
	9	f	1
	10	g	2
	11	h	3
3	12		0
	13		1
	14		2
	15		3
4	16	i	0
	17	j	1
	18	k	2
	19	l	3

(Hint)

frame size = page size = 4 words

Memory size = 20 words

no. of frames = $\frac{20}{4} = 5$

logical address = 2 bits + 2 bits = 4 bits

physical address = 3 bits + 2 bits = 5 bits

process size = 12 words

no. of pages = $\frac{12}{4} = 3$

$(p, d) = (1, 2)$ [g] in logical space

$2 \leq 4$ ✓

$f = 2$ $(f, d) = (2, 2)$

physical address $= 2 * 4 + 2 = 10$ [g] in physical space

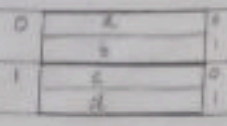
$(p, d) = \underbrace{(2, 0)}_{\text{logical add}} \rightarrow 0 \leq 4 \xrightarrow{\text{True}} \underbrace{(4, 0)}_{\text{physical add}} \rightarrow 4 * 4 + 0 = 16$

Hence Proved

$2^1 = 2$	$2^5 = 32$	$2^9 = 512$ $2^{40} = 1T$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1K$
$2^3 = 8$	$2^7 = 128$	$2^{20} = 1M$
$2^4 = 16$	$2^8 = 256$	$2^{30} = 1G$

Dt.
Pg. B

→



Logical address space

page no	frame no
0	2
1	4

page table

physical address space

frame size = page size = 2
memory size = 16
no. of frames 16/2 = 8
process size = 4
no of pages = 4/2 = 2

1bit	1bit		3bits	1bit
1	1	→	1 0 0	1
(2 → d)			(9 → d)	

0	1	→	0 1 0	1
(1 → b)			(5 → b)	

logical address = 1 bit + 1 bit
2 bits

Physical address = 3 bit + 1 bit
= 4 bits

→ LAS = 4 GB
PAS = 64MB
Page size = 4KB

no of pages? no of frames? size of LA? PA?
no. of entries in page table?
size of page table?

LA

20B	12B

PA

14B	12B

no. of pages = $\frac{\text{logical address space}}{\text{page size}} = \frac{4GB}{4KB} = \frac{2^{2+30}B}{2^{2+10}B} = 2^{20}$

no of bits req. for page no = 20 bits

no. of frames = $\frac{\text{physical address space}}{\text{frame size}} = \frac{64MB}{4KB} = \frac{2^{6+20}B}{2^{2+10}B} = 2^{14}$

no. of bits req for frame no = 14 bits

no. of bits req for logical address ⇒ $4GB = 2^{32}$ 32 bits

or bits for page no. + bits for offset, $4KB = 2^{12}$ 12 bits

20 + 12 = 32 bits

Tutorial

assumption - each word = 1 byte

size of physical address => 64MB = 2^{26} 26 bits

or bits for frame no + bits of offset

14 + 12 = 26 bits

no. of entries in page table = no. of pages = 2^{20}

size of page table = no. of pages x size of each entry

(2^{20} x 14) bits

↳ page table stores frame no.

→ LA = 7 bits PA = 6 bits page size = 8 words / byte

no. of pages? no. of frames? size of page table?

no. of bits req. for offset ⇒ 8 = 2^{3} 3 bits

no. of bits req. for page no. = size of LA - size of offset

= 7 - 3 = 4

LA : | 4B | 3B |

no. of bits req. for frame no. = size of PA - size of offset

6 - 3 = 3

PA : | 3B | 3B |

no. of pages = 2^4 = 16 pages

no. of frames = 2^3 = 8 frames

size of page table = no. of entries x size of an entry

= no. of pages x size of frame no.

= 16 x 3 = 48

size of LAS = 2^7 = 128 Bytes / words

size of PAS = 2^6 = 64 Bytes / words

Page Table

index denotes pg no

- It is used to keep all the information about the pages
- It's base address is stored in PTBR (Page Table Base Register)

Page table entries:

Frame No.	Valid/ Invalid (+ve / -ve)	Protection (RWX)	Reference (0/1)	Caching	Dirty (Modify)

optional

- Frame no: frame no. corresponding to every page. no. of bits depend on no. of frames.
- Valid / Invalid bit: 0 - if present, 1 - if absent. If a page is absent → page fault (It is used to support virtual memory)
- Protection bit: R - Read, W - Write, X - execute
- Referenced bit: 0 - if page is not accessed in last clk cycle. 1 - if accessed. (used in page replacement algo LRU)
- Caching: disabled - if page changes frequently & fresh page is req. (cache store old info.) enabled - if caching is required
- Dirty bit: 1 - if modified (i.e. write - access), 0 - if not modified (So that changes can be saved when page is evicted out)

Tutorial

Locality of ref. - Tendency of a program to access same set of instr multiple times or the instr whose address is near from
→ demand paging TLB

Dt _____
Pg _____ B+

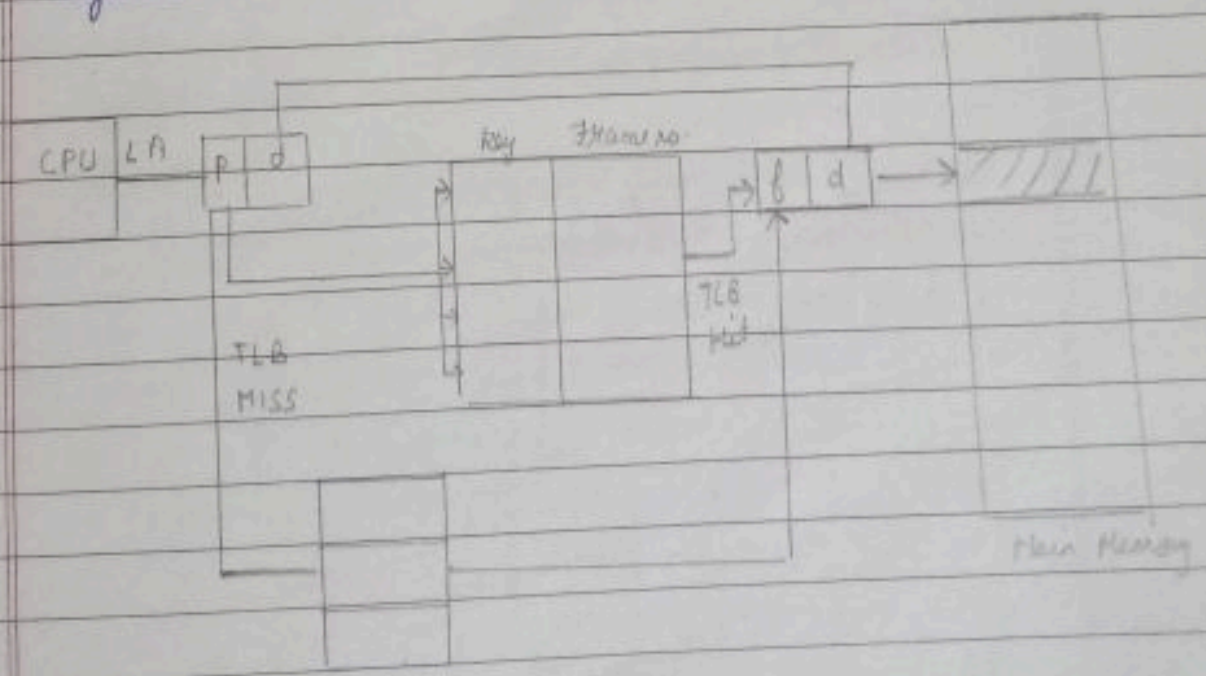
TLB
(Translation Lookaside buffer)

- It is a small sized, high speed memory cache which stores the frequently / recently accessed page entries in order to reduce the search time.

PID + Address space Identifier

It consists of
- key → Process ID + Page no
- value → frame no corresponding to the page

→ First page entry is searched in the TLB
→ If it is found → TLB HIT, frame no. is determined
→ If it is not found → TLB MISS, page no along with the process ID & the frame no. are entered in TLB
→ If TLB does not contain empty space, page replacement algorithms are used



Tutorial

MULTILEVEL PAGING

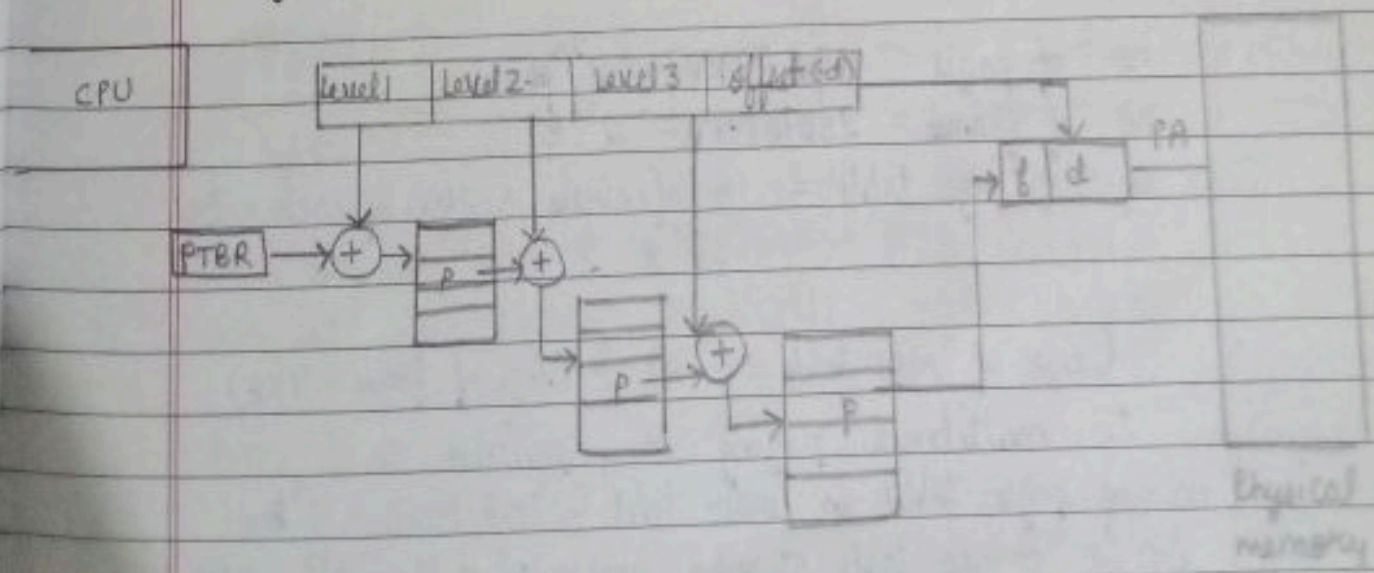
- It is a paging technique in which multiple levels of page tables are used (bcz size of page table > size of a frame ∴ page table too needs to be divided)
- The last level page tables contains actual frame info
- First level contains only 1 page table & its address is stored in PTBR
- One access to each level is required to find the actual frame.
- Each page table (except the last level) contains frame no. of next level page tables

Logical address:

Page no.			
Level 1	Level 2 - -	- - - Level n	offset

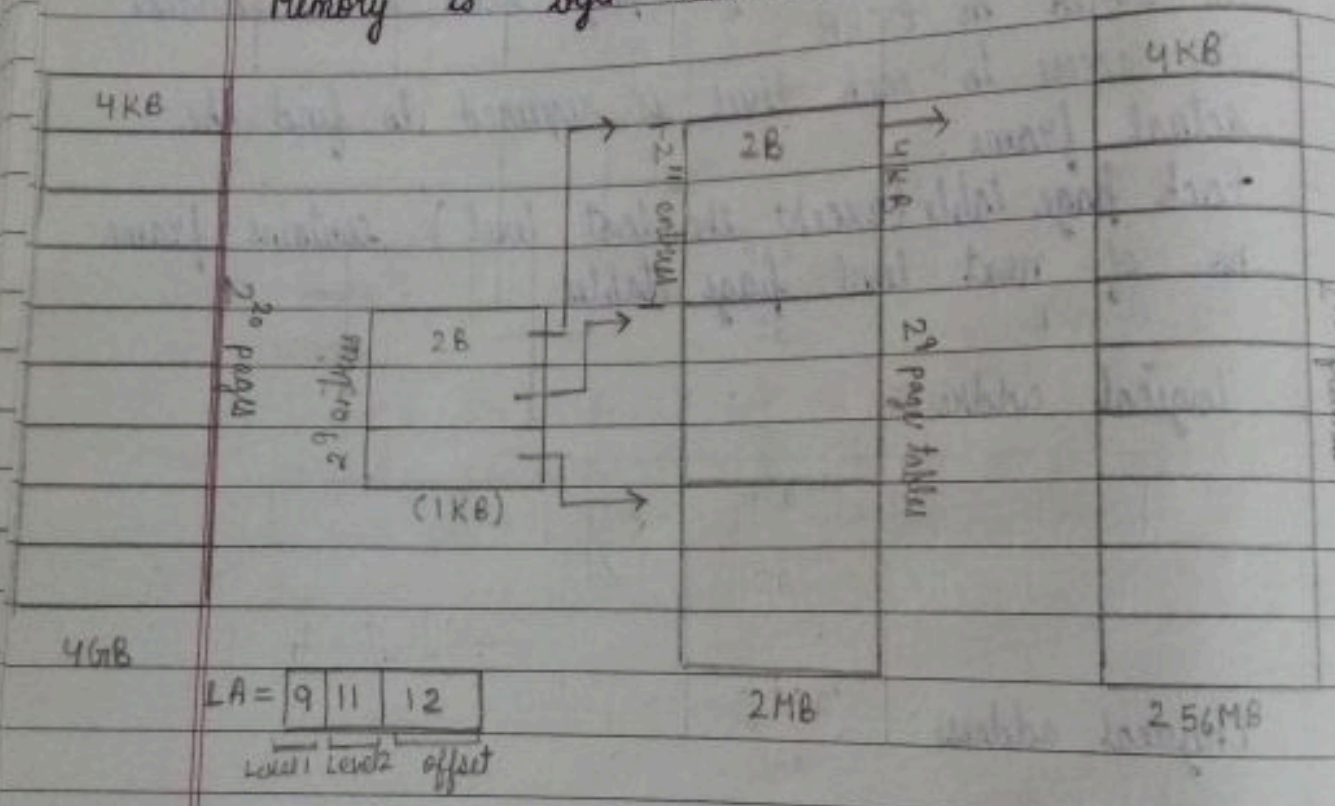
Physical address:

frame no	offset



Example :

Physical address space = 256 MB 2^{28} B
Logical address space = 4 GB 2^{32} B
Frame size = 4 KB 2^{12} B
Page table entry = 2 B
Memory is byte addressable



no. of pages = 4GB / 4KB = 2^{20} B

no. of frames = 256MB / 4KB = 2^{16} B

20 bits	12 bits	LA
16 bits	12 bits	PA

→ size of page table = no. of entries × size of each entry

$$\underbrace{2^{20}}_{\text{no of pages}} \times \underbrace{2}_{\text{no of bits for frame no.}} = 2^{21} = 2MB$$

(size of page table = 2MB) > (size of frame = 4KB)

∴ multilevel paging is required

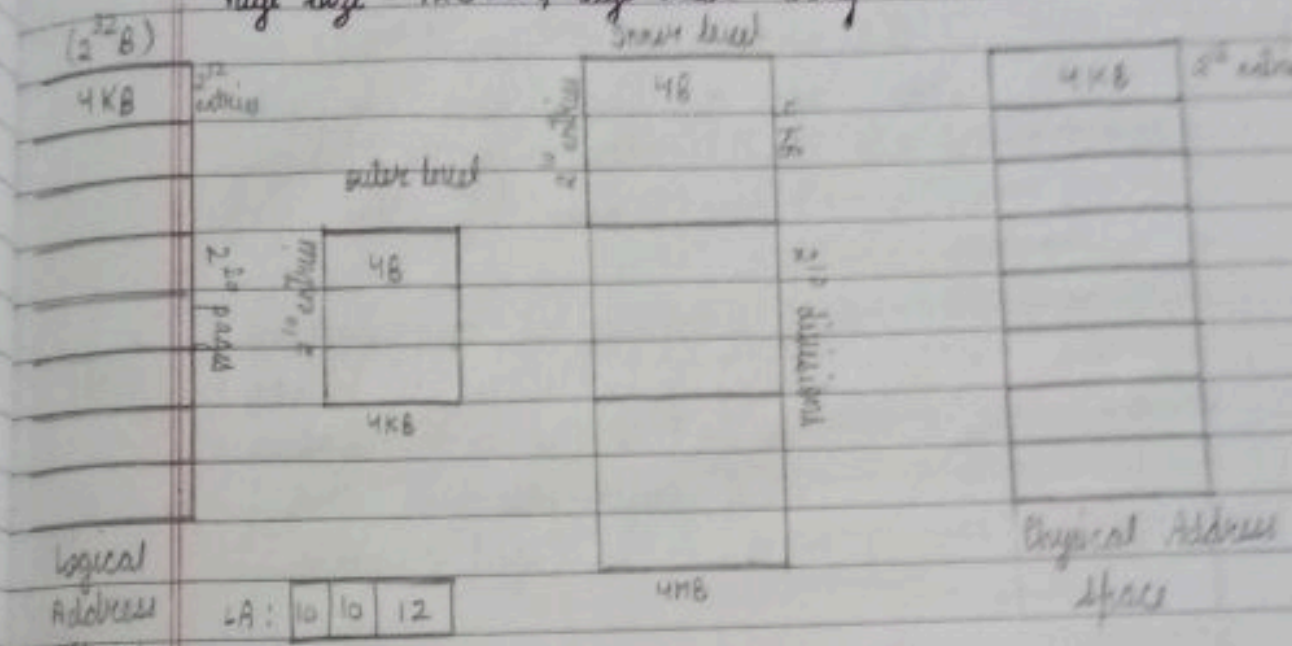
→ no. of pages tables in inner level = 2MB / 4KB = 2^9

size of page table at outer level = $2^9 \times 2 = 2^{10} = 1KB$

1KB < 4KB ∴ no further level required

Example LA = 32 bit

Page size = 4KB 2^{12}B, Page table entry = 4B 2^2B



LA: | 10 | 10 | 12 |

No of pages = (size of logical memory) / (size of a page)

$= 2^{32} / 2^{12} = 2^{20}$ pages

| 20 | 12 |

Inner level

no. of entries in page table = 2^{20}

size of each entry = 4B

∴ size of inner page table = $2^{20} \times 2^2 = 2^{22}$ B = 4MB

4MB > 4KB i.e. (size of page table > frame size)

∴ inner page table needs to be divided

No. of divisions of page table = 4MB/4KB = 2^{10}

Outer level

no. of entries = 2^{10}, size of each entry = 4B

∴ size of outer page table = $2^{10} \times 4B = 2^{12}$ B = 4KB

4KB = 4KB i.e. (size of outer page table) = (size of frame)

Rarely used due to ↑ search time

Dt. _____ Pg. _____ B+

INVERTED PAGING

In this paging technique, an inverted page table / frame table is maintained for the frames present in physical memory rather than maintaining separate page tables for each process.

Inverted page table / frame table
- no of entries = no of frames
- contains page no & page id (both combined refers to a unique page)

frame	Page No	Page Id
0	–	–
1	–	–
2	–	–
3	–	–

0	f_0
1	X
2	f_2
3	X

Page table of P_1

0	X
1	f_3
2	f_5
3	X

Page table of P_3

0	X
1	f_1
2	X
3	f_4

Page table of P_2

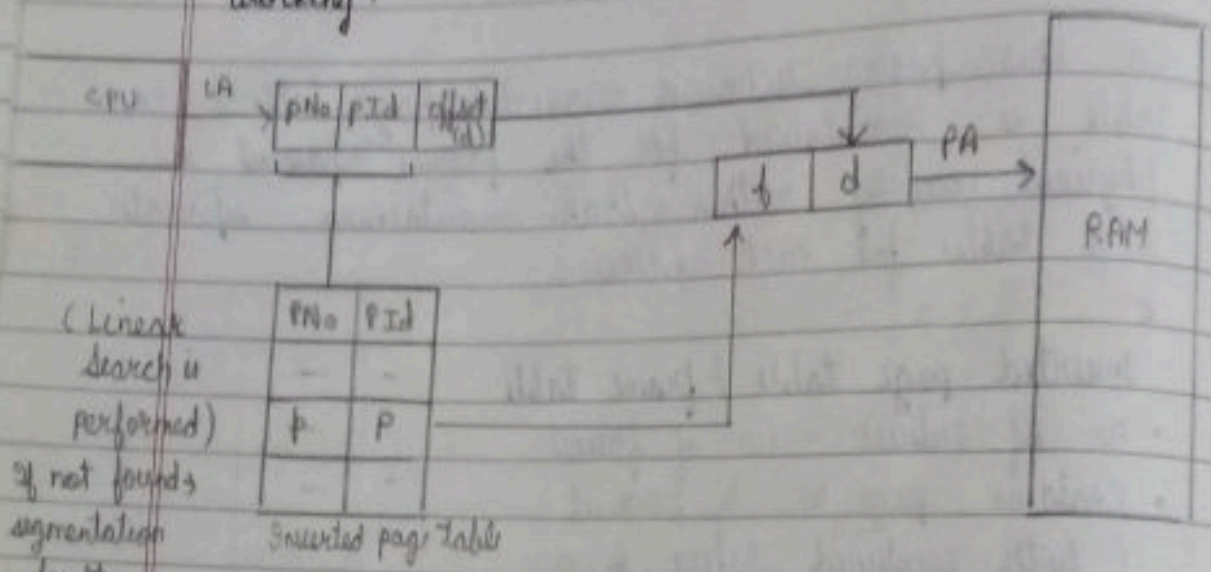
f No	Page No	Page Id
0	P_0	P_1
1	P_1	P_2
2	P_2	P_1
3	P_1	P_3
4	P_3	P_2
5	P_2	P_3

frame table / inverted page table

Paging	Inverted paging
→ Each process has its own P.T	→ Only one ~~page~~ inverted PT is mt in the memory.
→ If pages of n processes are mt in memory, then n P.T are mt in the memory (memory overhead)	→ Linear search is performed to find the frame no. (search time ↑)

Tutorial

Working :



(Linear search is performed)
If not found, segmentation fault

Advantages	Disadvantages
reduced memory space	search time ↑

Advantages of Paging	Disadvantages of Paging
- No external fragmentation - Swapping of pages is easy as pages are of same size. - Efficient use of memory	- Access time is high (LA → PA) - Internal fragmentation in last page of the process. - Large memory space is required as page tables are also stored

Dt. _____ B+
Pg. _____

VIRTUAL MEMORY

It is a memory allocation scheme in which secondary memory can be used as ~~main~~ a part of main memory at the times of shortage of RAM

- All memory references must be logical addresses and must be dynamically translated into physical addresses at run time. [∴ parts of the process can change locations during execution]

- Process must be divided into pieces.

In this scheme,

→ Only required parts of the process is loaded into the main memory & rest of the parts are stored in a ~~pa~~ secondary memory & fetched when required.

→ This section of secondary memory is termed as virtual memory.

→ It gives an illusion to the user that complete processes are loaded into the main memory and therefore it is large enough but in actual, real memory is limited.

Features:

- Processes which are larger than RAM can be executed
- Reduces external fragmentation (As entire process need not be loaded at the same time)
- The amount of space used by the process (in real memory) in main memory can be varied during execution
- Location of various processes may be varied during execution
- Degree of multiprogramming increases.
- It is implemented by 'Demand Paging'
- It can also be implemented by 'segmentation' & 'paged segmentation' techniques but it is quite complex.

Demand Paging [paging concept + virtual memory concept]

- The process of loading a page into the main memory when it is demanded. Lazy Swapper
- A page may or may not be present in the memory when it is demanded.
  - If the page is present in main memory, it is accessed (page hit)
  - If page is not present in main memory, ~~page~~ interrupt occurs & control is shifted from user to OS - Page Fault

FMT

Page Map Table (PMT)

- It contains frame no, valid / invalid bit
- If page is present in main memory, corresponding frame no. is entered & valid bit = 1
- If page is not present, frame no is empty & valid bit = 0

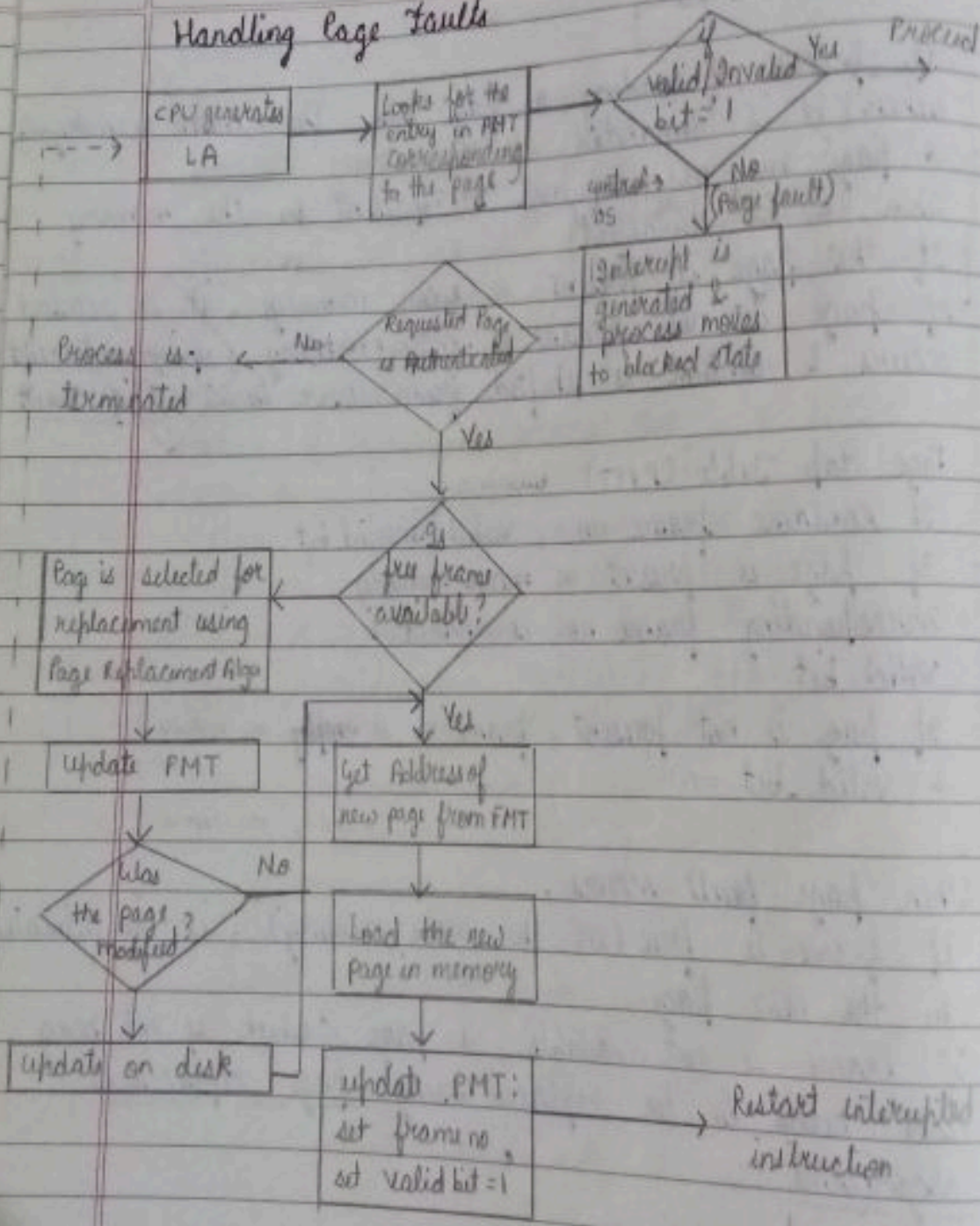
	fno	v/i
0	X	0
1	3	1
2	X	0
3	?	1

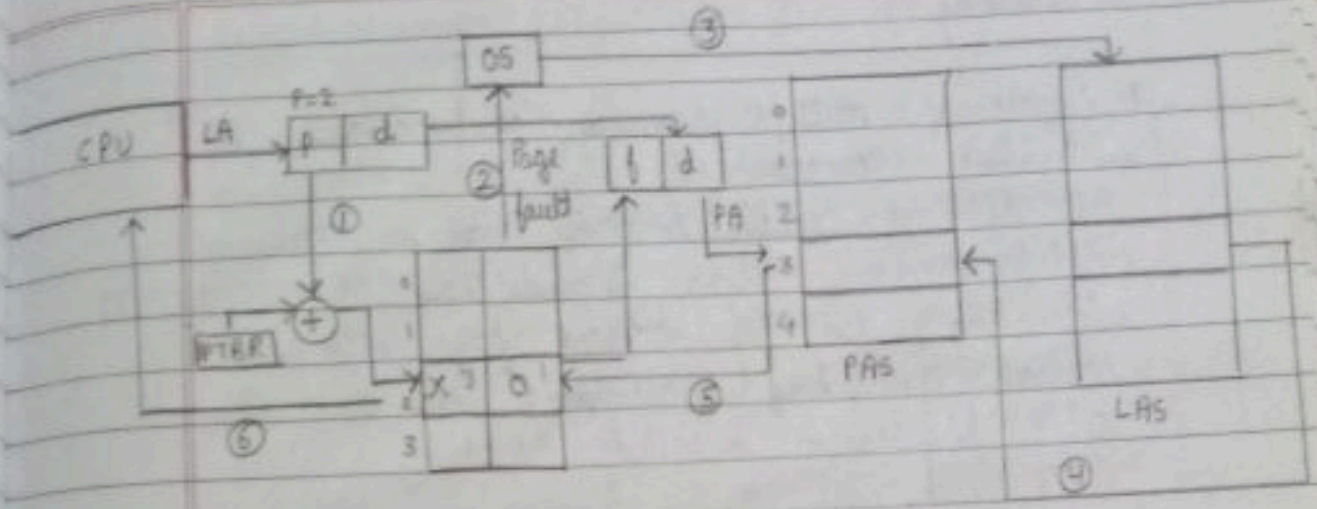
PMT

When page fault occurs,

- if frame is free (in the main memory), it is allocated to the new page.
- if frame is not available, a page (which is not being used) needs to be replaced using page replacement algorithms

Handling Page Faults





Effective Memory Access Time (EMAT)

page fault rate $= p$ $[0 \le p \le 1]$
(probability of page fault occurance)
page fault service time $= s$ (usually in millisec)
(time taken to handle page fault)
memory access time $= m$ (usually in nanosec)
(time taken to access the memory)

$$EMAT = p*(s+m) + (1-p)*m$$
$$= p*s + (1-p)*m \qquad [\because p\,m << s]$$

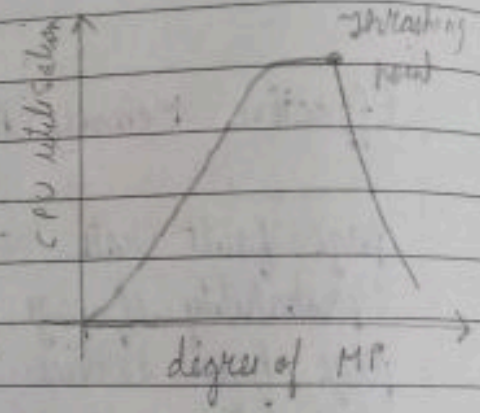
Example: MM Access time = 100 μ sec, Page fault rate = 40%, s = 4 m sec

$$EMAT = 0.4(4\times10^{-3}) + (0.6)(100\times10^{-6})$$
$$= 1.6\times10^{-3} + 0.6\times10^{-4}$$
$$= 16\times10^{-3} + 0.06\times10^{-3} = 1.66\times10^{-3}\text{ sec}$$
$$= 1660\ \mu\text{sec}$$

Dt
Pg

Thrashing

* Degree of multiprogramming is increased, CPU utilisation increases
But after a certain limit, inc in degree of multiprogramming leads to decrease in CPU utilisation due to increased page faults, context switching, swapping etc.
This phenomenon is called thrashing.



Cause As degree of MP is ↑, no of processes in MM ↑. As size of MM is fixed ∴ frames corresponding to the processes decrease. This leads to increased page faults. When page fault occurs frequently, more time is spent on swapping, page replacement, context switching etc & CPU remains idle for most of the time. Hence CPU utilisation ↓.

Solution - working set Model.

Page Replacement Algorithms

Algorithms that are used to decide which page needs to be replaced by the new page that is demanded, in case page fault occurs

1) FIFO (First In First Out)

The page that comes first in the main memory, is replaced first

OS maintains a queue for all the pages, front → oldest page.

Example:

Reference string	3	2	1	3	4	1	6	2	4	3	4	2	1	4	5	2	1	3	4
f_1	3	3	3	3	4	4	4	4	4	3	3	3	3	3	5	5	5	5	4
f_2		2	2	2	2	2	6	6	6	6	4	4	4	4	4	2	2	2	2
f_3			1	1	1	1	1	2	2	2	2	2	1	1	1	1	1	3	3
	M	M	M	H	M	H	M	M	H	M	M	H	M	H	M	M	H	M	M

no. of hits = 6 hit ratio = (6/19) × 100

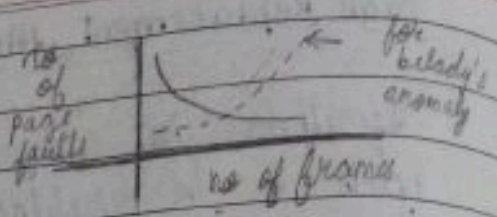
no. of miss = 13 miss ratio = (13/19) × 100

Reference string	7	0	1	2	0	3	0	4	2	3	0	3	1	2	0
f_1			1	1	1	1	0	0	0	3	3	3	3	2	2
f_2		0	0	0	0	3	3	3	2	2	2	2	1	1	1
f_3	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
	×	×	×	×	✓	×	×	×	×	×	×	✓	×	×	✓

no of hits = 3 hit ratio = 3/15 = 1/5 = 20%

miss = 12 miss ratio = 12/15 = 4/5 = 80%

Belady's Anomaly :
(in FIFO)



For some reference strings no of page faults increase with increase in no of frames. This is known as Belady's Anomaly.

Example :

Reference string	1	2	3	4	1	2	5	1	2	3	4	5
f_3			3	3	3	2	2	2	2	2	4	4
f_2		2	2	2	1	1	1	1	1	3	3	3
f_1	1	1	1	4	4	4	5	5	5	5	5	5
	X	X	X	X	X	X	X	✓	✓	X	X	✓

Hits = 3
Miss = 9

Reference string	1	2	3	4	1	2	5	1	2	3	4	5
f_4				4	4	4	4	4	4	3	3	3
f_3			3	3	3	3	3	3	2	2	2	2
f_2		2	2	2	2	2	2	1	1	1	1	5
f_1	1	1	1	1	1	1	5	5	5	5	4	4
	X	X	X	X	✓	✓	X	X	X	X	X	X

Hits = 2
Miss = 10

Last replaced page is the one demanded next

ucz future requests are not know
practically not possible, acts a theoretical benchmark for other algos B+

2) Optimal Page Replacement
(Best) for all ref. str

The page which is not used for longest period of time in the future is replaced.

Example:

Reference String	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
f_4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f_3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
f_2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f_1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	X	X	X	X	✓	X	✓	X	✓	✓	✓	✓	✓	X	✓	✓	✓	X	✓	✓

page hits = 12 page hit ratio = 12/20 = 3/5
page miss = 8 page miss ratio = 8/20 = 2/5

Tutorial

3) Least Recently Used (LRU)

Idea

The page that is least recently used is replaced

page that is not used for long time

Example:

Reference String	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	X	X	X	X	✓	X	✓	X	✓	✓	✓	✓	✓	X	✓	✓	✓	X	✓	✓

page hit = 12 hit ratio = 12/20 = 3/5

page miss = 8 miss ratio = 8/20 = 2/5

4) Most Recently Used (MRU)

The page that is most recently used is replaced

Reference String	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
f4				2	2	2	2	2	2	3	0	3	2	2	2	0	0	0	0	0
f3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2		0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4	4
f1	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	X	X	X	X	✓	X	X	X	✓	X	X	X	X	✓	✓	X	✓	✓	✓	✓

page hit = 8 hit ratio = 8/20 = 2/5

page miss = 12 miss ratio = 12/20 = 3/5

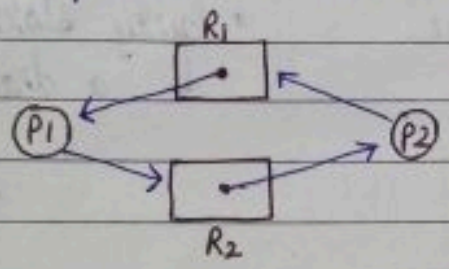
Dt
Pg B+

DEADLOCK

It is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other waiting process

→ waiting time is infinite and no progress at all (for all the processes present in deadlock)

example :



Coffman Conditions

(necessary conditions for deadlock to occur) not sufficient

1) Mutual Exclusion Atleast one resource must be non-shareable

2) Hold & Wait A process ~~can~~ holds a resource & waits for the other resources held by some waiting process.

3) No preemption Resources can't be taken from a process unless it releases the resource.

4) Circular wait set of processes waits for each other in a circular form

Tutorial

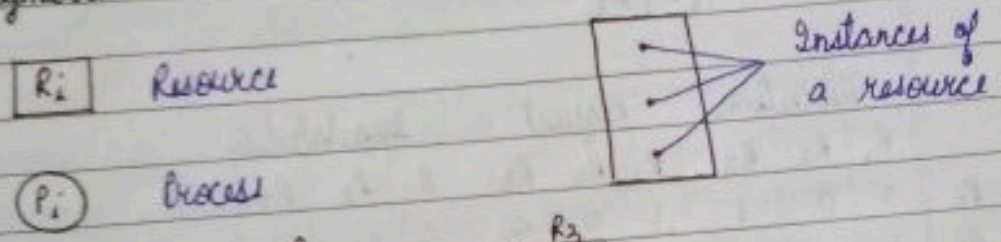
Deadlock	Starvation
• no process proceeds	• high priority processes proceed while LPP are blocked
• infinite waiting	• long time waiting, not ∞
• All the resources not in deadlock are held by waiting processes & are not being used.	• Resources are ~~used~~ being used continuously by high priority processes
• All deadlocks are starvations	• Every starvation need not be a deadlock

Resource Allocation Graph

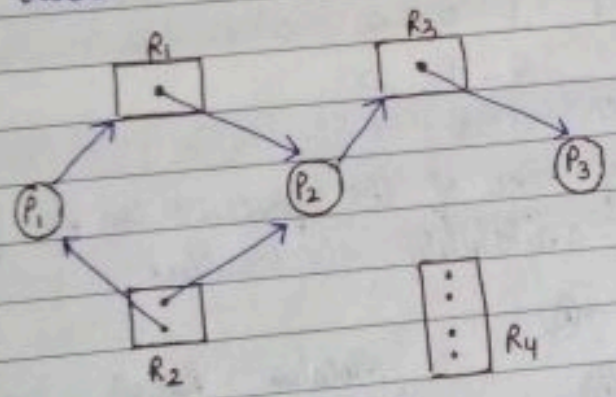
It is a set of

- Vertices → Processes $\{P_1, P_2, P_3, \ldots, P_n\}$
  - → Resources $\{R_1, R_2, R_3, \ldots, R_n\}$
- Edges → $P_i \longrightarrow R_j$ (Request Edge)
  - → $R_j \longrightarrow P_i$ (Assignment Edge)

Symbols:



Example:



Here, $P = \{P_1, P_2, P_3\}$

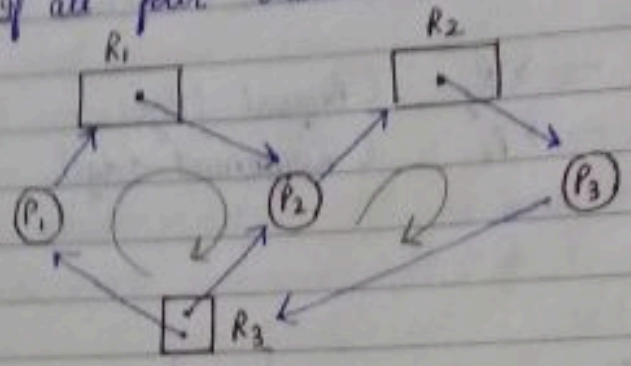
$R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3,$ Request edge

$R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$ Assignment edge

Detection of Deadlock

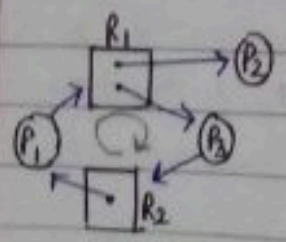
If any of the four coffman conditions is not fulfilled, deadlock doesn't occur
If all four conditions hold true, deadlock MAY occur

→ 

First check for cycle
If cycle exists, check whether any process has progress

	Allocation			Request			Availability		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	1	1	0	0	0	0	0
P_2	1	0	1	0	1	0			
P_3	0	1	0	0	0	1			

Request of none of the processes can be fulfilled based on the availability ∴ Deadlock

→ 

	Allocation		Request		Availability	
	R_1	R_2	R_1	R_2	R_1	R_2
✓ P_1	0	1	1	0	0	0
	1	1				
✓ P_2	1	0	0	0	1	0
✓ P_3	1	0	0	1	0	0
	1	1			1	1
					1	0 → 2 1

★ If cycle exists & each resource have only one instance, deadlock occurs for sure but if it has multiple instances, deadlock may occur

Deadlock Handling

i) Deadlock Prevention
ii) Deadlock Avoidance
iii) Deadlock detection & Recovery
iv) Deadlock ignorance (ostrich method)

1) Deadlock Prevention
deadlock can be prevented by violating any of the 4 necessary conditions

→ Removal of mutual exclusion
- shared resources must be used.
- can't be implemented

→ Removal of hold and wait either hold or wait, not both
Three ways:
- A process must acquire all the necessary resources before execution starts (not implementable) - inefficient
- If a process is holding some resources & requesting for additional resource, then it must release the acquired resources first (may lead to starvation)
- A process holding some resources must wait for another resource for a limited time only after which it must release all the resources held by it.

→ Removal of No-Preemption (used for system processes)

Two ways:

- If a process is holding some resources & requesting for another resource that can't be immediately allocated then all the acquired resources will be preempted.

- If a process is holding some resources & requests a resource.
  - If resource is available (allocated)
  - not available (resources ~~held~~ allocated to some other process that is waiting) - preempted from the waiting process & allocated to it.

→ Removal of Circular Wait

To request resources (implementable)

All the resources are associated with a no

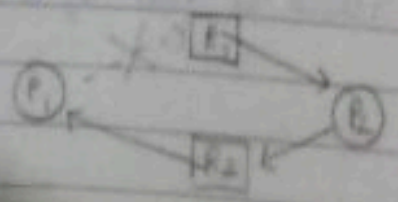
To request resources R_j, a process must first release all R_i s.t. $i >= j$

All the resources are requested & acquired in order.

eg: Printer (1), CPU (5), Memory (6), CD drive (7)

P_1 : 1, 6 $\xrightarrow{\text{Request(7)}}$ 1 $\longrightarrow$ 1, 5 $\longrightarrow$ 1, 5, 6

(can acquire/ request for 7) (release 6) (acquire 5) (acquire 6)



cycle never forms

process info like
- no of res allocated to a process
- max res required by a process
- order in which resources are required

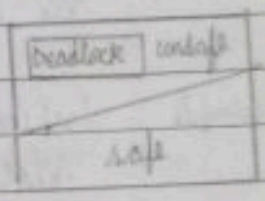
2) Deadlock Avoidance

→ System maintains some database using which it can take decision whether to entertain a request or not s.t it remains in safe state

→ System (kernel) analyse the database to determine whether granting a request can lead to deadlock in future

[if not, then request is granted
otherwise keep pending until they can be granted
(process may face long delay for obtaining a res)

Safe state: if system can allocate all the resources requested by all the processes without entering into deadlock otherwise → unsafe state (cycle not)



Initially system is in safe state. A request

→ Deadlock avoidance using RAG (for single instance)

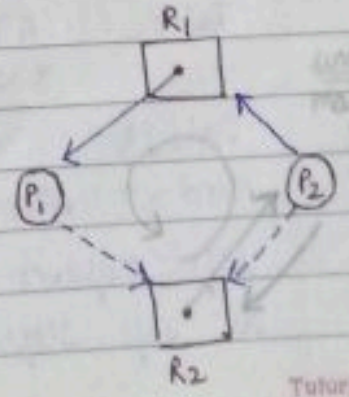
- $P_i \rightarrow R_j$ (request edge)
- $R_j \rightarrow P_i$ (assignment edge)
- $P_i \dashrightarrow R_j$ P_i may request R_j in future (claim edge)

if granted

after completion

Resources must be claimed in advance So, in the beg. all edges in RAG are claim edges

Example: If P_i requests R_j then request edge can only be converted into assignment edge if it doesn't form a cycle
Here, request of P_2 can't be granted



unsafe state may enter in deadlock

Tutorial

For safe state, atleast one safe sequence must exist.

Banker's Algorithm
used when resources have multiple instances

It requires:

- Max requirement of instances of each resource by each process. (2D array)
- Instances of each resource held by each process (2D array)
- Total instances of each resource available in the system (1D array)
- More instances required by each process (2D array)

Example: Snapshot at T_0

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
✓ P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
✓ P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
✓ P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
✓ P_3	0	6	3	2	0	6	5	2	2	14	12	8	0	0	2	0
✓ P_4	0	0	1	4	0	6	5	6	2	14	12	12	0	6	4	2
	2	3	10	12					3	14	12	12				

Need matrix? Available res.?

Is system in safe state? If yes find safe seq.

Total :

A	B	C	D
3	14	12	12

Available = Total - allocated

Need = ~~Allo~~ Max - allocated

safe sequence : P_0 P_2 P_3 P_4 P_1

As safe sequence exists ∴ It is in safe state

Example :

Snapshot at time T_0

	Allocation				Max				Available				Total			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2	3	3	2	1	12	12	8	10
P_1	3	1	2	1	5	2	5	2								
P_2	2	1	0	3	2	3	1	6								
P_3	1	3	1	2	1	4	2	4								
P_4	1	4	3	2	3	6	6	5								
	9	9	6	9												

Available Resource instances = Total − Total (Allocation)
Need = Max − Allocation

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
									3	3	2	1	2	2	1	1
✓ P_0	2	0	0	1	4	2	1	2	5	3	2	2	2	1	3	1
✓ P_1	3	1	2	1	5	2	5	2	6	6	3	4*	0	2	1	3
✓ P_2	2	1	0	3	2	3	1	6	7	10	6	6*	0	1	1	2
✓ P_3	1	3	1	2	1	4	2	4	10	11	8	7*	2	2	3	3
✓ P_4	1	4	3	2	3	6	6	5	12	12	8	10*				

safe sequence : $P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$
As safe sequence exists, system is in safe state

If request from P_1 arrives for (1,1,0,0), Can this request be granted immediately?

Request must not exceed the need
$(1,1,0,0) \leq (2,1,3,1)$

Check if Request $\leq$ Available
$(1,1,0,0) \leq (3,3,2,1) \Rightarrow$ True

$\therefore$ System pretends to grant the request & checks whether system remains in the safe state after granting the request.

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
✓ P_0	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
✓ P_1	4	2	2	1	5	2	5	2	4	2	2	2	1	0	3	1
✓ P_2	2	1	0	3	2	3	1	6	5	5	3	4	0	2	1	3
✓ P_3	1	3	1	2	1	4	2	4	6	9	6	6	0	1	1	2
✓ P_4	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3
									12	12	8	10				

Safe sequence : $P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$.
As safe sequence exists, system remains in safe state after granting the request
Therefore, request can be granted immediately.

If request from P_4 arrives for (0, 0, 2, 0), can it be granted immediately?

Request i.e (0, 0, 2, 0) ≤ Need i.e. (2, 2, 3, 3)

Check if Request ≤ Available
(0, 0, 2, 0) ≤ (3, 3, 2, 1) ⇒ True

System pretends to grant the request:

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

safe sequence: Since, need of none of the processes can be fulfilled with available resources, safe sequence doesn't exist ⇒ system is in unsafe state

Therefore, request can't be granted immediately

Algorithm

Input : - Processes
- any 2 out of (Max, Need, Allocation matrix)
- any 1 out of (Available, Total)

Safety Algorithm :

1. Let Work & Finish be vectors of length m & n.
   Work = Available
   Finish [i] = false for $i = 0, 1, \ldots, n-1$
   n = no. of processes
2. Find i such that :
   Finish [i] = false
   $Need_i \leq Work$
   If no such i exists, go to step 4.
3. Work = Work + Allocation$_i$
   Finish [i] = true
   go to step 2.
4. If finish [i] = true for all i, then system is in safe state
   else, it is in unsafe state

Time Complexity = $O(mn^2)$
m = no. of resources
n = no. of processes

Resource - Request Algorithm for Process P_i

1. Request = request vector for process P_i
If Request $\leq$ Need,
    go to step 2
Otherwise,
    raise error as process has exceeded its max claim.
2. If Request $\leq$ Available,
    go to step 3
Otherwise,
    P_i must wait as resources are not available.
3. Pretend to allocate requested resources to P_i by modifying:
    - Available = Available - Request;
    - $Allocation_i$ = $Allocation_i$ + $Request_i$;
    - $Need_i$ = $Need_i$ - $Request_i$;

4. Find whether system is in safe state after allocation
    - If safe $\Rightarrow$ resources are allocated to P_i
    - If unsafe $\Rightarrow$ P_i must wait & the old resource-allocation state is restored.

3) Deadlock Detection & Recovery

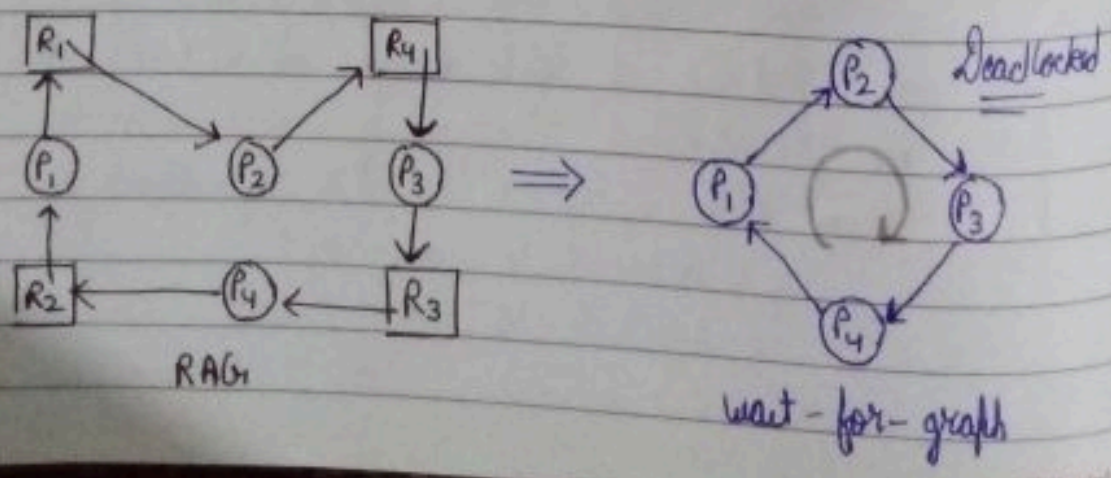
→ Allow the system to enter into deadlocked state
→ System checks for deadlock periodically.
→ Deadlock is detected using deadlock detection algorithms & recover the system using some recovery techniques

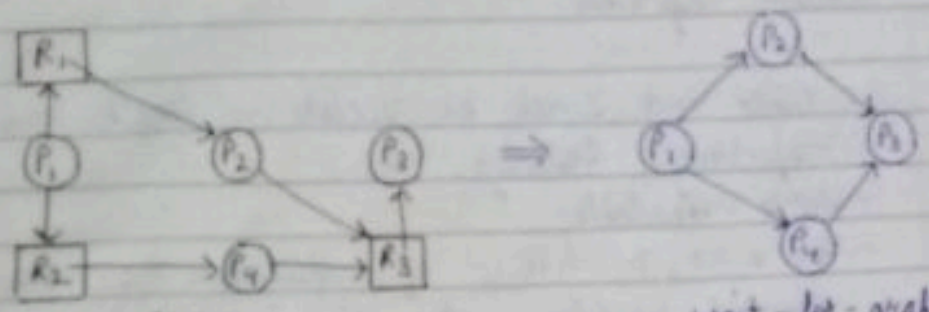
Deadlock detection algorithms

1) Wait-for-graph (for single instances)
- It is a RAG which consists of only processes as its nodes.
- set of
  - nodes - Processes (P_i)
  - edges - $P_i \rightarrow P_j$
    (P_i is waiting for the resource held by P_j)

This algorithm checks for a cycle in the wait-for-graph
- If cycle is detected ⇒ system is in deadlock
- otherwise ⇒ system is not in deadlock

Example:





RAG

wait-for-graph
(no cycle, no deadlock)

2) For multiple instances

It requires
- Total no of instances available for each resource in the system (1D array)
- Instances of each resource held by each process (2D array)
- Current request of ~~ea~~ instances of each resource by each process (2D array)

Example	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
✓ P_0	0	1	0	0	0	0	0	0	0
✓ P_1	2	0	0	2	0	2	0	1	0
✓ P_2	3	0	3	0	0	0	3	1	3
✓ P_3	2	1	1	1	0	0	5	2	4
✓ P_4	0	0	2	0	0	2	5	2	6
							7	2	6

safe sequence : $P_0 \to P_2 \to P_3 \to P_4 \to P_1$
System is not in deadlock

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2	0	1	0
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

safe seq : P_0 → further no request can be granted
P_1, P_2, P_3, P_4 are in deadlock

Dt
Pg

Detection Algorithm

1) Let Work and Finish be vectors of length m & n respectively, Initialize
   - Work = Available
   - For $i = 1, 2, \ldots n$, if Allocation $\neq 0$, then Finish [i] = false, else Finish [i] = true

2) Find an index i s.t. both
   - Finish [i] == false
   - Request$_i$ $\leq$ Work

   If no such i exists, go to step 4

3) Work = Work + Allocation$_i$
   Finish [i] = true
   go to step 2

4) If Finish [i] == false for some $1 \leq i \leq n$, system is in deadlock state & P_i is deadlocked

Time Complexity = $O(mn^2)$

Recovery Techniques

Recovery Techniques are used to recover a system from deadlock when detected by some deadlock detection algo.

Two approaches:

1) Process Termination (Pessimistic Approach)

i) Abort all deadlocked processes.
- All the partially completed processes not in deadlock are aborted even if they have computed for a long time
- This leads to high expenses

ii) Abort one process at a time
- Abort one process at a time and run deadlock detection algorithm. If proc system is still in deadlocked state, abort another process and continue till deadlock is removed.
- Overhead of executing deadlock detection algo multiple times
- Process to be aborted is decided on the basis of:
  - priority of processes.
  - time for which the process has computed
  - time required for completion.
  - No & type of resources utilised
  - No & type of resources needed.

2) Resource Preemption (optimistic approach)
Pre-empt some resources from processes & give those resources to other processes until deadlock cycle is broken.
Three issues:

i) Selecting a victim
resources to be preempted must be decided in accordance with the cost minimization.

ii) Rollback
The process from which resources are preempted must be either
- rollbacked completely & restarted again or
- rollbacked to a safe state

iii) Starvation
If a process is picked as a victim multiple times, it may lead to starvation.
Therefore a process must be selected as a victim for only a finite no of times.

4) Deadlock Ignorance

- It is used in the systems in which deadlock occurs rarely. Deadlock is completely ignored and rather than spending resources on it, more frequently occurring issues are handled such as compiler errors, system crashes due to hardware failure, OS bugs.
- This approach is used in UNIX & Windows
- If deadlock occurs, it is handled by rebooting

Dt.
Pg. B+

INTER PROCESS COMMUNICATION

Process synchronization

In multiprogramming

- It is a mechanism that allows processes to communicate with each other and synchronize their actions
- Intercommunication allows a process letting another process know that some event has occurred.
- It is used for exchanging threads in one useful information between numerous threads in one or more programs.

Processes (Concurrent)

- Independent : doesn't affect other processes
- Cooperative : affects other processes (shares resources)
  requires IPC ∴ sync

Approaches for inter process comm.

- Pipes
- Socket
- File
- Signal
- Shared memory
- Message passing

Process synchronization is required for IPC

Race ~~around~~ Condition

- Race condition occurs when two or more processes try to modify a shared data at the same time and the order in which they are executed is imp.
- As a process may be preempted any time, which leads to change of order, it may change the result
- In this condition both the processes are 'racing' to access & modify the shared data & the result depends on the order in which data is accessed & modified
- The order of execution is based on CPU scheduling algo

A = 7 12 (14)

Example

P1 → P2
(A = 19)

P_1	P_2
Read(A) 7	Read(A) 7
A = A * 2 14 →	A = A + 5 12
write (A) 14	← write (A) 12

Data inconsistency

S = 5 6 (4)

P1 → P2
A = 5

P_1	P_2
int X = 5; 5	int Y = 5; 5
X++; 6	Y--; 4
sleep(1);	sleep(1)
S = X; 6	S = Y; 4

Dt. _____ B+
Pg. _____

Producer - Consumer Problem

In this problem, a producer & a consumer thread run concurrently
- ~~There is~~ a producer ~~who~~ ~~which~~ produces an item &
- A consumer ~~which~~ consumes the item.
- They both share ~~a~~ the same memory buffer (fixed size)

- A producer can produce only one item at a time & consumer can consume only one item at a time.
- If buffer is full, producer is not allowed to produce any item & if buffer is empty, consumer is not allowed to consume.
- Producer & consumer can't access the memory buffer at the same time.

Both producer / consumer can produce / consume multiple times continuously

Requirements :

Buffer : It is a shared memory buffer used by both producer & consumer. (size - N)

Count : Keeps track of no. of elements currently present in the buffer $(0 \le count \le N)$

in : represents the next empty buffer cell to be filled $(0 \le in \le N-1)$

out : represents the next filled buffer cell to be consumed. $(0 \le out \le N-1)$

Initially : Buffer is empty
count = 0 , in = 0 , out = 0

Tutorial

int count = 0;

Producer code

```
void Producer (void)
{
    int itemp; in = 0;
    while (true)
    {
        produce_item (itemp);
        while (count == n);
        // Buffer full do nthg
        Buffer [in] = itemp;
        in = (in + 1) mod n;
        count = count + 1;
    }
}
```

Load R_p, M[count]
INR R_p
STORE M[count], R_p

Consumer code

```
void Consumer (void)
{
    int itemc; out = 0;
    while (true)
    {
        while (count == 0);
        // Buffer empty do nthg
        itemc = Buffer [out];
        out = (out + 1) mod n
        consume_item (itemc);
        count = count - 1;
    }
}
```

Load R_c, M[count]
DER R_c
Store M[count], R_c

Expected Result:

Buffer [8]

0	x	
1	y	
2	z	
3	a	
4		
5		
6		
7		

in: 3 4

out: 0 1

Producer → Produces (a)

Consumer → Consumes (z)

Count 3 4 3

Here,

Buffer contains 3 items - x, y, z

Initially count = 3, In = 3, out = 0

Producer produces an item - a.

Buffer contains 4 items - x, y, z, a

count = 4, In = 4, out = 0

Consumer consumes an item - x,

Buffer contains 3 items - y, z, a

count = 3, In = 4, out = 1

~~An~~ One item is produced & One is consumed ∴ no. of items in buffer remains 3.

Actual Result:

Buffer [8]

0	~~x~~
1	y
2	z
3	a
4	
5	
6	
7	

Producer → Produces (a) (preempted before it increments count)

In: [3 4]

Out: [0 1]

Consumer → Consumes (x)

Load	R_p M[count]	$R_p = 3$
INR	R_p	$R_p = 4$
Store	M[count], R_p	M[count] = 4

Count [3 2 4]

Load R_c, M[count]	$R_c = 3$
DECR R_c	$R_c = 2$
Store M[count], R_c	M[count] = 3

Race condition → b/w producer & consumer

Here,

Buffer contains 3 items => x, y, z

Initially Count = 3 In = 3 , out = 0

Producer produces an item → a

Buffer now contains 4 items => x, y, z, a

But it is preempted before it stores incremented count value.

∴ Count = 3 In = 4 Out = 0

Consumer consumes an item → x

Buffer now contains 3 items → y, z, a

Count = 2 In = 4 Out = 1

Now producer stores the incremented count value

∴ Count = 4 In = 4 out = 1

Buffer contains only 3 items → y, z, a

But Count = 4

This situation occurs due to race condition & is known as producer consumer problem

Printer Spooler problem

- Printer is a peripheral device & hence slower as compared to CPU
- ∴ Files that are to be printed are stored in a spooler directory by spooler program
- Printer then prints the files from spooler directory
- Programs that Concurrent processes share the spooler code and 'In' variable, spooler directory

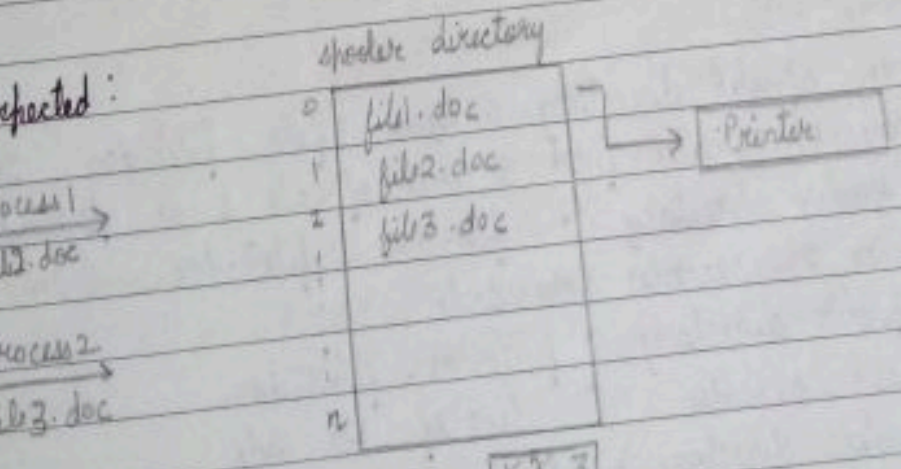
Requirement :

Spooler directory : stores all the files to be printed (n)
In shared variable which stores index where next file is to be entered

Spooler code :

```
Load Ri , m[in]
Store SD[Ri] , "filename"
INCR Ri
Store m[in] , Ri
```

Expected :



Here,
spooler directory contains 1 file - file1.doc
Initially in = 1
Now, process 2 executes spooler program
spooler directory contains 2 files - file1.doc, file2.doc
in = 2
Process 2 executes spooler program
spooler directory contains 3 files - file1.doc, file2.doc, file3.doc
in = 3

Actual Result:

Process1	Load R_1, m[in]	$R_1=1$	file1.doc	→ Printer
	Store SD[R_1], "file2.doc"		file2.doc file3.doc	
preempted	INCR R_1	$R_1=2$		
	Store m[in], R_1	m[in]=2		
Process2	Load R_2, m[in]	$R_2=1$		
	Store SD[R_2], "file3.doc"	in 2		
	INCR R_2	$R_2=2$		
	Store m[in], R_2	m[in]=2		

Race condition
Data Loss

Here, spooler directory contains 1 file - file1.doc, in = 1
Process1 executes first 3 lines of code & is preempted
spooler directory → file1.doc, file2.doc, in = 1
Process 2 executes the code
spooler directory → file1.doc, file3.doc, in = 2
Process1 executes 4th line of the code
spooler directory → file1 doc, file3.doc, in = 2

B+

Critical Section

- It is a section of a program which where shared variables or resources are accessed.
- shared resource can be any memory location, I/O devices etc
- Concurrent, cooperative processes can't execute their critical sections simultaneously → this may lead to race condition

Program:

non-critical section
critical section
non-critical section

- To ~~ensu~~ Critical sections need to be synchronized.
- To ensure atomic execution of a critical section, an entry & exit code is inserted before and after the critical section

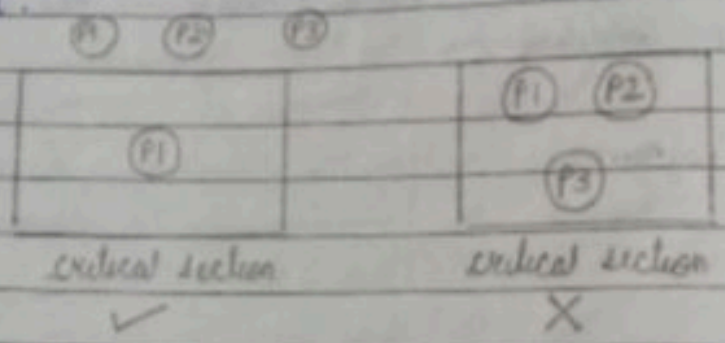
non-critical section	
entry section	← notifies program has entered into critical section
critical section	
exit section	← notifies program has exit from critical section
non-critical section	

A process is allowed to enter into a critical section only after passing through the entry section.

Requirements for process synchronization
(solutions for ~~atomic~~ execution of critical section)

Primary (must)

1) Mutual exclusion
If one process is executing inside critical section, other processes must not enter in the critical section.



2) Progress
A process executing in the non critical section must not stop other processes from entering into the critical section

Secondary

3) Bounded waiting
Waiting time for a process to enter into the critical section must be bounded / finite. A process must not be endlessly waiting to get into the critical section.

4) Architectural neutrality
The mechanism must work on all types of h/w architectures & diff. platforms.

SEMAPHORES

Semaphore is a integer variable which is used for process synchronization using wait() and signal() operations. shared by concurrent processes in mutually exclusive manner

Two types:

1) Counting semaphore value : ($-\infty$ to ∞)
used to provide access to the resources having multiple instances.

program
non critical section
entry section → P(S)
critical section
exit section → V(S)
non critical section

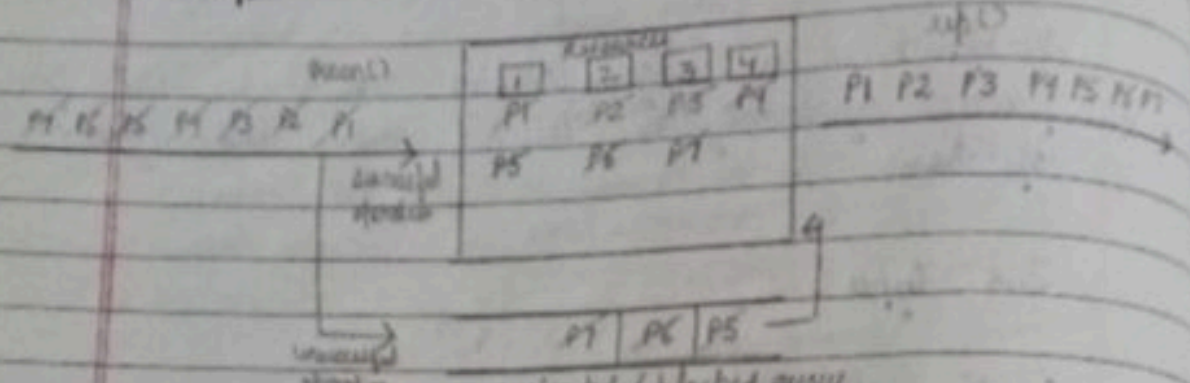
Consists of two operations:
- Down() / P() / wait() / lock()
- Up() / V() / signal / post / release()

Shared Semaphore { int value; }

```
down (semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        Put process (PCB) in
        suspendent list, sleep();
    }
    else
        return;
}
```

```
up (semaphore S)
{
    S.value = S.value + 1
    if (S.value ≤ 0)
    {
        select a process from
        suspended list, wake up();
    }
    else
        return;
}
```

Example: S value 4 3 2 1 0 -1 -2 -3 -4 -5, 0 1 2 3 4



If S = 0 → no res. instance is avl

If S = n > 0 → n res instances are idle

If S = n < 0 → n processes are waiting in blocked state

→ If S = 7 , 3P 2V 4P 1V S = ?

~~7+3-2+4-1~~ S = 7 - 3 + 2 - 4 + 1 = 3

waiting queue remains empty

Dt.
Pg. B+

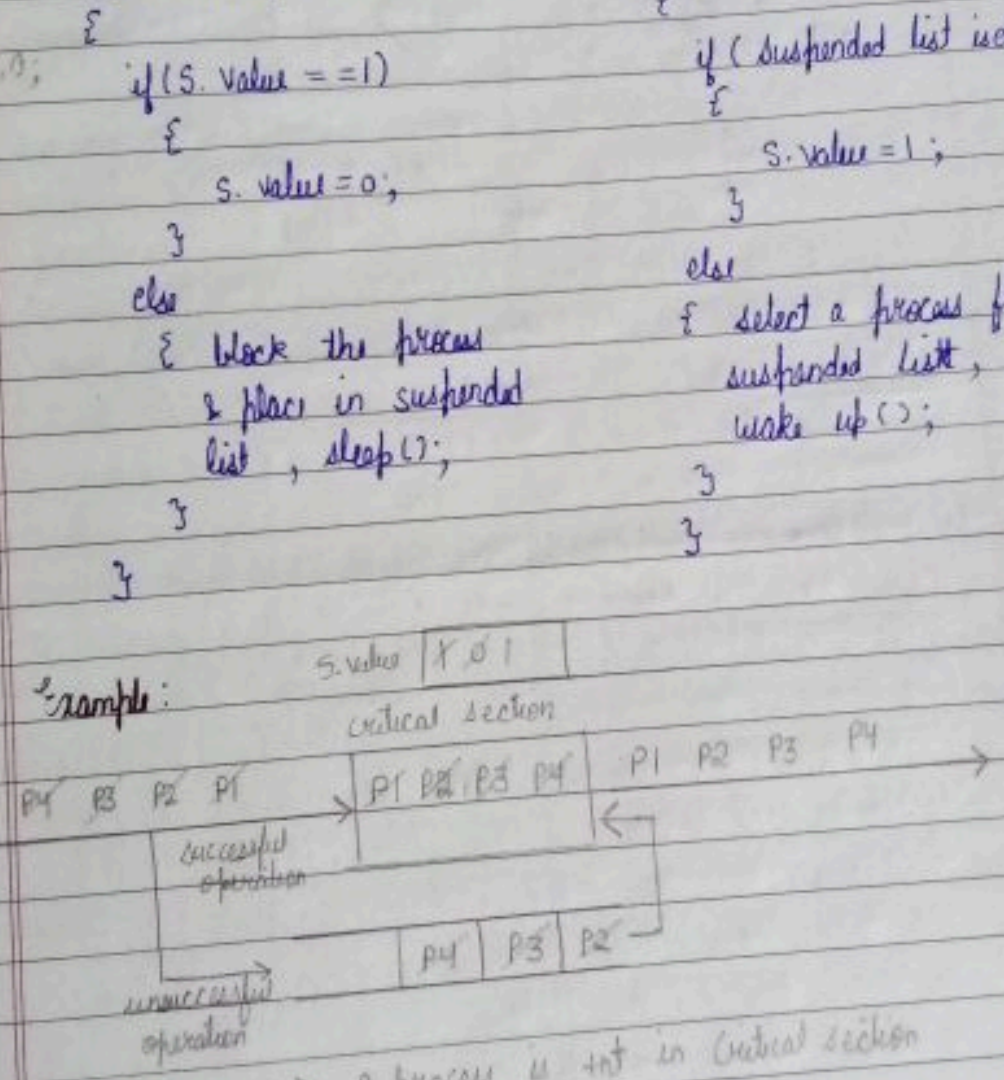
2) Binary semaphore value : {0, 1}
used to deal with critical section

struct semaphore
{
enum value {0, 1};

```
Down ( semaphore S)
{
    if (S. value ==1)
    {
        S. value = 0;
    }
    else
    { block the process
      & place in suspended
      list , sleep();
    }
}
```

```
up ( semaphore S)
{
    if ( suspended list is empty)
    {
        S. value = 1;
    }
    else
    { select a process from
      suspended list,
      wake up();
    }
}
```

Example:



If S = 0 → A process is int in Critical section
S = 1 → CS is empty

→ If S = 1 , 3P 1V 5P 3V , S = ? 0 Ans
S = 1 → 0 → 0 → 0 → 0
BQ : 2 1 6 3 3 slot remaining in SQ
Max = 6

Tutorial

Producer Consumer Problem - Solution
(using semaphores)

We use three semaphores - 2 counting & 1 binary semaphore

Counting semaphores : full (no of filled cells) initially = 0
empty (no of empty cells) initially = N

Binary semaphore : Mutex initially = 1

Producer

```
Void Producer (void)
{ int itemp; in = 0
  while (true)
  {
1     produce-item (itemp);
2 entry  Down (empty);
3 section  Down (Mutex);
4 critical  Buffer [in] = itemp;
5 section  in = (in + 1) mod n;
6 exit  up (Mutex);
7 section  up (Full);
  }
}
```

Consumer

```
Void Consumer (void)
{ int item, out = 0
  while (true)
  {
entry  Down (Full);   1
section  Down (Mutex);   2
critical  item = Buffer[out];   3
section  out = (out + 1) mod n;   4
exit  Up (Mutex);   5
section  Up (empty);   6
  Process-item (item);   7
  }
}
```

- Block which blocks a process → not executed again when unblocked
- when up (binary sem after) check BL before set → 1
- when up → check for BL

Pg _____ B+

In [3 4]
out [0 1]

	Buffer (8)	
0	x	Producer executes till line 2 (produces x)
1	y	↓ Consumer executes till line 3
2	z	↓ Producer executes line 3
3	a	(Blocked)
4		↓ Consumer resumes from line 4
5		(producer unblocked at line 5)
6		↓
7		Producer resumes from line 4

Full [3 ~~4~~ 3]
Empty [5 4 5]
Mutex [1 0 1]

→ In case Buffer is Full
producer executes → blocked at line 2 → Consumer executes
producer unblocked at line 6 → producer executes

Full [8 ~~7~~ 8]
Empty [0 ~~1~~ 0]
Mutex [~~1~~ 0 ~~1~~ 0 1]
In [0 1]
out [0 1]

→ In case Buffer is Empty
Consumer executes → blocked at line 1 → producer executes
→ consumer unblocked at line 7

Full [0 ~~1~~ 0]
Empty [8 ~~7~~ 8]
Mutex [1 0 1 0 1]
In [0 1]
Out [0 1]

Dt
Pg B+

Readers Writers Problem - Solution
(using semaphores)

Readers-writers problem occurs when a file or database is shared between multiple reader, writer processes.
Readers are the processes that want to read the data
Writers — write/change -
Four cases arise :
- Reader & writer execute simultaneously - may cause problem
- Reader & Reader — - doesn't cause -
- Writer & Reader — - may cause problem
- Writer & Writer — - —

~~We~~ Readers Writers problem can be solved with the help of semaphores :
We use two binary semaphores : Mutex Initially 1
db Initially 1

A variable rc (readers count) Initially 0

Writers

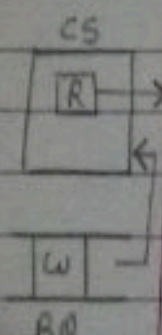
```
void writer (void)
{ while (true)
  {
    down (db);        entry section
    <Access database>  CS
    up (db);          exit section
  }
}
```

Tutorial

Dt
Pg

Reader

```
void Reader (void)
{
    while (true)
    {
1       down (Mutex)
2   entry   rc = rc + 1
3   section   if (rc == 1) then down (db)
4       up (Mutex)
5   CS  < Access database >
6       down (Mutex)
7   exit    rc = rc - 1
8   section   if (rc == 0) then up (db)
9       up (Mutex)
    }
}
```



Case : I Reader - Writer

- Reader enters in CS
- Writer executes line 1 & gets blocked. [∵ db = 0]

Mutex	~~1~~	~~0~~	~~1~~	~~0~~ 1
db	~~1~~	~~0~~	1	
rc	~~0~~	~~1~~	0	

- When reader completes the execution of critical section, it executes the exit section. (Writer is unblocked at line 8)
- Writer executes from line 2

Case II Reader - Reader

- Reader1 executes entry section & enters into CS
- Reader2 executes entry section & enters the CS
- Reader1 completes execution of CS & executes exit section
- Reader2 completes execution of CS & executes exit section

Case III Writer - Reader

- Writer executes entry section & enters into CS
- Reader executes entry section & gets blocked at line 2
- Writer completes execution of CS & enters exit section (reader is unblocked at line 12)
- Reader resumes from line 4

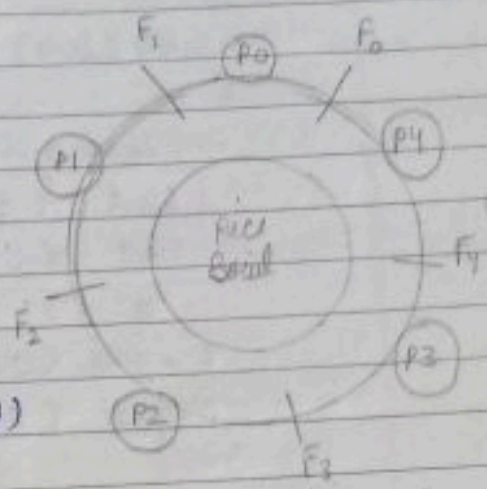
Case IV Writer - Writer

- Writer1 executes entry section & enters the CS
- Writer2 executes entry section & gets blocked at line 1
- Writer1 ~~exec~~ completes execution of CS & enters exit section (writer2 is unblocked at line 12)
- Writer2 resumes from line 2

Dt. _____ B+
Pg. _____

Dining Philosophers Problem - Solution
(using semaphores)

There are 5 philosophers & 5 forks. Philosophers share a circular table on which a bowl of rice is placed. Each philosopher thinks & eats. A philosopher can eat only if both forks surrounding him (left & right) are available



Philosophers = 5
Forks = 5

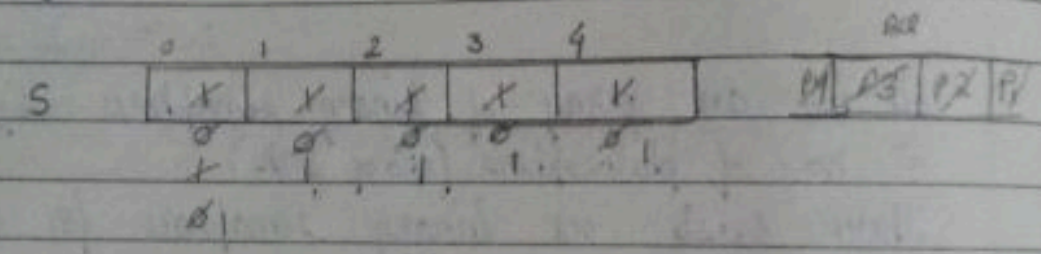
We use an array of binary semaphores. Size of array = no. of philosophers (= no of forks)
There exists one binary semaphore for each fork.
Initially array is initialised to 1 bcz all the forks are available.

	S[0]	S[1]	S[2]	S[3]	S[4]
S	1	1	1	1	1

A philosopher i requires i^{th} & $[(i+1) \bmod n]^{th}$ fork to eat. ∴ semaphores required are:

P_0 requires S_0 & S_1
P_1 requires S_1 & S_2
P_2 requires S_2 & S_3
P_3 requires S_3 & S_4
P_4 requires S_4 & S_0

```
void Philosopher (int i)
{
    while (true)
    {
        Thinking ();
1       wait (S[i]);                    entry section   ] Take forks
2       wait (S[(i+1) mod 5]);
3       Eat ();                         CS
4       signal (S[i]);                  exit section    ] Put forks
5       signal (S[(i+1) mod 5]);
    }
}
```



Philosopher P_0 ~~enters~~ executes entry section & enters CS
S[1]=0 Philosopher P_1 executes entry section & gets blocked at line 1
Philosopher P_0 completes CS & enters exit section
(Philosopher P_1 is unblocked at line 5 & S[1] remains 0)
Philosopher P_1 resumes from line 2 & enters CS
S[2]=0 Philosopher P_2 executes entry section & gets blocked at line 1
Philosopher P_1 completes CS & enters exit section
(Philosopher P_2 is unblocked at line 5 & S[2] remains 0)
Philosopher P_2 resumes from line 2 & enters CS
S[3]=0 Philosopher P_3 executes entry section & gets blocked at line 1
Philosopher P2 completes CS & enters exit section
(Philosopher P_3 is unblocked at line 5 & S[3] remains 0)

Philosopher P_2 resumes from line 2 & enters CS
Philosopher P_4 executes entry section & gets blocked at line 1
Philosopher P_3 completes CS & enters exit section
(Philosopher P_4 is unblocked at line 5 & S[4] remains 0)
Philosopher P_4 resumes from line 2 & enters CS
Philosopher P_4 completes CS & enters exit section

This solution makes sure no two adjacent philosophers eat at the same time
But it may lead to deadlock.

	0	1	2	3	4
S	✗	✗	✓	✗	✗

Philosopher P_0 executes line 1 & gets preempted
Philosopher P_1 executes line 1 —
Philosopher P_2 —
Philosopher P_3 —
Philosopher P_4 —

This leads to circular wait resulting in a deadlock
This can be prevented by reversing the order of in which a philosopher acquires the forks for any one of the philosophers suppose order of forks is altered for P_4

P_0 requires forks	f_0 & f_1	then f_1	$(S_0$ & $S_1)$
P_1 —	f_1 &	then f_2	$(S_1$ & $S_2)$
P_2 —	f_2	— f_3	$(S_2$ & $S_3)$
P_3 —	f_3	— f_4	$(S_3$ & $S_4)$
P_4 —	f_0	— f_1	$(S_0$ & $S_4)$

P_4 gets resumes from line 1 & 2 & 3 & 4 & 5

	0	1	2	3	4
S	✗	✗	✗	✗	✗
	$\emptyset_1$	$\emptyset_1 \emptyset_1$	$\emptyset_1 \emptyset_1$	$\emptyset_1 \emptyset_1$	$\emptyset_1 \emptyset_1$

Philosopher P_0 executes line 1 & gets preempted
Philosopher P_1 ——
Philosopher P_2 ——
—— P_3 ——
S[0]=0 Philosopher P_4 executes line 1 & gets blocked
Philosopher P_3 resumes from line 2 & completes execution
Philosopher P_2 ——
Philosopher P_1 —
Philosopher P_0 —
(P_4 is unblocked at line 4 & S[0] remains 0)
Philosopher P_4 resumes from line 2 & completes execution

∴ deadlock is avoided.