02/08/20

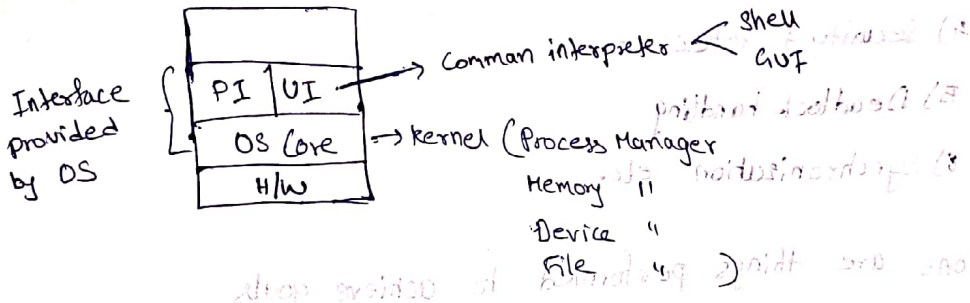
Operating System

(8-10M)

Introduction & Background:

what is an OS?

→ Interface b/w user & hardware
→ Resource Manager
→ set of utilities to aid application Development
→ control program (controls various h/w & s/w units)
→ Acts like a government



PI - programmer Interface

Programmers are given seperate interface i.e., PI

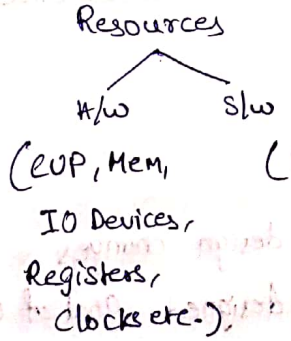
this is provided using API (Application Programmers Interface)

or SCI (System Call Interface)

→ In GATE OS is studied on Von Neumann Architecture only.

OS as Resource Manager:

* Manages resources & provides a platform to use those resources

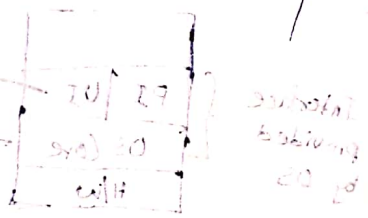


Functions & Goals of OS:

→ 1) User Convenience } Major goals
2) Efficiency
3) Reliability
4) Scalability (ability to evolve i.e. expandable)
5) Robustness (Strong and should be able to bear error)
6) Portability (usable across various platforms)

Functions

1) CPU Scheduling
2) Memory Management
3) Device "
4) Security & Protection
5) Deadlock handling
6) Synchronization etc.



* Functions are things performed to achieve goals

(Q1) what must be the primary goal of OS?

a) Convenience
b) efficiency
c) Reliability
d) Scalability

Sol:

Even though efficiency is a major goal,

people are more concerned about convenience.

∴ opt (a)

Note:

→ If architecture changes then OS design changes
→ Based on the domains for which OS is being designed, goal of os changes

Eg: for PC, it is convenience

for real time systems, it is efficiency and highly reliable and robustness

for smartphones, it is convenience & power efficiency.

* Real time systems (RTS) are the systems which are constrained by strict time deadlines.

Eg: Missile control, Nuclear systems etc.

for Distributed systems' OS, it is scalability

03/08/20

Types of O.S:

Uniprogramming:

→ ~~only one program will be ready running at any given time~~. only one program is present in main memory at any given time.

The area of main memory in which OS is stored is called system area. The rest of the memory is called user area.

Drawback:

* when the program goes for I/O, the CPU becomes idle.

Multiprogramming:

→ Here OS has ability to manage multiple ready programs in memory.

So even if ~~one of the ready proce~~ executing process goes for I/O, one of the ready processes will be allocated CPU.

Thus utilization of CPU is maximized.

Hence throughput is increased.

throughput: No of processes executed per unit time

Types of Multiprogramming (M.Pr):

- Non-Preemptive (process leaves CPU only when it completes or I/O or syscall)
- Pre-emptive (i.e., forceful deallocation)

* → Drawback of non-pre-emptive M.Pr is
lack of responsiveness or interactiveness
whereas pre-emptive is more interactive.

→ Pre-emptive Multiprogramming is also refered as multitasking.

Architectural Requirement for implementation of a pre-emptive based M.Pr O.S:

1. IO (secondary storage):

It is desirable to have DMA capability

2. Main Memory:

Should be capable to perform address translation.
(i.e., logical add. to phy. address)

This functionality is provided by Memory Management Unit (MMU).

3. CPU:

The processor must be able to support dual mode operation.

The two modes are:
(i) user mode (non-perviliaged mode)
(ii) kernel mode (Previlage mode or monitor mode)

* All OS programs run in kernel mode
* All user programs run in user mode
* OS programs to execute atomically (i.e., no preemption)
* User programs are preemptive nature
* To keep track in which mode CPU is present we used mode bit.

This mode bit is present in Condition register (Processor status word (PSW))
- 0: User mode
- 1: kernel mode

(or vice versa)

Mode shifting: Many a time while execution of program, it is necessary to shift from user mode to kernal mode and KM to UM. This process of shifting is called mode shifting.

Eg: If a user process needs OS service, then the OS service is done in kernel mode. Once the service is finished mode is changed to user mode.

Eg:
```
main()
{
    int a,b,c;
    c=f(a,b);
    printf("%d", c);
}
f(a)
{
    :
}
```

The above program doesn't use any OS service and finishes all of its execution in user mode.

Eg:
```
main()
{
    int a,b,c;      UM
    c=f(a,b);       UM
    fork();         KM
    printf("%d", c); UM
}
f()
{
    :
}
```

fork(): System call (provided in API)

The above program uses fork(). fork() is a system call and it is service provided by OS.

Every system call is translated by C compiler to an SVC (supervisory call). This is a privilaged instruction.

All non-privilaged function calls are translated to BSA (Branch & Save Address) by C compiler.

Thus at the time of execution of SVC, a software interrupt is generated.

The corresponding ISR (Interrupt Service Routine) handles the interrupt.

→ ISR does 2 two things:

i) Mode shifting (from UM to KM) (mode bit in PSW is changed)

ii) From the dispatch table, the required address of the function call is found

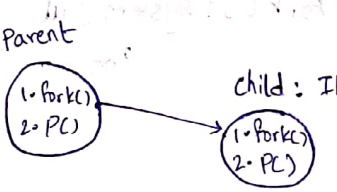
(Dispatch table contains one entry per each system call)

The last instruction of fork() contains an SVC which changes KM to UM.

fork() System call:

→
```
main()
{
    printf("Hello");
}
```
o/p: Hello

→
```
main()
{
  1. fork();
  2. printf("Hello");
}
```



Execution in child starts from the instruction which is next to the fork() which created the child.

∴ child doesn't execute fork()

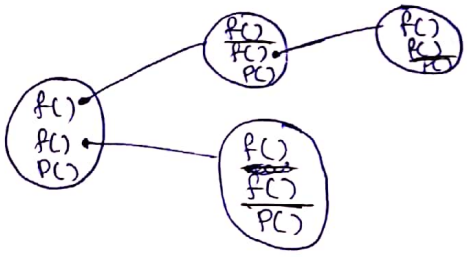
It executes only printf()

i.e., execution of child starts from PC of parent process

i.e., PC of child is set to $PC_p + 1$ where PC_p is PC of parent

∴ o/p: Hello
Hello

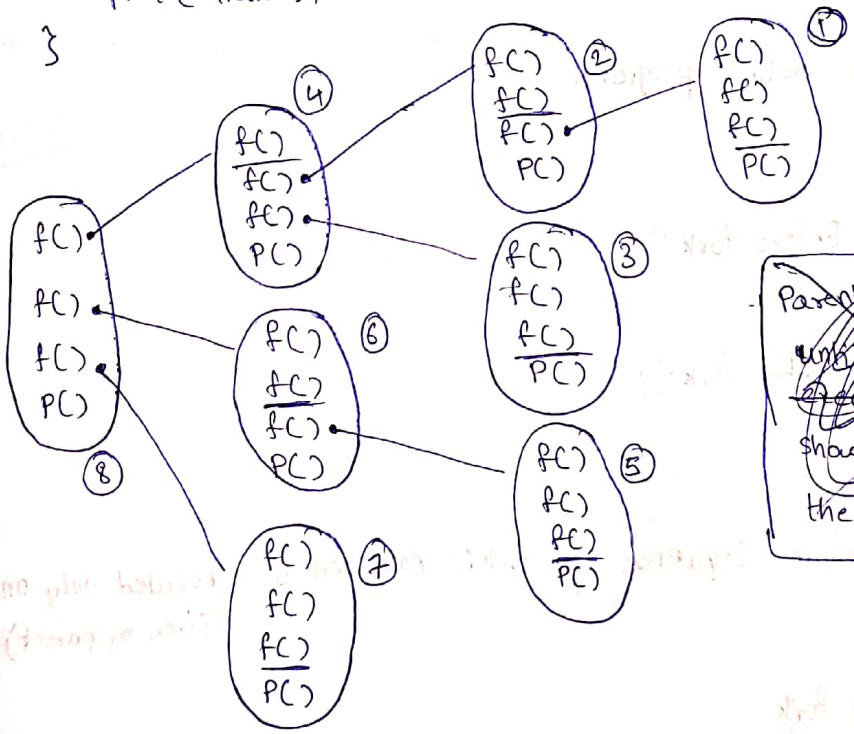
→ main()
{
fork();
fork();
printf("Hello");
}



Here a total of 4 processes are created. Hence 'Hello' is printed 4 times.

Hello
Hello
Hello
Hello

→ main()
{
fork()
fork()
fork()
print("Hello");
}



Parent doesn't execute until child finishes its execution. So the numbering shows the order in which the processes print hello

∴ 8 processes ⇒ Hello is printed 8 times

Note:
→ In the same way if we have n forks in sequence, 2^n process will exist.

(Q2)
```
main()
{
    int i,n;
    for(i=1; i<=n; ++i)
        fork();
}
```

For the above program, no of child processes created is _____

a) n-1 b) n^2-1 c) 2^n-1 d) n!-1

Sol:

```
for(i=1; i<=n; ++i)
    fork();
```

can be replaced by sequence of 'n' forks()

∴ total processes = 2^n

total no of child processes = 2^n-1.

(Q3) Find the o/p of below program.

```
main()
{
    printf("Before fork");
    fork();
    printf("After fork");
}
```

Sol:

Anything before sequence of fork() calls will be executed only once. (i.e., by parent)

∴ o/p:

Before fork
After fork
After fork

Process Management:

i) Process Concepts:

Program vs Process

Program (.exe)

- Instruction
- Data
  - (Static --- size is fixed, allocated ~~at zero~~ before run time
  - or
  - Dynamic) — variable size, allocated at runtime. (Heap allocation),

* Program is a passive entity.

* Program is in Disk and process is in main memory

Process:

* Process is a program in execution
* Process is an instance of a program (for a program we can create any number of processes)
* Active entity
* unit of CPU utilization
* Locus of control (i.e., if there is no process, then there is no activity for CPU).
* Animated ~~spirit~~ spirit.

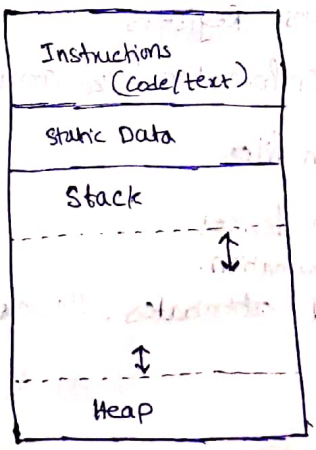
→ Process is viewed as an Abstract Data Type (ADT).

04/08/20

an ADT has

<Defn; Representation (or) structure; operations; Attributes>

Abstract view of Process in Memory or Process Structure



Dynamic Memory Allocation

Process Operations:

These operations are carried out by OS on behalf of users

(i) Create():

At the time of creation certain resources are allocated, PCB is created, devices are allocated. Eg: fork() is used to create()

(ii) Schedule() : selecting which process to run

(iii) Dispatch() : allocating CPU to process

(iv) Block() : Block for I/o

(v) Running() : running on CPU.

(vi) Suspend() : Process is sent to Disk

(vii) resume() : Process is brought is main memory from disk.

(viii) terminate() :

* Here resource deallocation takes place and PCB is deleted

Process Attributes (Properties / characteristics):

* Process id. : a unique id for every process
  Sometimes it may even have parent_id.
  Sometimes if the process is in any group it may even have group_id.

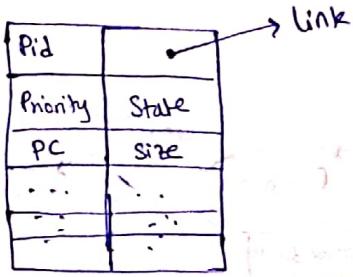
  } Identification related

* Process state
* Process size
* Process type (like background process, foreground process etc.)
* Priority
* Program Counter
* General Purpose Registers
* Accounting Information (like arrival time, burst time, IO_time etc)
* List of open files
* list of open devices
* Protection information.

These are general attributes. However different OS's use different attributes...

Process Control Block (PCB) / Task Control Block (TCB):

→ Process Attributes are stored in a Data structure known as PCB or Process Descriptor or Process Object.

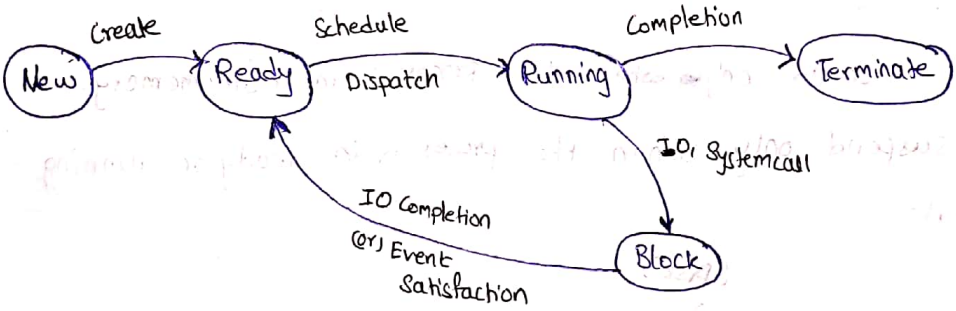


The content kept in PCB is called process context or process environment.

→ Each process has its own PCB.

Process states and state transition diagram

<New; Ready; Running; Block; terminate>



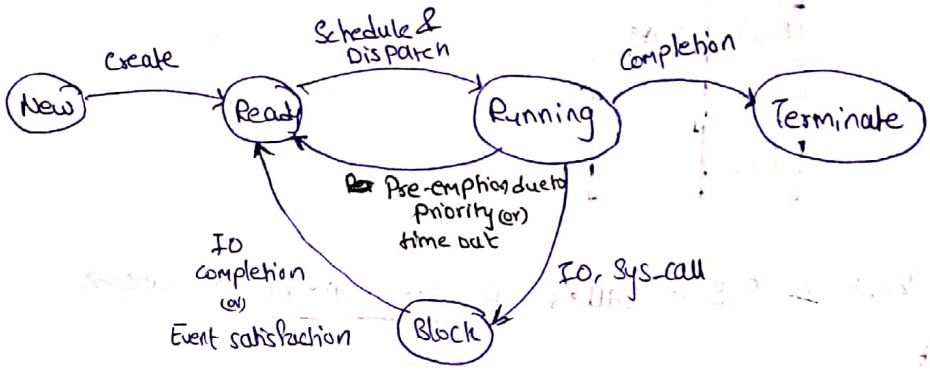
In New the process is in the process of creation.

* The above diagram is the Illustration for multiprogrammed OS because we have ready state.

In uniprogrammed OS we will not have Ready state

* Above state diagram is for non-preemptive based multiprogrammed OS. It is because the processes moves from running to block under two conditions (IO, system call) i.e., voluntary release of CPU and no forceful disallocation of CPU

* So to make in the state diagram we add a transition from running to ready ~~addition~~ under two conditions (preemption, priority) as shown below.

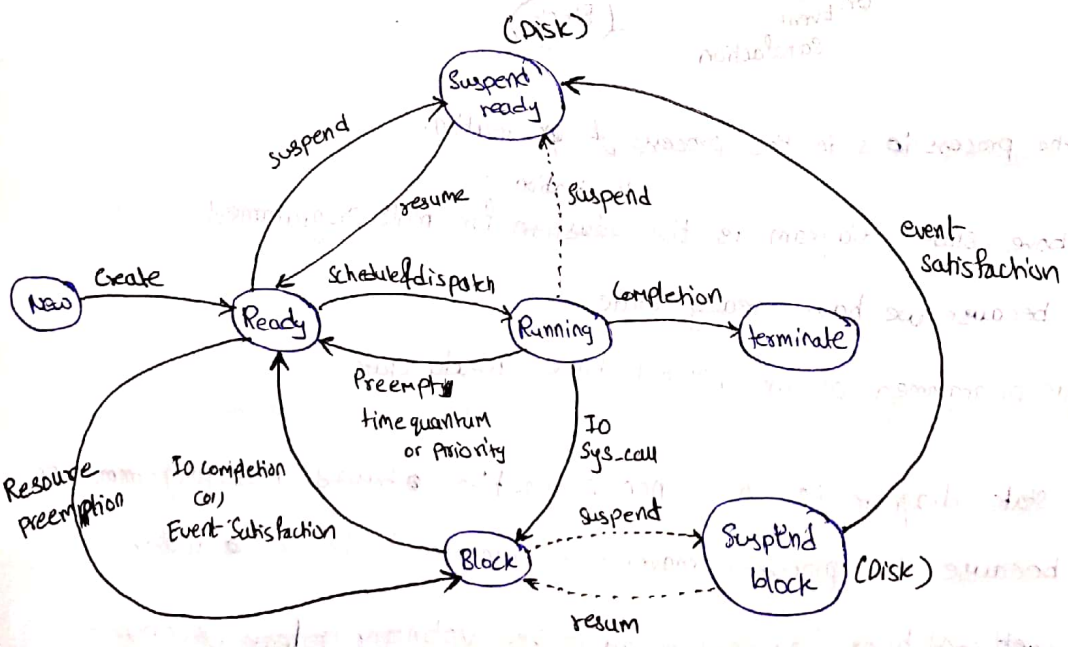


→ Now the above transition diagram is for pre-emptive based multiprogrammed OS.

Process ~~Sch~~ Suspension:

It is ~~swapping~~ swapping out the ~~proc~~ process from main memory to disk. And later we may resume the process and bring it back to main memory.

* It possible to suspend only when the process is in main memory.
So we can suspend only when the process is in ready or running or block state.

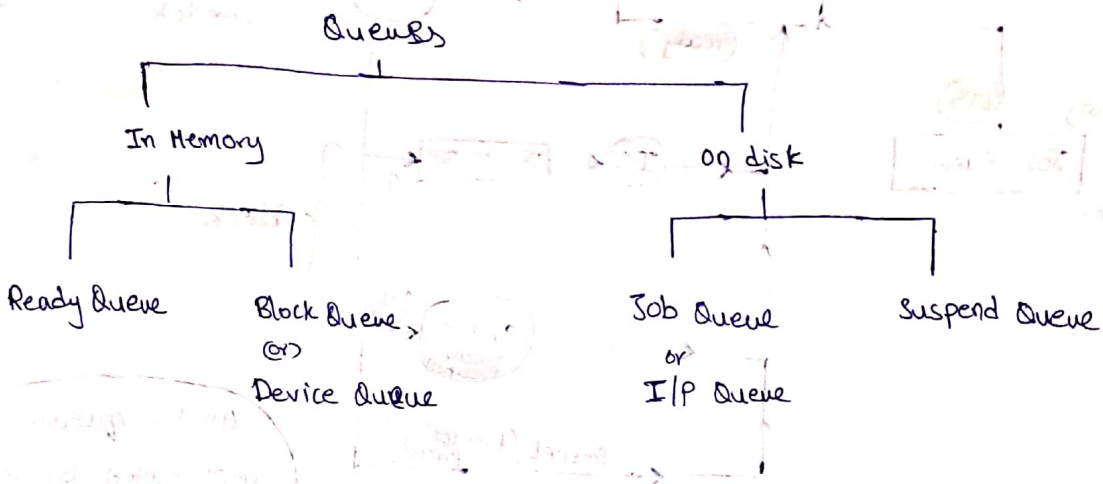


For special arrangements, IO operations are possible even in suspend block.

..... transition line here means not much desirable.

→ Resource preemption is taking resources from the process in ready state.

Scheduling Queues and Queuing diagrams:



Ready queue: list of PCBs of ready processes
These PCBs are maintained as linked list.

Block queue: Every IO device is associated with its own device queue (PCBs of processes)
Every device waiting IO through certain device is added to the respective device's queue.

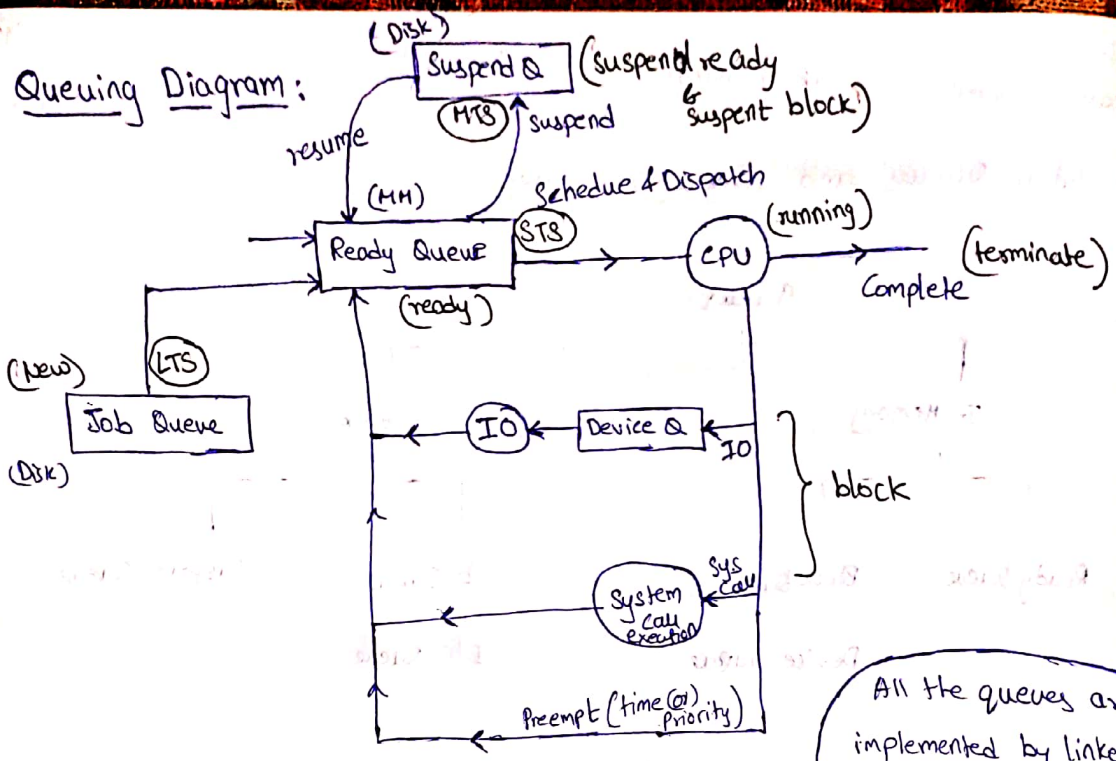
Job Queue (or) Input Queue:
Programs that are ready to be loaded in memory. Memory may not be enough to have all the processes. So rest are placed in job queue. The programs in job queue are present in new state.

Suspend Queue:
Processes that are suspended from memory either from ready or from block states.

CPU bound process: A process with high amount of CPU burst compared to IO burst.

IO bound process: A process with high amount of IO burst compared to CPU burst.

Queuing Diagram:



All the queues are implemented by linked list of PCBs.

Schedulers & Dispatcher:

Scheduler is a component of process manager that makes decision.

Short term scheduler (Process scheduler) (CPU scheduler)

It decides which ready process should run next on CPU.

Long term scheduler (Job scheduler):

It works on job Queues.

It loads programs into main memory

Long, medium, short are given on the basis of the frequency with which those schedulers run.

Medium term scheduler:

It carries out the responsibility of suspension & resumption.

Degree of Multiprogramming:

It is no of processes present in Memory.

LTS controls deg of M.Pr.

Dispatcher:

Dispatcher carries out the activity of context switching.

* Context switching is activity of saving and loading PCBs of processes during process switching on CPU, context switching also involves loading page table.

→ Time taken for context switching (save PCB & load PCB) is called context switch time (or) Dispatch latency.

(H1/1) n CPUs k processes

	lower bound	Upper bound
ready	0	k
running	0	n
block state	0	k

CPU Scheduling / Process Scheduling

→ It design of short term scheduler.

Functions & goals of STS:

functions:

* Scheduling

goals:

* Maximising throughput
* Minimize turn around time
* Minimize waiting time
* Minimize response time.

Process times:

Arrival time (AT) the time at which process comes into ready queue for the first time.

Waiting time (WT):

the time spent by a process in ready queue.

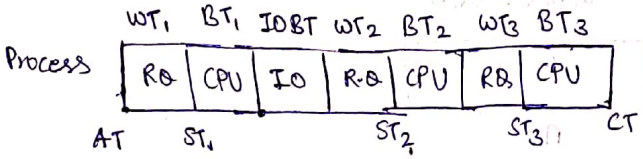
Scheduling time (ST):

the time at which the process gets on to CPU.

Burst time (BT):

CPU BT: time spent by process running on CPU

IO BT: time spent by process doing IO activity



Completion time (CT):

the time at which process finishes its execution.

Turn Around time (TAT):

the time spent by process from its arrival to Complete.

$$TAT = CT - AT$$

$$Avg - TAT = \frac{\sum_{i=1}^{n} TAT_i}{n}$$

$$Normalized\ TAT = \frac{TAT}{BT}$$

→ Here BT includes only CPU BT

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

if IOBT = 0, then

$$WT = TAT - BT$$

$$Avg.WT = \frac{\sum_{i=1}^{n} WT_i}{n}$$

Scheduling frame work:

Let n processes ($P_1 .. P_n$)

$AT(P_i) \longrightarrow A_i$

$BT(P_i) \longrightarrow X_i$

$IOBT(P_i) \rightarrow Y_i$

$CT(P_i) \longrightarrow C_i$

Now we have

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

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

$$\text{Normalized TAT}(P_i) = \frac{C_i - A_i}{X_i}$$

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

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

c) Schedule-length (L)

total time taken for all n processes to finish execution is called schedule length.

> L = CT of last process − AT of first process
>
> i.e., $L = \max(C_i) - \min(A_i)$

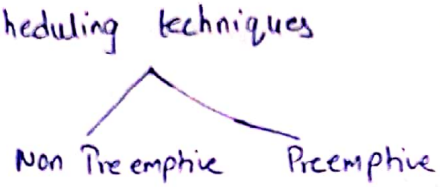
Note that L is not equal to sum of all burst times because at certain time all processes may be in block state

d) Throughput (μ):

process executed per unit time.

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

CPU scheduling techniques:



1) First Come First Serve (FCFS):

Selection criteria : AT

Mode of operation : Non-preemptive

Conflict resolution : Lower Pid process is scheduled first

Assumptions:

→ time is in clock ticks.

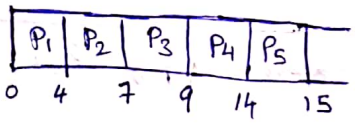
→ IOBT is zero

→ Context switch overhead (CS) : Negligible

Eg:

P.No	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	0	3	7	7	4
3	0	2	9	9	7
4	0	5	14	14	9
5	0	1	15	15	14

Gantt chart



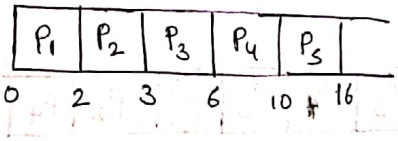
Schedule length = 15 − 0 = 15

$$\text{Avg TAT} = \frac{4+7+9+14+15}{5} = 9.8$$

$$\text{Avg. WT} = \frac{0+4+7+9+14}{5} = 6.8$$

(Q5)

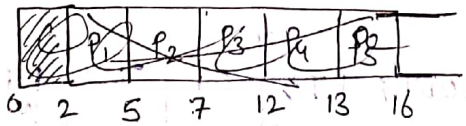
PNO	AT	BT	CT	TAT	WT
1	0	2	2	2	0
2	1	1	3	2	1
3	2	3	6	4	1
4	3	4	10	7	3
5	5	6	16	11	5

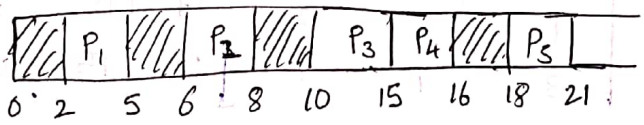


Schedule length = 16

(Q6) Compute Avg. TAT, Avg WT, Schedule length

PNO	AT	BT	CT	TAT	WT
1	2	3	5	3	0
2	6	2	8	2	0
3	10	5	15	5	0
4	12	1	16	4	3
5	18	3	21	3	0





$$\text{Avg TAT} = \frac{3+2+5+4+3}{5} = \frac{17}{5} = 3.4$$

$$\text{Avg WT} = \frac{3}{5} = 0.6$$

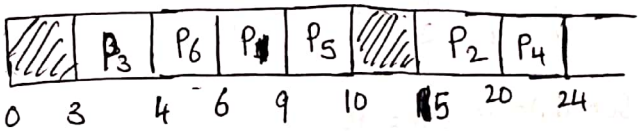
Schedule length = 21 − 2 = 19

Draw gantt chart

(Q7)

PNO	AT	BT
1	5	3
2	15	5
3	3	1
4	18	4
5	5	1
6	4	2

Gantt chart:



$L = 24 - 3 = 21$

$$\text{Avg TAT} = \frac{1+2+4+5+5+6}{6} = \frac{23}{6}$$

$$\text{Avg WT} = \frac{0+0+1+4+0+2}{6} = \frac{7}{6}$$

FCFS with Non-zero IO BT's & Non Negligible scheduling overhead (S):

Scheduling overhead (S) is context switch time.

Eg:

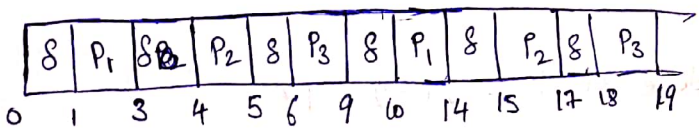
PNO	AT	<BT ; IOBT ; BT>	CT	TAT	WT
1	0	2 3 4	14	14	5-2
2	2	1 5 2	17	15	7-2
3	4	3 1 1	19	15	10-2

(calculate WT)

Scheduling overhead, $S = 1$

Assume concurrent IO is possible

i.e., Multiple devices can perform IO at the same time



TAT

$P_1 - 6$

$P_2 - 10$

$P_3 - 10$

In waiting time we don't consider time for scheduling overhead.

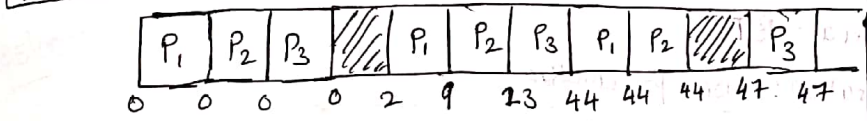
∴ $WT = TAT - (BT + IOBT + n \cdot s)$

where n is no of scheduling overheads
i.e., no of times the process is scheduled.

Schedule length = 19 - 0 = 19

H1/2

PNO	AT	< IO	BT	IO >	CT	TAT	WT
1	0	2	7	1	44	44	34
2	0	4	14	2	44	44	24
3	0	6	21	3	47	47	17



A process cannot go to IO directly from ready queue. So initially all 3 processes will run on CPU for a negligible amount of time

(i) Avg TAT = $\frac{44 + 44 + 47}{3} = \frac{135}{3} = 45$

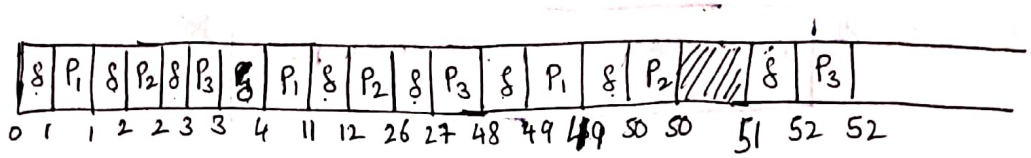
*→ Also a process is in block state while performing IO. So before termination it must come to running state. so finally it run on CPU for a negligible amount of time

(ii) % of CPU idleness = $\frac{5}{47} \times 100 = 10.16\%$

Schedule length = 47 - 0 = 47

→ FCFS shows convoy effect. i.e., All other processes wait for one big process to get off the CPU.

Now assuming S=1, we repeat the same problem



L = 52

P1 P2 P3 P1 P2 P3 P1 P2 P3

% CPU idleness = $\frac{1}{52} \times 100$

% CPU performing overhead = $\frac{9}{52} \times 100$

CPU efficiency = $\left(1 - \frac{1}{52} - \frac{9}{52}\right) \times 100 = \frac{42}{52} \times 100$

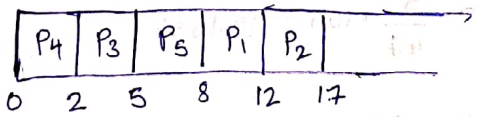
P1 - 3
P2 - 6
P3 - 9
P1 - 12
P2 - 28
P3 - 51

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

Selection Criteria : BT
Mode of operation : Non-Pre Emptive
Conflict Resolution : Lower pid

Eg:

PNO	AT	BT	CT	TAT	WT
1	0	4	12	12	8
2	0	5	17	17	12
3	0	3	5	5	2
4	0	2	2	2	0
5	0	3	8	8	5



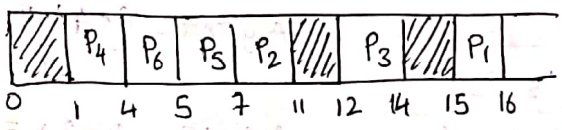
Avg TAT = $\frac{12+17+5+2+8}{5} = \frac{44}{5} = 10.8$

Avg WT = $\frac{8+12+2+0+5}{5} = \frac{27}{5} = 5.4$

L = 17 - 0 = 17

Q8)

PNO	AT	BT	CT	TAT	WT
1	15	1	16	1	0
2	2	4	11	9	5
3	12	2	14	2	0
4	1	3	4	3	0
5	4	2	7	3	1
6	3	1	5	2	1



Avg TAT = $\frac{20}{6} = \frac{10}{3}$

Avg WT = $\frac{7}{6}$

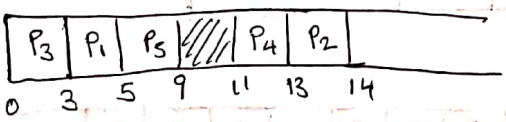
L = 16 − 1 = 15

% CPU idleness = $\frac{2}{15} \times 100$ → (Idleness must be calculated over schedule length)

05/08/20

Q9)

PNO	AT	BT	CT	TAT	WT
1	2	2	5	3	1
2	12	1	14	2	1
3	0	3	3	3	0
4	11	2	13	3	1
5	4	4	9	5	1



Shortest Remaining Time First (SRTF) / Pre Emphive SJF

Criteria : BT

Mode : Pre Emphive

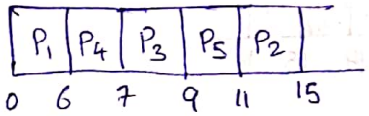
Conflict Resolution : Lower Pid

→ PreEmption of running process is based on availability / arrival of a strictly shorter process

Eg:

PNO	AT	BT	CT	TAT	WT
1	0	6 5	15	15	9
2	1	4 3	10	9	5
• 3	2	2	4	2	0
• 4	3	1	5	2	1
5	4	2	7	3	1

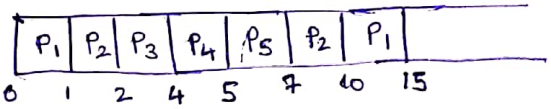
SJF



Avg TAT = $\frac{6+4+7+7+14}{5} = \frac{38}{5} = 7.6$

Avg WT = $\frac{0+3+5+5+10}{5} = \frac{23}{5} = 4.6$

SRTF

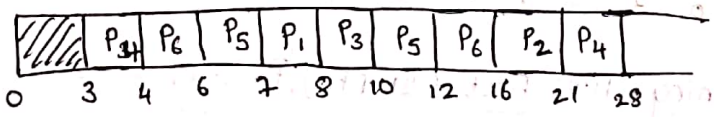


Avg TAT = $\frac{31}{5} = 6.2$

Avg WT = $\frac{16}{5} = 3.1$

Q10

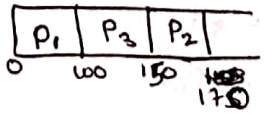
PNO	AT	BT	CT	TAT	WT
✓ 1	• 7	1	8	1	0
✓ 2	• 5	5	21	16	9
✓ 3	• 8	2	10	2	0
4	• 3	8 7	28	25	17
✓ 5	• 6	8 2	12	6	3
✓ 6	• 4	8 4	16	12	6



Q11

PNO	AT	BT
1	0	100
2	25	50
3	50	20

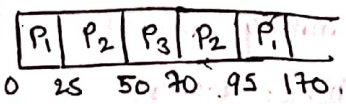
SJF



Avg TAT = $\frac{100+125+125}{3}$ = 350/3

Avg TAT = 115

SRTF



Avg TAT = 86.6

Note:

> Avg TAT (SRTF) ≤ Avg TAT (SJF),
> under the assumption that δ = 0

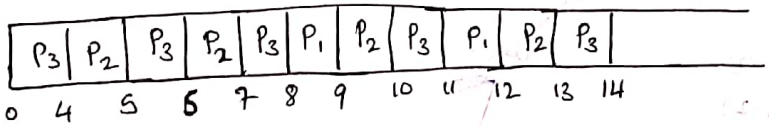
→ No of scheduling overheads are more in SRTF.

(Q12) Compute the Avg TAT for the following processes using LRTF scheduling assuming that in case of a conflict b/w processes then favour the process having lower pid always.

↳ condition holds while preemption also

Sol:

PNO	AT	BT	BT
1	0	2	2
2	0	4 ~~2~~	4
3	0	8 ~~7~~ 1	8



$$\text{Avg TAT} = \frac{12+13+14}{3} = \frac{39}{3} = 13$$

Performance of SJF:

Advantages:

→ optimal algorithm
→ Maximum throughput
→ Minimum Avg TAT & Avg WT

Disadvantage

→ Causes starvation to longer processes

→ One limitation is that SJF is not practically possible to implement. Its because we can't know the burst time of a process before it executes.

→ SJF is used as a benchmark to measure/compare performance of algorithms w.r.to SJF

→ However, we can implement SJF with predicted burst times.

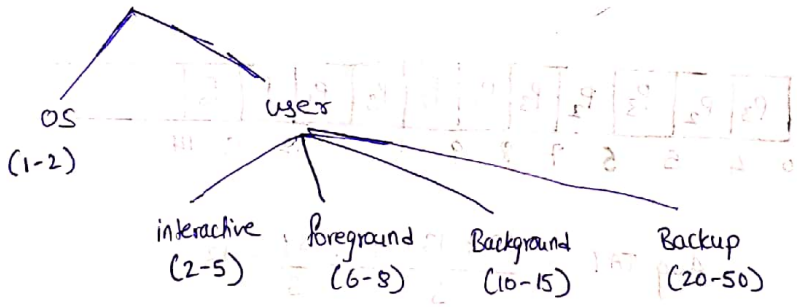
Prediction Techniques of Burst times:

Static Techniques:

(i) Size of process:

If any other process of same size (approx) ran before then based on its BT we predict BT of new process.

(ii) Type of process



Based on type of process we predict a BT for each type.

Dynamic Prediction Techniques

→ we predict only partial Burst time
i.e., next CPU burst

06/8/20

Exponential Averaging technique (Aging algorithm):

1) Let P_i be process
2) Let t_i be complete B.T of process
3) Let τ_i be predicted B.T
4) Let τ_{n+1} be next predicted B.T

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n, \quad 0 \le \alpha \le 1$$

The parameter α controls the relative weight of recent and past history in our prediction

Now

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\left[\alpha t_{n-1} + (1-\alpha)\tau_{n-1}\right]$$

$$= \alpha t_n + \alpha(1-\alpha) t_{n-1} + (1-\alpha)^2 \tau_{n-1}$$

$$\tau_{n+1} = \alpha t_n + \alpha(1-\alpha) t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + (1-\alpha)^3 \tau_{n-2}$$

* Given the value of α, τ_1 & previous BT's we can find any τ_{n+1}

→ All the above approach is for a single process and partial bursts of the process.

(Q13) Given $\alpha = 0.5$, $\tau_1 = 10$

Compute next CPU BT of the process, if its previous runs has BTs of 8, 12, 14, 10.

Sol:

$$\tau_5 = \alpha t_4 + \alpha(1-\alpha) t_3 + \alpha(1-\alpha)^2 t_2 + \alpha(1-\alpha)^3 t_1 + (1-\alpha)^4 \tau_1$$

$$= \frac{1}{2}(10) + \left(\frac{1}{2}\right)^2 14 + \left(\frac{1}{2}\right)^3 12 + \left(\frac{1}{2}\right)^4 8 + \left(\frac{1}{2}\right)^4 10$$

$$= 5 + 3.5 + 1.5 + 0.5 + 0.625$$

$$= 11.125$$

4) Highest Responce Ratio Next (HRRN):

→ This algorithm not only favours shorter processes but also limits the waiting time of longer process.

Criteria: Response Ratio

$$\text{Response ratio} = \frac{w+s}{s}$$

w = waiting time of the process so far

s = service time (Burst time)

Mode: Non-PreEmptive

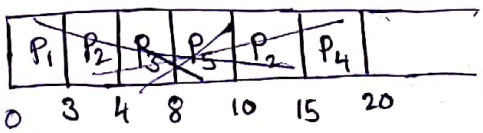
→ $\frac{w+s}{s}$ is high when 's' is less and thus favours shorter processes.

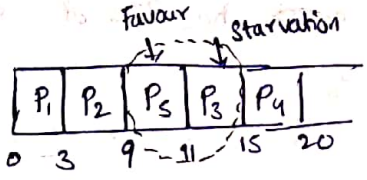
→ $\frac{w+s}{s}$ is high when 'w' is high and thus favours longer BT processes which have been waiting till now.

Eg:

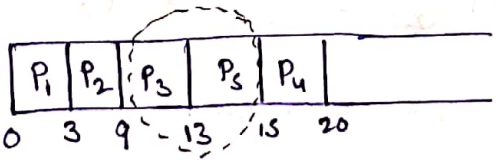
PNO	AT	BT
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

SJF:





HRRN:



Here longer process is favoured and thus starvation is overcome.

at t = 3

$rr_2 = \frac{1+1}{1} = 2$

$rr_3 = 0+0$

at t=9

$rr_3 = \frac{5+4}{4} = 9/4 = 2.25$

$rr_4 = \frac{3+5}{5} = \frac{8}{5} = 1.6$

$rr_5 = \frac{1+2}{2} = \frac{3}{2} = 1.5$

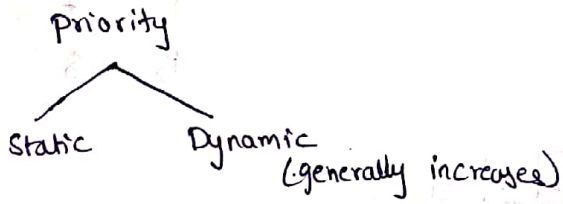
at t=13

$rr_4 = \frac{7+5}{5} = 2.4$

$rr_5 = \frac{5+2}{2} = 3.5$

5. Priority Based Scheduling

* Priority is an attribute of process
* Based on type, size of process and resources the processuse, the priority is computed.
*



SJF is a special case of priority based scheduling. In SJF priority is inverse of next cpu burst.

Criteria : Priority

Mode : Non-PreEmptive or PreEmptive

Conflict resolution : ~~lower~~ ~~FCFS~~ Equal priority processes are scheduled in FCFS order.

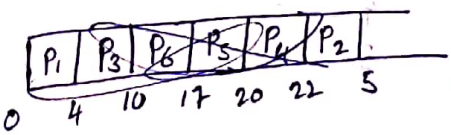
* lower priority processes suffers starvation.

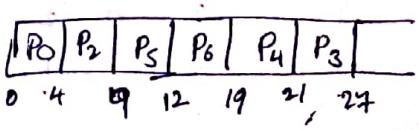
Eg:

Priority	PNo	AT	BT
4	1	0	4
6	2	2	5
5	3	3	6
8	4	5	2
12	5	7	3
15	6	10	7

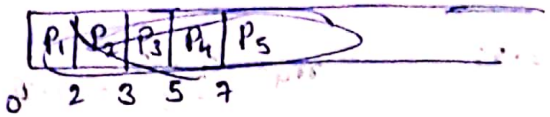
Assuming high number is higher priority

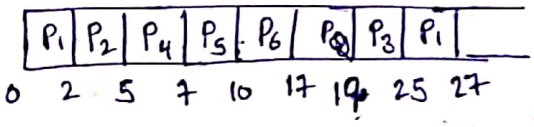
Non PreEmpt





Pre Emptive

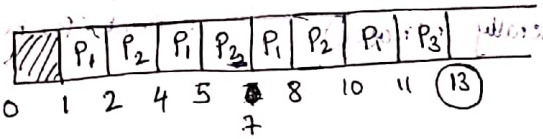




One solution from preventing lower priority process from starving is aging.
Aging involves gradually increasing the priority of processes that have been waiting for long time.

H1/3 ① P_1 - (1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31 ...)
② P_2 - (1, 8, 15, 22, 29, 36, 43, ...)
④ P_3 - (1, 21, 41 ...)

PNO	AT	BT
P_1	1	4
P_2	1	2
P_3	1	4
P_1	4	1
P_1	7	1
P_2	8	2



6) Round Robin Scheduling (PreEmptive FCFS):

* It is mostly used in (Multiprogrammed + Time shared OS)

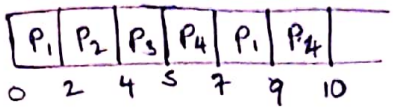
Criteria: AT + Time Quantum
Mode: Pre Emptive
Ready Queue: FIFO
Conflict resolution: lower pid

→ Running process is preEmpted based on time quantum.
* → Time quantum doesn't include time spent in context-switching. only the time spent for process execution is considered.

Eg:

PNO	AT	BT	CT	TAT	WT
1	0	4	9	9	5
2	0	2	4	4	2
3	0	1	5	5	4
4	10	3	10	10	7

Time Quantum = 2



$$\text{Avg TAT} = \frac{28}{4} = 7$$

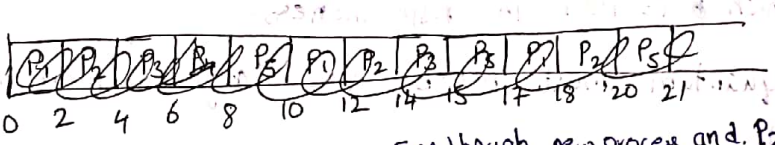
→ Generally Avg.WT is high is RR scheduling.

(Q14)

P.NO	AT	BT
1	0	5
2	1	6
3	3	3
4	4	2
5	7	5

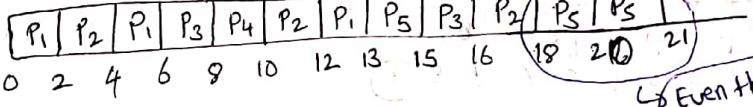
Assume TQ=2 and prepare the gantt chart and find Avg. TAT

Sol:



Ready Q: P1, P2, P1, P3, P4, P2, P1, P5, P3, P2, P5, P5

Even though new process and P2 both are came at the same time, the new process is given more priority.



Even though we have only P5 here still context switch must take place. It is cuz context switch is an interrupt

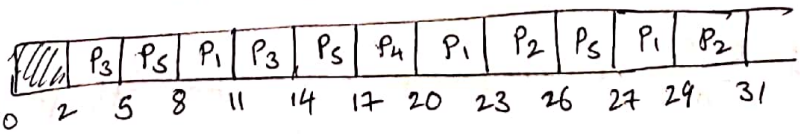
$$\text{Avg TAT} = \frac{13+17+13+6+14}{5} = \frac{63}{5} = 12.6$$

(Q15)

PNO	AT	BT
1	5	5 2
2	12	5 2
3	2	6 3
4	10	3
5	3	4 1

TQ=3

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



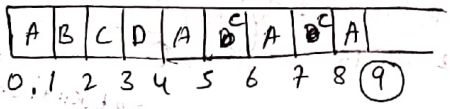
Performance of Round Robin:

* depends on time quantum

Small	Large
* More Context Switches (more overhead)	* Less Context Switching
* Improves interactiveness	* poor interactive and less response

* when time quantum is very large, the round robin becomes close FCFS which is poor in responsiveness.
* so time quantum must not be very large or must not be very small.

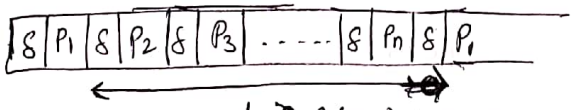
(H1/4)



(H1/5)

Context switch time = s

TQ = q



$t = q(n-1) + s(n)$

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

$q > \frac{t-ns}{n-1}$: Every process gets its next turn only after waiting for atleast 't' sec

$q < \frac{t-ns}{n-1}$: Every process is executed atleast once within every interval of t sec.

(Hr/8) I. $\beta > \alpha > 0$

Consider a process is scheduled.

Its priority ↑ at the rate of β

Priority of process in RQ ↑ at the rate of α

$\therefore \beta > \alpha$, there is no way that priority of processes in RQ goes higher than priority of running process and hence no preEmption.

* Every new process will have 0 priority which is less than that of previously waiting processes.
* The more the procers waits, the higher the priority is.

$\therefore$ FCFS

II. $\alpha < \beta < 0$

Here since they are -ve we decrese the priorities.

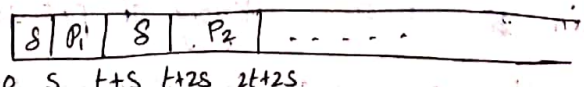
The priority of waiting processes are decreased at higher rate than that of process which is running and hence no ready process can preEmpt running process

However a new process with priority '0' can preEmpt running process

$\therefore$ Last Come first Serve, with PreEmption.

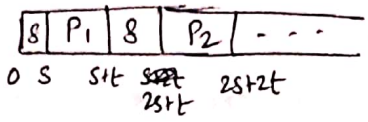
(Hr/7) $T_Q = Q$ $S = s$ Process run $= t$

a) $Q = \infty$



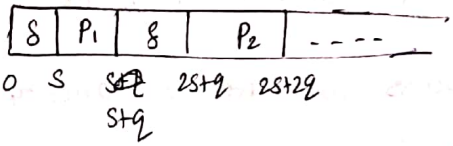
$$\text{Eff}(\mu) = \frac{\text{useful work}}{\text{total time}} = \frac{t}{s+t}$$

b) $q > t$



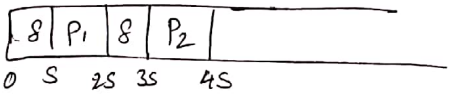
$\mu = \frac{t}{S+t}$

c) $S < q < t$



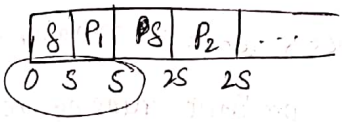
$\therefore \mu = \frac{q}{S+q}$

d) $q = S$



$\mu = \frac{S}{S+S} = \frac{1}{2}$ i.e., 50%

e) $q \cong 0$



$\mu = \frac{0}{S} = 0$

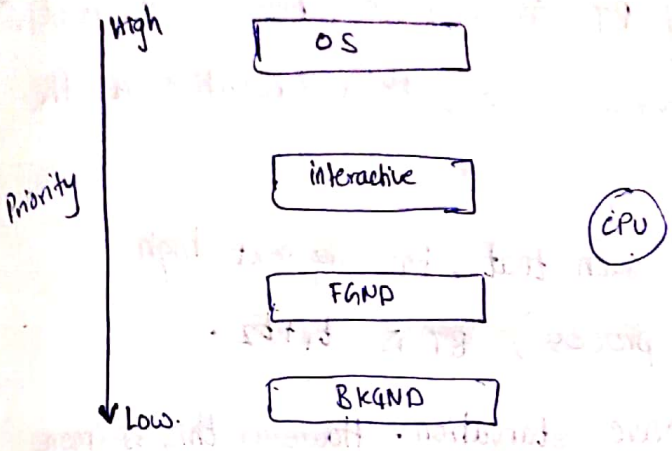
7. Multilevel Queue Scheduling:

Using single queue, to search a particular type of process it takes time. Also we can use only one scheduling technique.

→ To overcome this problem we use multiple queues.

→ Each queue contain a one category of processes.

→ since we have more than one queue, we may use different scheduling algorithms can be used for each queue.

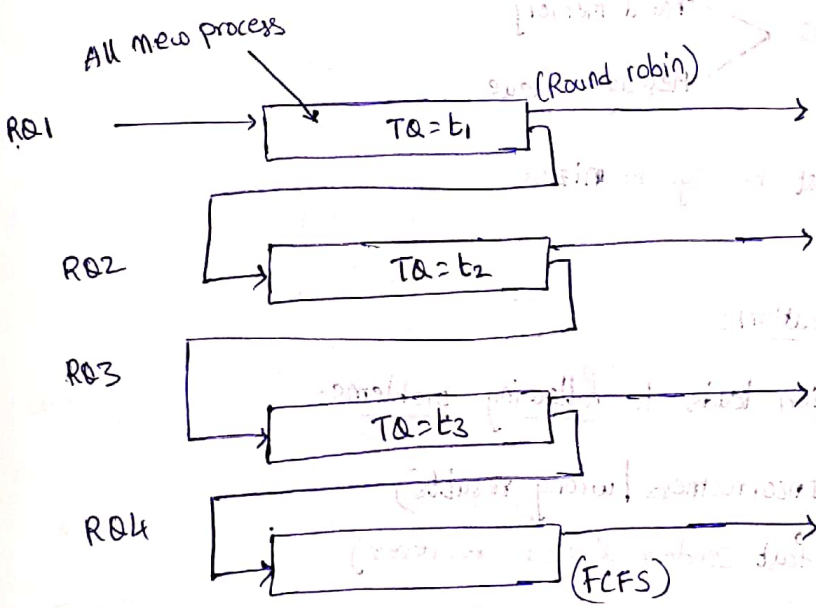


lower priority processes will run only if higher priority processes' queues are empty

Disadvantage:

Processes in lower level queues (low priority) may suffer starvation.

8. Multilevel feedback Queues



Generally, $(t_1 < t_2 < t_3)$

→ All types processes enter RQ1. They will be serviced for time t_1 and if its total BT > t_1 then it is fedback to RQ2. In that way if process for RQ2 are fedback to RQ3 and so on.

Process of RQ2 are scheduled only if RQ1 is empty

→ The last queue has no TQ. since it needs to ensure completion So FCFS is used at the last queue.

→ we choose t_1 such that avg BT of OS processes (high priority) is t_1. So that all high priority processes finishes their execution at the 1st queue itself.

→ Then ~~& then~~ t_2 is choosen such that, the ~~nxt~~ next high priority process's (interactive process) BT is t_1+t_2.

*→ This algorithm may also have starvation. However this is more interactive than multilevel queue scheduling.

Process Synchronization:

IPC: Inter Process Communication

IPC mechanisms < shared memory
Message Queue

→ Communication must be synchronized.

08/08/20

Need for Synchronization:

Lack of synchronization leads to following problems:

1. Inconsistency (Incorrectness | wrong results)
2. Loss of data (fast sender & slow reciever)
3. Deadlocks

Types of Synchronization

i) Competition Synchronization:

Two or more processes are said to be in competion synchronization, iff they compete/contend for accessing a shared variable

Eg: Using a shared variable.

→ Lack of competion synchronization leads to inconsistency & dataloss.

(ii) Co-operation Synchronization

Two (more processes are said to be in co-operative synchronization iff they get affected by each other.
i.e., execution of one process affects the other process.

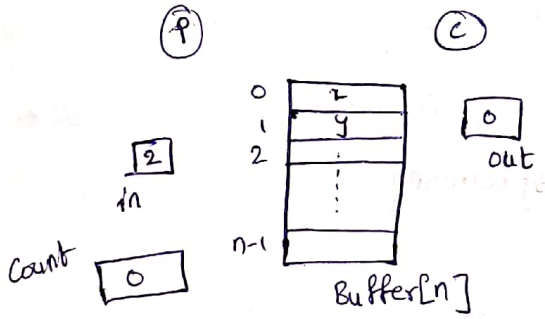
Eg: Producer - Consumer Problem.

→ Lack of co-operation synchronization leads to deadlocks.

→ If processes are not communicating with each other, such processes are said to be independent processes.

Case-Study I : Producer Consumer:

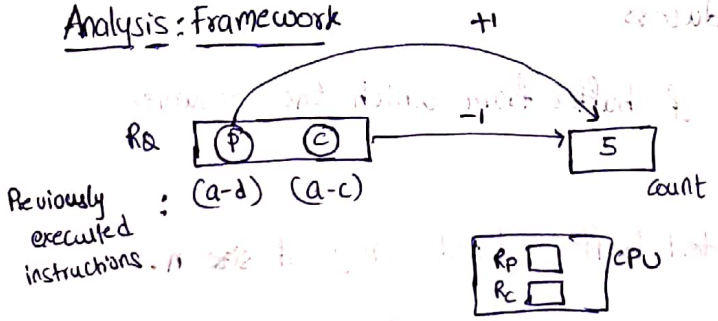
Produce & Consumer are two processes



→ Producer produces data & places it in the buffer.
→ Consumer consumes the data produced by producer.
→ variable 'in' says index of buffer in which the data has to be placed by the producess
→ variable 'out' says index of buffer from which the consumer consumes that data
→ Here the buffer is bounded buffer (fixed size) of size n.
→ life cycle of producer & consumer is infinite.
→ Here we assume the buffer is circular buffer.

```
#define N 100 // size of buffer
int buffer[N], count = 0;
void producer (void)
{
    int itemp, in = 0;
    while (1)
    {
        a) producer_item (itemp)
        b) while (count == N); // Busy waiting
        c) buffer[in] = itemp;
        d) in = (in+1) % N; // circular buffer
        e) count = count + 1;
    }
}
void consumer (void)
{
    int itemc, out = 0;
    while (1)
    {
        a) while (count == 0); // busy waiting
        b) itemc = buffer[out];
        c) out = (out+1) % N;
        d) count = count - 1;
        e) process_item (itemc);
    }
}
```

Analysis : Framework



At time t_1:

Assume producer is scheduled

t_1: I;

R_p [5]

II;

R_p [6]

Assume Producer is preEmpted now.

count = count + 1

I. load R_p, count

II. Inc R_p

III. Store count, R_p

count = count - 1

I. load R_c, count

II. Inc R_c

III. Store count, R_c

At time t_2:

Assume consumer is scheduled

t_2: I;

R_c [5]

II;

R_c [4]

Now, no matter how the rest of the code is executed the final value in count will be either 4 or 6, but not 5.

This inconsistency and thus the code is incorrect.

→ Here the producer & consumer are in competition synchronization and since they aren't synchronized, it led to inconsistency.

Necessary Conditions for Synchronization problems to arise in IPC Env:

Critical Section: It is the part of program/process where share resources are accessed.

Race Conditions: It is the condition in which two or more process race for accessing critical section.

PreEmption: If there is no preEmption, then we will not have any problem. even if the above two conditions are met.

→ ~~This problem of not having synchronization is called critical section problem.~~

Requirements of CS problems:

a) **Mutual Exclusion:**

No two process may be present in their CS at the same time.

b) **Progress:**

No process running outside the critical section should block the other interested processes from entering CS.

c) **Bounded-waiting:**

No process has to wait forever to access its CS, when it is interested.

* If bounded waiting is not ensured, it may lead to starvation.

Assumptions:

→ Each process spends finite time in CS and come out

→ If a process is in CS, others have to wait.

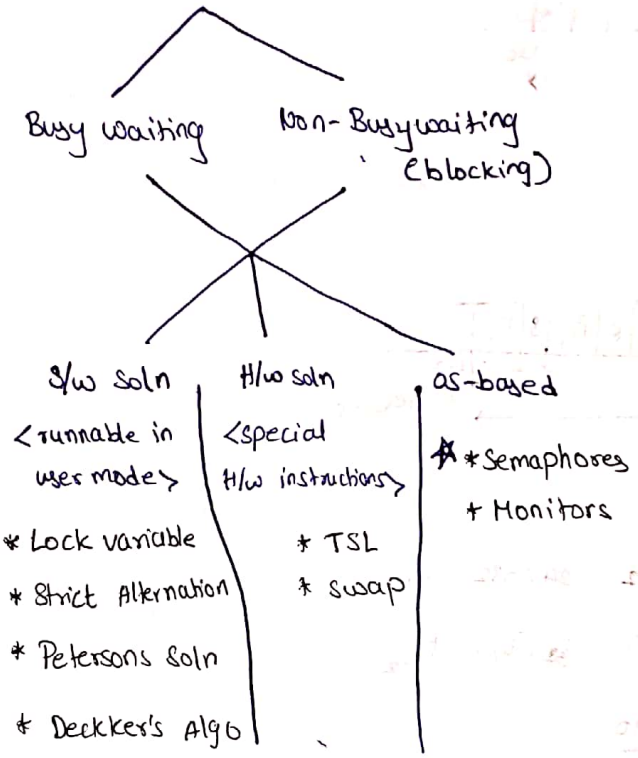
→ A process is said to leave CS, only if it has executed exit section.

→ Process can get preEmpted while executing entry section or in CS or exit section or just after coming out of CS.

→ whether preEmption happens or not the synchronization mechanism must always satisfy M.E, progress, Bounded waiting.

→ when multiple processes are trying to enter CS, processes may be allowed to enter CS in any order.

Synchronization Mechanisms (SM)



Response time:

Each process may be associated with several requests.

→ Every request has its own starting time and may take some time for completion

→ Time from submission of request to the time it generates its response is known as response-time.

→ The point at which response occurs will be specified in the question itself.

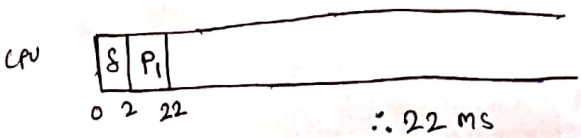
(H1/6) $n=10$ $<P_1, P_2, P_3 \ldots P_{10}>$

P_i : 20 requests

$S=2ms$ $TQ=20ms$

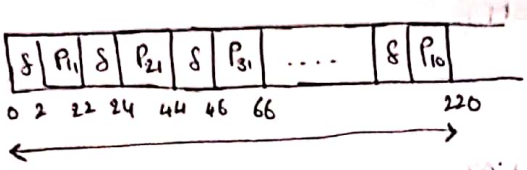
(i) R.T(P_{11}):

RQ: $P_1, P_2 \ldots P_{10}$



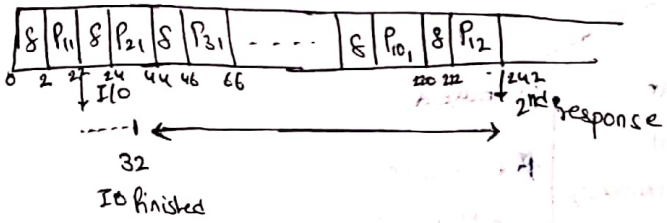
∴ 22 ms

(ii)



∴ 220ms

(iii)



∴ Response time of P_{12} = 242 − 32 = 210

Same is for P_{22}, P_{32}, P_{42} ... P_{102}

∴ 210

09/08/20

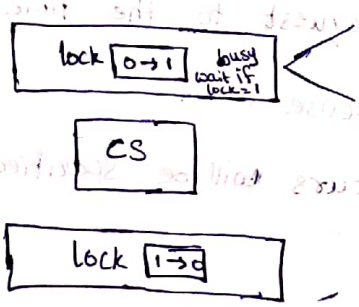
1. Lock Variable Sync. Mechanism

→ Busy Waiting

→ S/w Solution.

→ Multi process solution.

Ideology:



Code:

```
int lock=0; // initially no process will be in CS. So lock=0.
void process(int i)
{
    while(1)
    {
        a) non-CSC);
```

```
    b) while(lock != 0); // busy waiting  } Entry section
    c) lock = 1;
    d) < Critical Section >
    e) lock = 0; } Exit Section.
  }
}
```

Low level implementation of lock variable:

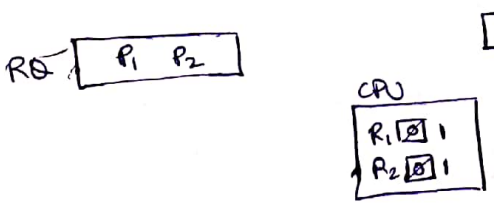
```
lock = 0; // global variable

process segment:

    a) Non_CS
    b) Load Ri, lock      ┐
    c) Cmp Ri, #0         ├ Entry section (busy waiting: b, c, d)
    d) JNZ Step b         ┘
    e) Store lock, #1     ┘
    f) <CS>
    g) Store lock, #0 ] Exit section
```

Analysis:

i) Mutual Exclusion



t_i : (P1) : a; b; c; d; — CS: [ ]

— PreEmption

t_j : (P2) : a; b; c; d; e; f → P2 is now in CS — CS: [P2]

— PreEmption

t_k : (P1) : e; f; → P1 is also in CS — CS: [P1, P2] → More than one process in CS.

∴ It fails to guarantee Mutual Exclusion.

→ However, if no preEmption occurs within entry section (b to e) then ME is guaranteed.

(ii) Progress:

→ Progress is guaranteed.

→ If no process is in cs, the from the code the 'lock' must be '0'. Thus any other process will surely enter the cs if needed.

Hence progress is guaranteed.

(iii) Bounded Waiting:

we say bounded waiting is guaranteed if there is a bound on how many times a process can execute it cs before some other process gets into its cs.

Assume the below scenario for HLL implementation

P_1: a, b, c, d;
——— PreEmpt
P_2: a, b; 2 busy wait
——— PreEmpt
P_3: a, b; 2 busy wait
——— PreEmpt
P_1: e; a; b; c; d;
——— PreEmpt
P_2: a, b; 2 busy wait
⋮

Thus only P_1 may execute forever and there is no bound on how many times P_1 can go into cs before P_2 & P_3 gets their 1st chance.

∴ Bounded waiting is not guaranteed by Lock variable.

∴ For lock variable

Mutual Exclusion	..X
Progress	✓
Bounded waiting	X

Also since lock variable has busy waiting, it results in wastage of CPU time.

→ Any synchronization mechanism which has busy waiting wastes CPU time.

H2/1

a) <Code unrelate to device>

b) Repeat

c) until busy = false ; } busy waiting } Entry section

d) busy = true;

e) <code relate to device use> - CS

f) busy = false; } Exist section

g) <code unrelated to device>

This is nothing but lock variable and hence we can directly say ME is not guaranteed

busy: false $\xrightarrow{P_2}$ true $\xrightarrow{P_1}$ true

t_1: P_1 : a; b; c;

Here P_1 see busy is false and is about is execute d

t_2: P_2 : a; b; c; d; e; (P_2 in cs) cs P_2

t_3: P_1 : d; e; (P_1 in cs) cs P_1, P_2

∴ ME is not guaranteed

also CPU time is wasted in busy waition

∴ opt (e)

2. Strict Alternation

→ Strictly on alternate basis processes take their turn to enter CS.

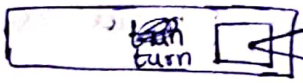
→ Busy waiting

→ Slw solution

→ 2-process solution (Pi & Pj)

Ideology :

Entry

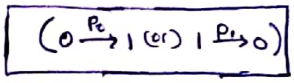


0: Po can enter CS

1: Pi can enter CS

CS

Exit

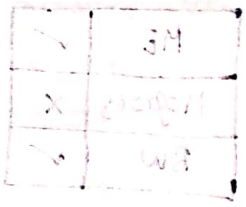


Code

```
int turn = rand(0,1);
void process(0)
{
    while(1)
    {
        a) Non_CS;
        b) while(turn != 0); // busy wait } Entry section
        c) <CS>
        d) turn = 1; } Exist section
    }
}
void process(1)
{
    while(1)
    {
        a) Non_CS;
        b) while(turn != 1);
        c) <CS>
        d) turn = 0;
    }
}
```

general code for two process P_i, P_j :

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



Analysis :

(i) Mutual Exclusion :

→ If turn = i, no matter where P_i or P_j gets preEmpted only P_i can enter cs.

Slly if turn = j, only P_j can enter cs.

∴ ME is guaranteed by strict alternation.

(ii) Progress ;

Assume turn = 0

Assume P_0 is in non_cs and not interested in going to cs.

Still P_1 can't go to cs since turn = 0.

So P_0 is blocking P_1 from entering cs.

∴ Progress is not guaranteed by strict alternation.

(iii) Bounded Waiting :

Assume turn is '0' and both P_0, P_1 are interested in entering cs.

since turn is '0', only P_i can enter.

After execution of cs of P_i, turn is set to 1.

→ Now even if P_0 is interested in reentering cs, it can't until P_1 finishes.

→ which means there is bound on how many times a process can enter cs before other process does. i.e., 1 time.

∴ Bounded waiting is guaranteed by strict alternation.

∴ For strict alternation

ME	✓
Progress	x
BW	✓

(H2/2) The given soln is similar to strict alternation.

∴ progress is violated.

3. Peterson's Solution

→ Busy waiting

→ s/w soln

→ 2-process soln.

* It is combination of lock variable & strict alternation.

Data structures used:

```
#define N 2 // no of processes
#define TRUE 1
#define FALSE 0
int flag[N] = {FALSE};
int turn;
```

→ Each of the two process are associated with one flag variable each. using this processes show their interest of entering cs by setting flag to true.

* → If both the processes are interested in entering cs, then we resolve this conflict using the variable turn.

* → ~~Also in that case we let the which ever process sets turn 1st~~

Code:

~~shows the interest 1st to enter the cs first~~

~~setting flag~~

```
void process (int i);
{
    int j = NOT(i);
    while (1)
    {
        a) non_cs();
Entry   b) flag[i] = ~~true~~ TRUE;
section c) turn = i;
        d) while (turn == i && flag[j] == TRUE); // busy waiting
        e) <cs>
Exit    f) flag[i] = FALSE;
section
    }
}
```

If two processes are interested in entering cs, we let whichever process sets turn 1st, to enter cs.

(turn == i : says P_j has set turn 1st)

(flag[j] == TRUE : says both the processes interested in entering cs)

we can even have c & d stmts as
c) turn = j;
d) while (turn == j && flag[j] == T);

Analysis:

Mutual Exclusion:

RQ | P_0 P_1 |

flag[0] = ~~F~~ T

flag[1] = ~~F~~ T

turn | 1 |

t_0 : P_0 i=0, j=1 : a; b;

---

t_5 : P_1 i=1; j=0; a; b; c; d ↺ busy wait

---

t_7 : P_0 i=0; j=1; c; d ↺ busy wait

---

t_{10} : P_1 i=1; j=0 d; e; (P_1 enters cs since it is the 1st one to set the turn)

→ Thus peterson soln guarantees ME always

Progress:

whenever a process is not interested in entering cs, then interested process can ~~see its~~ that the flag value of the uninterested process is false and can thus enter cs.

∴ Hence, progress is also guaranteed.

Bounded Waiting:

In the ~~previous~~ analysis presented in mutual exclusion, consider P_i finished its execution. Now if P_i is again interested in entering cs then it sets flag to true and modifies turn value.

However, P_0 will be the first one to modify turn and hence P_0 will enter cs. Hence peterson's soln guarantee's bounded waiting.

So for peterson's soln

Mutual exclusion	✓
Progress	✓
Bounded waiting	✓

Thus peterson's soln is one of the correct solutions.

Limitations:

(i) Limited to work with only 2-processes.

(ii) It is busy waiting soln ⇒ wastage of cpu time.

(Q16) In peterson's soln, if line 'd' is replace with 'below line

```
while(flag[j]==T);
```

then what would happen?

Ans: There is a possibility for deadlock.

Synchronization Hardware

a) TSL b) SWAP

→ H/w solns
→ busy waiting
→ Multi process solutions
→ These are extensions of lock variable.
→ TSL & SWAP are atomic (H/w) instructions.

TSL : Test and Set Lock

a) TSL:

Syntax : TSL (&flag);

process when executes TSL, returns current value of lock through variable flag and sets the value of lock through flag, always, to true.

(i) if lock = 0, then TSL returns 0 and sets lock to 1.
(ii) if lock = 1, then TSL return 1 and sets lock to 1.

Implementation of TSL:

```
Boolean TSL (Boolean *flag) // flag points to lock.
{
    int rv;
    atomic execution {
        rv = *flag;
        *flag = TRUE;
        return rv;
    }
}
```

* TSL is called lock based mechanism.

```
#define FALSE 0
#define TRUE 1

Boolean lock = FALSE;
void process (int i)
{
    while (1)
    {
        Non_cs();
    Entry [ while (TSL(&lock) == TRUE);
        <cs>
    Exit [ lock = FALSE;
    }
}
```

Analysis:

(i) Mutual Exclusion:

Lock variable doesn't guarantee ME because there is a chance that it could preEmpted after while loop in the entry section.

However, TSL has the two instruction of the entry section of lock variable, as an atomic instruction, it guarakees ME.

(ii) Progress:

Lock variable guarantees progress.

So ~~this~~ TSL also does.

(iii) Bounded waiting:

Bounded waiting is not guaranteed.

(Same explanation like in the case of lock variable)

However for the below implementation of TSL, bounded waiting is guaranteed.

→ Here we have array waiting[n];

∞ If waiting[i] is true, it means that P_i is waiting to enter cs.

variable key holds the value of lock key: false (cs is free) / true (cs in use)

Code:

```
waiting[i] = {false};   } initial values
lock = false;
do
{
    waiting[i] = true; // waiting to enter CS
    key = true;
    while (waiting[i] && key) // busy wait.
        key = TSL(&lock);
    waiting[i] = false; // wait is over

    <CS>

    j = (i+1) % n;
    while (j != i && waiting[j] == false) // checking if any other process is waiting
        j = (j+1) % n;
    if (j == i) // No process is waiting
        lock = false;
    else
        waiting[j] = false; // Process Pj is waiting

} while (true);
```

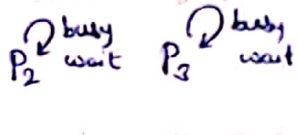
Mutual Exclusion: working:

The first process which wants to enter cs will find the lock value to be false and will enter cs.

While the 1st process is in cs, assume we have another process P_2 which wants to enter cs. Now waiting[2] = true and key = true and thus P_2 will busy wait.

→ Assume P_3 has arrived and it also wants to enter CS.

Like P_2, P_3 will also busy wait.

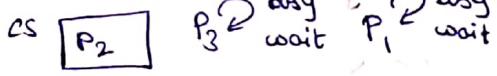


→ Once P_1 finishes its CS, it checks if waiting[] of any other process is true. If no process is waiting then lock is reset.

→ However here we have P_2 waiting i.e., waiting[2] = true

So waiting[2] is set to false P_1 and thus P_2 will now enter CS.

→ Assume now P_1 again wants to enter CS. Now P_1 busy waits.



→ Once P_2 finishes its CS, it checks if any other process wants to enter CS and thus waiting[3] is set to false and thus P_3 enters CS.

→ Andes later P_1 enters CS.

→ Here the processes enter CS in the order ~~in which they show the inte~~ of $(i+1, i+2 \cdots n-1, 0, 1, 2, \cdots i-1)$

where P_i is currently ~~exec~~ executing CS.

Mutual Exclusion :

A process ~~can~~ P_i can enter CS only if

waiting[i] = false or key = false

key will be false only if no process is in CS.

waiting[i] will be false, iff some process exists CS.

Thus ME is guaranteed.

Progress:

If no process is interested in entering cs then value of key will be false.

Thus any interested process can enter cs.

Hence progress is guaranteed.

Bounded waiting:

Assume process P_i is executing in cs. After P_i exists cs it gets chance to enter es again only after processes $P_{i+1}, P_{i+2} \cdots P_{n-1}, P_0, P_1, \cdots P_{i-1}$ enter their es if they are interested.

Thus Bounded waiting is also guaranteed.

∴ For TSL based on the way we implement, the correctness depends.

b) SWAP: (XCHG)

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

* SWAP is called lock-key based mechanism.

```
#define FALSE 0
#define TRUE 1
Bool lock = FALSE;
void Process (int i)
{
    Bool key;
    while(1)
    {
        Non_CS();
Entry   do
        {
            SWAP(&lock, &key);
        }while(key == TRUE); // busy wait
        <CS>
Exit    lock = FALSE;
    }
}
```

key = TRUE;

Mutual Exclusion:

→ Every process before entering CS sets lock to true.

→ Thus if another process wants to enter CS, it is not allowed until the lock is made false

∴ ME is guaranteed.

Progress:

→ If no process is interested in entering CS then lock will be false. Thus any other interested process can enter CS.

∴ Progress is guaranteed.

Bounded waiting:

→ The above particular design does not guarantee bounded waiting.

→ However like in the case of TSL we can extend the above implementation to guarantee bounded waiting.

Note:

* In general for TSL & SWAP

ME	✓
Progress	✓
BW	✓

Note:

→ Peterson's Algo and Dekker's algo are correct Slw solns. (2-process)

→ TSL & SWAP are correct Hlw solns. (Multiprocess)

→ All the four of them suffers from busy waiting

*→ Busy waiting leads to two problems:

(i) Wasting CPU time.

(ii) Priority Inversion problem.

Priority Inversion

→ This type of problem occurs if the CPU scheduling algo used is PreEmptive priority based.

→ Assume we have two processes P_L (low priority) & P_H (high priority)

Consider initially we have only P_L in RQ. Let us assume P_L want to enter CS and it entered. while P_L is in CS assume P_H has arrived into RQ. Now P_L is preEmpted and P_H is scheduled.

Now P_H can't enter CS due to synchronization mechanism (Peterson (or) Dekker or TSL or SWAP). Also P_H can't leave CPU due to scheduling algorithm.

Hence ~~deadlock~~ occurs.
(Livelock)

Lock

Spin lock

i.e busy wait

The lock variable that uses busy waiting is called a spin lock.

Live lock

In the case of livelock the state of the processes is either running or ready but not blocked

dead lock

In the case of deadlock the state of the processes may be blocked

Priority Inheritance:

→ Priority inheritance is soln for priority inversion problem.

→ Here P_L temporarily gains the priority equal to that of P_H and thus P_L can be scheduled for CPU and later after the execution of the cs of P_L it's old priority is given back.

10/08/20

Non-Busy Waiting / Blocking Mechanisms

we have 3 such mechanisms

a) sleep() & wakeup():

→ These are OS primitives

→ Multi process soln

Entry section

```
if (cs is free) then
    <cs>;
else
    sleep();
```

Exit section

```
if (there is/are blocked processes in queue)
{
    wakeup(Q.process);
}
```

Producer & Consumer problem using sleep() & wakeup():

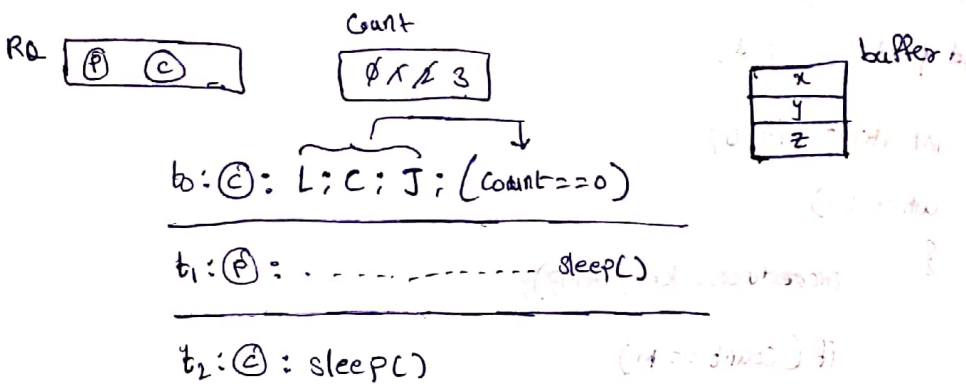
```
#define N 100
int buffer[N], count=0;

void producer(void)
{
    int itemp, in=0;
    while(1)
    {
        produce_item(itemp);
        if(count==N)
            sleep();
        buffer[in]=itemp;
        in=(in+1)%N;
        count=count+1;
        if(count==1)
            wakeup(consumer);
    }
}
void consumer(void)
{
    int itemc, out=0;
    while(1)
    {
        if(count==0)
            sleep();
        itemc=buffer[out];
        out=(out+1)%N;
        count=count-1;
        if(count==N-1)
            wakeup(producer);
    }
}
```

The two process are in both co-operative synchronization and in competive synchronization.

The above implementation may lead to deadlock. The below is illustration for that.

Let N=3



Now both the processes are in sleep() (i.e., blocked)

This is a deadlock (but not livelock).

One soln to avoid this deadlock is using a wakeup bit. Refer Tanenbaum for detailed explanation.

This deadlock has occured due to incorrect co-operation.

Also implementation has inconsistency due to incorrect competitive synchronization.

Semaphores

→ Non-Busy waiting Mechanism.

→ OS resource.

→ Multi process soln

→ General purpose utility

i.e., it is not only used for soluing critical section problem but also used in concurrency mechanisms and in IPC.

→ Semaphore is implemented as an ADT

→ As an ADT, semaphore supports two operations

Down (P) (wait)

up (V) (signal/release)

→ Semaphore operations are atomic (i.e., no preEmption)

Formal defn: Semaphore is a variable (ADT) that takes only integer values;

Based on the value that a semaphore takes, it is classified into two types

i) Binary Semaphore (0 or 1)

ii) Counting Semaphore ($-\infty$ to ∞)

→ Since counting semaphore can also take values 0 (or) 1, any problem that can be solved by binary semaphore can also be solved using a counting semaphore

→ However, counting semaphore can also be implemented using binary semaphore.

→ Binary semaphores are also called as mutex locks.

Counting Semaphore:

Strong Semaphore: Semaphore with FIFO Queues

Weak Semaphore: " without " "
i.e., with some other Queue

```
typedef struct
{
    int value;
    QueueType L; // consists of list of processes that got blocked while
                 performing 'DOWN' operation unsuccessfully.
} CSEM;
```

→ Any semaphore variable must be initialized with a value before we use it.

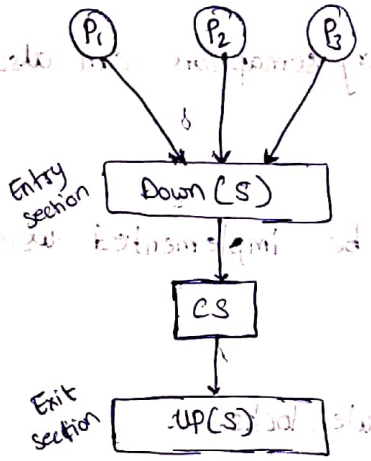
```
DOWN(CSEM S)
{
    S.value = S.value - 1;
    if (S.value < 0) // unsuccessful down
    {
        put this process's PCD in S.L(Q) & block it (sleep)
    }
    else
        return;
}
```

→ If S.value = 4, then we can say that 4 Processes can perform down operation successfully.

+ve val of semaphore ⟶ those many processes can perform down

-ve val of semaphore ⟶ the no of blocked processes.

Solving the CS problem: using semaphore



Here we must let only one process to enter CS.

So we must initialize S.value to 1.

```
SEM S;
```

S.value = 1; S 1 L P2, P3 block Ques

P1 : D(S); S 0

P2 : D(S); S -1

P3 : D(S); S -2

P1 : <cs> ; U(S) S -1

P2 : <cs> ; U(S) S 0

P3 : <cs> ; U(S) S 1

```
UP(CSEM s)
{
    s.value = s.value + 1;
    if (s.value <= 0) // checking if there is any blocked process
    {
        select a process found s.L(Q) & unblock it (wakeup);
    }
    else // no block processes.
        return;
}
```

→ A process may get blocked while performing Down/P operation & never gets blocked while performing Up/V operation.

(Q17) A counting semaphore is initialized to 5. Then the following operations were completed on it in the given order.

15P; 3V; 18P; 4V; 10P; 5V

The current value of semaphore is ——

Sol:

$S=5 \xrightarrow{15P} -10 \xrightarrow{3V} -7 \xrightarrow{18P} -25 \xrightarrow{4V} -21 \xrightarrow{10P} -31 \xrightarrow{5V} -26$

i.e., right now we have 26 blocked processes in the queue

(Q18) With some initial value of semaphore, following operations were computed on it.

10P; 4V; 6P; 5V; 15P; 2V

The present value of semaphore is -10. Find initial value of semaphore

Sol:

$x - 10 + 4 - 6 + 5 - 15 + 2 = -10$

$x + 11 - 21 = 0$

$x = 10$

∴ initial val of semaphore is 10

#2/4

Let initial value is x

$x - 20 + 12$

$= x - 8$

Atleast one process is blocked

i.e., value of semaphore is ≤ -1

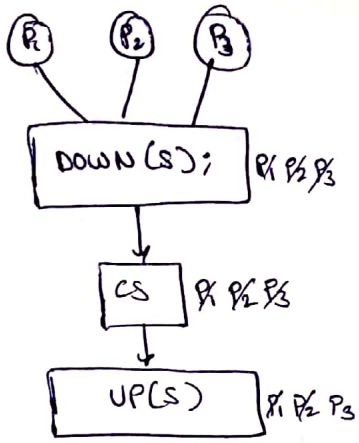
$\therefore x - 8 \leq -1 \Rightarrow x \leq 8 - 1$

$\Rightarrow x \leq 7$

Binary Semaphore:

```
typedef struct
{
    enum value(0,1);// 1 → CS is free, 0 → CS is in use
    QueueType L;// list of PCB's that got blocked while performing DOWN unsuccessfully.
}BSEM;

DOWN(BSEM s)
{
    if(s.value==1) // success
    {
        s.value=0;
        return;
    }
    else //unsuccessful
    {
        put this process (PCB) in s.L(Q) & Block_it()
    }
}
UP(BSEM s)
{
    if(s.L(Q) is not empty) // Blocked processes are present
    {
        select a process from s.L(Q) & wakeup();
    }
    else // No blocked processes
        s.value=1;
}
```



Summary :

* S=1 $\xrightarrow{\text{DOWN}}$ successful & S=0

* S=0 $\xrightarrow{\text{DOWN}}$ unsuccessful & S=0 & process added to Queue.

* S=0 $\xrightarrow{\text{UP}}$ S=0, if there is a blocked process
S=1, if there is no any blocked processes.

* S=1 $\xrightarrow{\text{UP}}$ S=1 (This case may not be possible for solving CS problem, but semaphore are used for other applications too. So it is possible that UP can be performed even when S=1)

* If S=0 and the first operation performed on S is V(S).

Now find S after operation V(S)

Soln:

since it is said that V(S) is the first operation, it means that Queue is empty (if Q is not empty, then some process must have performed P(S) and should go to Q. So the Q is empty)

Thus from the code of UP() we can say that
S will be initialized to 1.

∴ S=1

11/08/20

H2/3 Dekker's Algorithm

```
do
{
    flag[i]= true;
    while(flag[j])
    {
        if(turn==j)
        {
            flag[i] = false;// temporarily looses interest
            while(turn==j);// busy wait
            flag[i] = true;
        }
    }
    /* Critical Section*/
    turn=j;
    flag[i]= false;
} while(true);
```

Analysis :

Here we assume initial value of turn is any of i or j. and initial value of both the flags is false.

→ flag indicates the interest of process in entering cs.

Mutual Exclusion:

If both the processes are interested in entering cs, then initially based the value of turn, the process which should enter the cs is decided.

Progress:

If only one process is interested in entering the cs then seeing the flags of other process as 'false', then interested process directly enters cs.

Bounded waiting:

assume P_0, P_i have shown interest, more or less, at the same time. Let us say turn is 0 and thus P_0 enters CS. when P_0 executes its exit section turn is set to 1. Now even if P_0 wants to enter CS again immediately, P_0 can't do so cuz the turn is 1. Thus bounded waiting is ensured.

(H2/5)

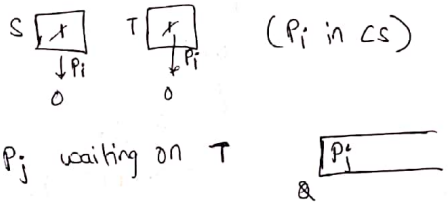
Mutual Exclusion ✓

Progress ✓

Bounded waiting ✓

Assume P_i has entered CS and later P_j also wants to enter CS.

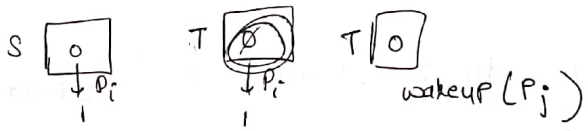
Now P_j waits on semaphore T



once P_i finishes CS, P_j performs up on T and S

up V(T) keeps T as 0 and wakesup P_j

where V(S) finds Q empty and changes S to 1.



Now if P_i immediately wants to enter CS, it finds T as 0 and hence sleep waits in Q until P_j wakes it up.

Deadlock: ✓

P_i : P(S);

P_j : P(T);

P_i : P(T); --- block

P_j : P(S); --- block

} Deadlock

Inorder to remove the deadlock, the order of downs must be same.

i.e.,

```
Pri;                          Prj
{                             {
  while(1)                      while(1)
  {                             {
    P(S);  ---------------------  P(S);
    P(T);  ---------------------  P(T);
    <CS>                          <CS>
    V(T);                         V(S);
    V(S);                         V(T);
  }                             }
}
```

The order of up doesn't matter cuz up never blocks a process

In the same way if we use n semaphores all must have same order.

(H2/6) BSem S=1 T=0

```
Pri;                          Prj;
{                             {
  while(1)                      while(1)
  {                             {
    P(T);                         P(S);
    print('1');                   print('0');
    print('1');                   print('0');
    V(S);                         V(T);
  }                             }
}                             }
```

What is the O/P stream generated?

Ans:

Even if Pri; is scheduled first, it must wait on T.

So Prj will execute prints first

i.e., 00

Prj performs performs up on T

thus not Pri executes

i.e., 11

& :

∴ O/P: 00110011....

Also this implementation is similar to strict alternation mechanism

This also ME ✓
Prog ✗
BW ✓

(H2/7) Bsem m[0...4] = [1]; //array of semaphores

```
Pgi;  i=0..4 // 5 processes
{
    while(1)
    {
        P(m[i]);
        P(m[(i+1)%4]);
        <cs>
        V(m[i]);
        V(m[(i+1)%4]);
    }
}
```

P_0	P_1	P_2	P_3	P_4 :
P(m[0])	P(m[1]);	P(m[2]);	P(m[3]);	P(m[4]);
P(m[1])	P(m[2]);	P(m[3]);	P(m[4]);	P(m[0]);

(i) is Mutual exclusion guaranteed?

Assume P_0 & P_2 wants to enter cs.

Since they both perform down on different semaphores

P_0 : P(m[0]); P(m[1]); <cs>

P_2 : P(m[2]); P(m[3]); <cs>

∴ ME is not guaranteed.

(ii) What is the maximum number of processes allowed into cs directly?

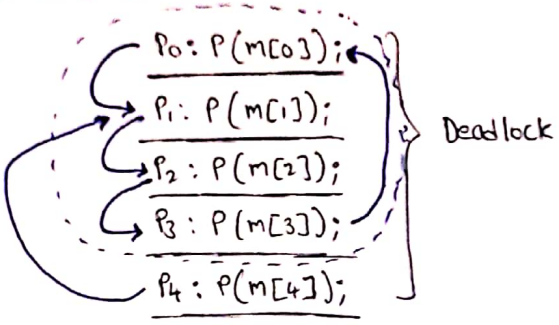
P_0 enters cs | 0 | 0 | 1 | 1 | 1 |

P_2 enters cs | 0 | 0 | 0 | 0 | 1 |

Now no other process can enter cs

∴ Maximum of 2 processes can be in the <cs>

(iii) Does it suffer from deadlock?



Here deadlock occurs when every process executes the first down operation successfully.

(H2/8) Bsem S=1, T=0, Z=0

```
Pri                Prj              Prk
{                  {                {
  while(1)           P(T);            P(Z);
  {                  V(S);            V(S);
    P(S);          }                }
    printf(*);
    V(T);
    V(Z);
  }
}
```

(i) What is the maximum no of * that can be printed.

Only P_i is in infinite loop but not P_k & P_j

P_i : P(S); print(*); V(T);

P_j : P(T); V(S);

P_i : P(S); V(Z);

P(S); print(*); V(T); V(Z);

P_k : P(Z); V(S);

P_i : P(S); print(*); V(T); V(Z);

(ii) what is the minimum no of * that can be printed.

P_i : P(S); print(*); V(T); V(Z);

P_j : P(T); V(S);

P_k : P(Z); V(S);

P_i : P(S); print(*); V(T); V(Z);

∴ minimum of 2 processes

(Q19) Let $P_1 \ldots P_{10}$ Processes.

BSEM mutex = 1;

```
Process(i)  i = 1...9
{
   while(1);
   {
    P(mutex);
     <CS>
    V(mutex);
   }
}

Process(j)  j = 10
{
     V(mutex);
     <CS>
     V(mutex);
}
```

(i) Does this mechanism guarantee mutual exclusion?

Sol:

P_1 : P(mutex); <CS> mutex 0

P_{10} : V(mutex); mutex 1

<CS>

P_1 & P_{10} can be in <CS>

∴ ME is not guaranteed.

(ii) what is max no of processes that can be in CS?

P_1 : P(mutex); <CS>

Mutex 0

P_2 : P(mutex); → waits in Q Q: P_2

P_{10} : V(mutex); → remove P_2 from Q and allows it to enter CS

<CS>

P_2 : <CS>

∴ P_1, P_2, P_{10} can be in CS at the same time

∴ 3

Even if P_{10} comes out only one process can enter CS in its place.

II. Classical IPC Problems: Semaphore based Implementation

(Applications of semaphores)

a) Producer-Consumer problem:

Here buffer should be treated as Critical Section.

```
#define N 100
int buffer[N];
CSEM empty=N; // No of empty slots in buffer
CSEM full=0; // No of filled slots in the buffer
BSEM mutex=1; // used by producer & consumer to access the CS

void Producer(void)
{
    int itemp, in=0;
    while(1)
    {
        a) Produce_item(itemp);
        b) DOWN(empty);          } Entry
        c) DOWN(mutex);
        d) buffer[in]=itemp;     } CS
        e) in=(in+1)%N
        f) UP(mutex);            } Exit
        g) UP(full);
    }
}

void Consumer(void)
{
    int itemc, out=0;
    while(1)
    {
        a) DOWN(full);           } Entry
        b) DOWN(mutex);
        c) itemc=buffer[out];    } CS
        d) out=(out+1)%N;
        e) UP(mutex);            } Exit
        f) UP(empty);
        g) consume_item(itemc);
    }
}
```

Co-operation b/w the processes

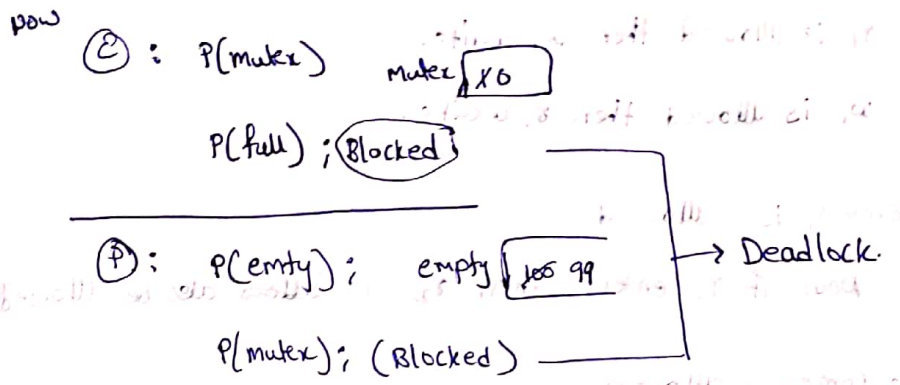
Competitive synchronization.

→ If we change the order of DOWN operations as shown below,

Prod	Consumer
DOWN(Empty);	DOWN(mutex);
DOWN(mutex);	DOWN(full);

Consider below situation where

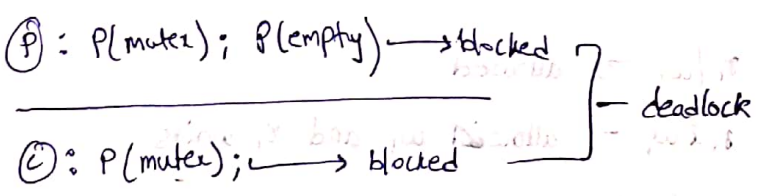
empty = N, full = 0, mutex = 1



→ Even if the order is below one still we will have deadlock

P:	P:	C:
P(empty)	P(mutex)	P(mutex)
Dw(mu	P(empty)	P(full)

Assume empty = 0, full = N, mutex = 1



b) Readers - Writers Problem:

Here we have n readers and m writers accessing records. (n>1 & m>1)

DB is critical section:

(i) First Reader - Writer:

→ Readers get prioritized

t_0: if we have only one reader or one writer then it is allowed.

if we have both reader & writer then any of them may be allowed.

t_1: if r_1 is allowed, then w_1 waits.

if w_1 is allowed, then r_1 waits.

t_2: Assume r_1 is allowed

Now if r_2 enters then r_2 can also be allowed.

t_3: r_3 comes : allowed

i.e., Multiple readers can exist in DB;

i.e., prioritizing readers over writers.

* So writers suffer starvation.

(ii) First Writer - Reader / Second Reader - Writer:

→ writers are prioritized

case(i):

t_0: r_1 / w_1 => allowed

r_1 & w_1 => allowed w_1 and r_1 waits

w_1 DB

t_1: $w_2; r_2; w_3; r_3$ .... all are blocked.

Case (ii)

t_0: r_1 comes : allow

t_1: r_2 comes : allow

t_3: r_3 comes : allow

t_4: w_1 X : w_1 waits

t_5: r_4 X : NOT allowed // to prevent w_1 from starvation.

This waiting of r_4 for readers w_1 not to starve is called principle of ~~starvation~~ transition. (or) principle of turnstile

→ Here we limit starvation to writers.

Implementation of first reader-writer using Semaphores:

Here we have multiple critical sections

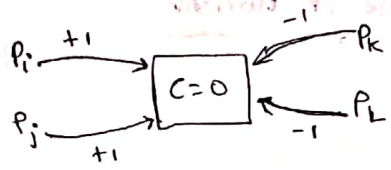
```
int rc=0; // no. of readers currently accessing database
BSEM mutex=1; // used by readers to update rc
BSEM db=1; // used by readers & writers to access DB as CS

void writer(void)
{
    while(1)
    {
        DOWN(db); // writer doesn't need to bother abt how many readers are present. All that it needs to see is if the database is free or not.
        <WRITE_DB>
        UP(db);
    }
}

void reader(void)
{
    while(1)
    {
        a) DOWN(mutex);
        b) rc=rc+1;
        c) if(rc==1) // If this is the 1st reader then it has to lock
            d) Down(db); the database. If it is not, then it can straight away enter CS and access database.
        e) up(mutex); → If the 1st read is block here then all the other readers will block at @ trying to perform down on mutex.
        f) <READ_DB>
        g) DOWN(mutex); → Allowing other readers to perform DOWN on mutex and thus access DB
        h) rc=rc-1;
        i) if(rc==0) // The last reader has to release database.
        j)     up(db);
    } k) up(mutex);
}
```

a) Solving Producer Consumer Problem:

(H2/9)



C = C + 1 ; C = C - 1 ;

a) Load R_0, C	a) Load R_0, C
b) Inc R_0	b) dec R_0
c) Store C, R_0	c) Store C, R_0

Minimum

P_k : a; b; c; c[-1]

P_L : a; b; c[-1] R_L[-2]

P_i : a; b; c; c[0]

P_j : a; b; c; c[1]

P_L : c; c[-2]

∴ -2 is minimum

Maximum :

P_i : a; b; c c[1]

P_j : a; b; c[1] R_j[2]

P_k : a; b; c; c[0]

P_L : a; b; c; c[-1]

P_j : c; c[2]

Implementation of 2nd reader-writer using Semaphores

```
BSEM mutex=1;
int wc=0;
BSEM a=1;
int rc=0;
BSEM b=1;
BSEM db=1;
BSEM mutex
```

```
void writer(void)
{
  while(1)
  {
    DOWN(a);
    wc=wc+1;
    if(wc==1)
      DOWN(b);
    UP(a);
    DOWN(db)
    <WRITE.DB>
    UP(db)
    DOWN(a);
    wc=wc-1;
    if(wc==0)
      UP(b);
    UP(a);
  }
}
```

```
void reader(void)
{
  while(1)
  {
    DOWN(b);
    DOWN(mutex);
    rc=rc+1;
    if(rc==1)
      DOWN(db);
    UP(mutex);
    UP(b);
    <READ_DB>
    DOWN(mutex);
    rc=rc-1;
    if(rc==0)
      UP(db);
    UP(mutex);
  }
}
```

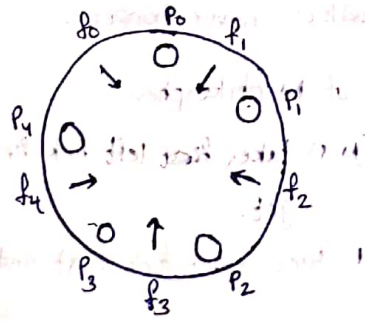
when a writer comes no more new readers are allowed

13/08/20

c) Dining Philosophers Problem:

→ N: Philosophers (P_0 ... P_4, N=5)

are given a research problem



P- philosopher (process)

f- fork

```
#define N 5
void philosopher (int i)
{
    while(1)
    {
        a) think(i);
        b) take_fork(i); // right fork
           take_fork((i+1)%N); // left fork
        c) take_fork((i+1)%N); // left fork
        d) eat(i);
        e) put_fork(i);
        f) put_fork((i+1)%N);
    }
}
```

If all philosophers are hungry at the same time, then deadlock may occur.

$P_0 \leftarrow f_0$

$P_1 \leftarrow f_1$

$P_2 \leftarrow f_2$

$P_3 \leftarrow f_3$

$P_4 \leftarrow f_4$

14/08/20

→ The max no. of philosophers that can be in eating state at a given point of time.

Solution for deadlock in dining philosophers problem:

- Semaphore based solution (Pg: 48 Ace material)
  - * we consider fork as CS
    - taking fork (+)
    - releasing fork (−)
- Non-Semaphore based solution
  - * If we have an additional fork then deadlock never happens.
  - * out of N philosophers
    - (N-1) takes left fork first and then right.
    - 1 takes right fork first and then left

* One more non-semaphore soln is that all even numbered philosophers take ~~even~~ left fork first and all odd numbered philosophers take right fork first.

This soln also guarantees maximum concurrency

i.e., if all the philosophers wants to eat at the same time 2 (max) will be given chance.

But in the 2nd soln only 1 may get chance in certain situations.

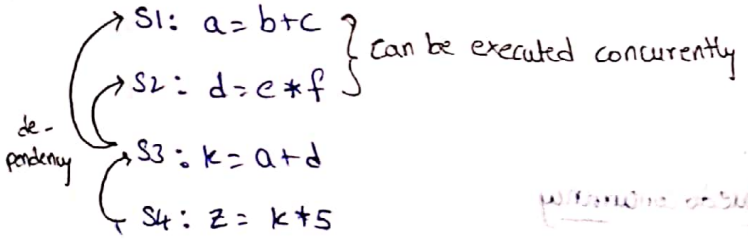
d) Sleeping Barber Problem:

(Refer in Tanenbaum or volume 1)

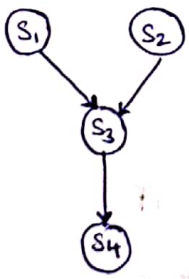
e) Cigarette-Smokers:

(Refer in Galvin (exercise problem))

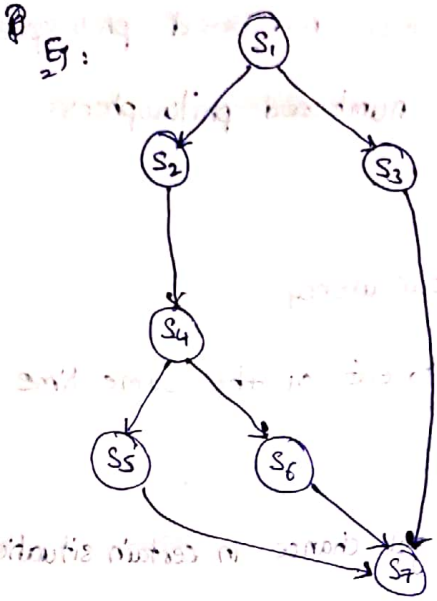
Concurrency Mechanisms:



Precedence graph:



Eg:



→ S_1 is started 1st

→ S_2 & S_3 may run concurrently

→ If S_2 finishes its execution, then S_3 & S_4 also may run concurrently

→ S_5, S_6 & S_3 can also run concurrently

→ S_7 must start its execution only after S_3, S_5, S_6 finishes their execution.

Types of Concurrency

Physical Concurrency

→ Real Concurrency

→ Multi Processor system (es) Multicore architecture

Pseudo Concurrency

→ single CPU system

→ Pre Emptive executions

Concurrency Conditions

* → two stmts S_i & S_j are said to concurrent if

→ if o/p one stmt not i/p of next statement

Eg:

a = b+c	a = b+c
d = k*f	d = b*c

are concurrent

a=b+c	a=b+c
d=a*k	a=k*m

are not concurrent

Bernsteins Concurrency conditions

For every stmt S we have two sets.

S → Read_set : R(S) : set of variables whose values are read
S → write_set : W(S) : set of variables whose values are updated.

Eg :

S: a=b+c	S: a+=b+c;
R(S) = {b, c}	R(S) = {b, c, a}
W(S) = {a}	W(S) = {a}

If we have two stmts S_i & S_j then

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

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

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

Above 3 conditions must hold for any two stmts for them to execute concurrent.

Mechanisms

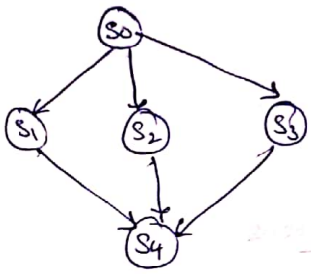
a) Parbegin - Parend / Cobegin - Coend :

Par : parallel Co : Concurrent

```
begin
    S1;
    S2;
    S3;
end
```

⇒

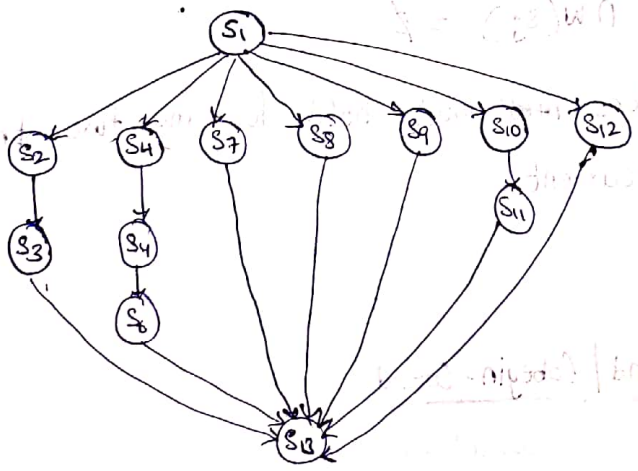
```
S0;
Parbegin (cobegin)
    S1;
    S2;
    S3;
Parend (coend)
    S4
```



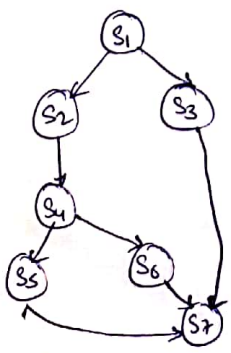
Eg:

```
S1;
Parbegin
    begin  S2; S3;  end;
    begin  S4 ; S5;  S6;  end;
    S7;
    S8;
    S9;
    begin  S10;  S11;  end
    S12;
Parend
    S13;
```

Precedence graph:

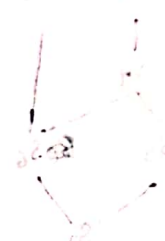


Eg: Implement below graph using parbegin & parend.

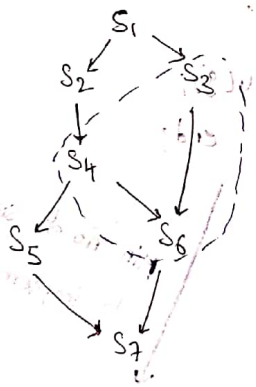


(Q19)

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



(Q20) Implement below graph using parbegin & parend



Sol:

→ This graph is not implementable with ~~parend~~ parbegin & parend

→ It is because of multilevel dependency of the graph

i.e., S_6 depends on S_4 & S_3 which at different levels.

```
S1;
parbegin
    begin S2; S4; end
    S3;
parend
par begin
    S5;
    S6;
par end
 S7;
end
```

The above code is wrong implementation cuz it say S5 ~~also~~ depends on S3 cuz S5 & S3 can run in parallel

Concurrency mechanisms with semaphores

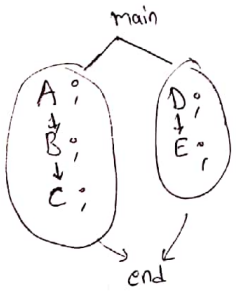
We have seen that this graph is not implementable using parbegin & parend in (H20)

So we use semaphores to implement it.

```
BSEM a,b,c,d,e,f,g = {0};
Parbegin
    begin  S1; V(a); V(b); end
    begin  P(a); S2; S4; V(c); V(d); end
    begin  P(b); S3; V(e); end
    begin  P(c); S5; V(f);
    begin  P(d); P(e); S6; V(g);
    begin  P(f); P(g); S7; end;
Parend
```

Min no of semaphore req to implement above graph = 3

(H2/12)



1. A B C D E ✓
2. D E A B C ✓
3. A D B E C ✓
4. A E B D C
   E → D
   ∴ not possible
5. D C E B A
   C → B
   ∴ not possible

```
BSEM a,b,c,d,e,=0;
S1;
Parbegin
    begin S2; S4; V(a); V(b); end;
    begin S3; V(c); end;
    begin P(a); S5; V(f); end;
    begin P(b); P(c); S6; V(g); end;
Parend
S7;
```

H2/13

begin ① x=1; ② y=y+x; end

begin ③ y=2; ④ x=x+3; end

1 2 3 4
3 4 1 2
1 3 4 2
3 1 2 4
1 3 2 4
3 1 4 2

I) x=1 y=2

x=1 is possible only if order is

x=x+3
↓
x=1

∴ y=2
x=x+3
x=1
y=y+x

∴ x=1 y=3

II) x=1 y=3

∴ II is true

III) for y to be 6

③ y=2;
① x=1; (or) 1342 also valid
④ x=x+3;
② y=y+x;

x=4; y=6

∴ III is possible

H2/14

For the same question if no semaphore are used what could be the possible o/p?

I. x=x+1 } (21)
x=y+1 }

II. x=y+1 } (22)
x=x+1 }

III. Load R, x
Inc R R 1
x=y+1 x 21
Store x, R x 1 } (1)

x=x+1

Load R, x
Inc R
Store x, R

```
Parbegin
  C1 [ begin
         P(mx);
         x=x+1;
         V(mx);
       end
  C2 [ begin
         P(mx);
         x= y+1;
         V(mx);
       end
Parend
```

Here one C_1 is in the middle of its execution and get preEmpted, C_2 must wait on mx.

However we can't gaurantee whether C_1 executes first or C_2 executes first.

∴ possible o/p: 21, 22

C1,C2 C2,C1

~~Par 8~~

Consider below program

```
Parbegin
    begin
        P(mx);
        x=x+1;
        V(mx);
    end
    begin
        P(my);
        x=y+1;
        V(my);
    end
Parend
```

Now

possible o/p: 1, 21, 22

Consider below program

```
BSEM mx=0; my=1;
Parbegin
    begin
        P(mx);
        x=x+1;
        V(my);
    end
    begin
        P(my);
        x=y+1;
        V(mx);
    end
Parend
```

Possible value of x: 22

(12/15)

1. C = B-1 ;
2. B = 2*C ;
3. D = 2*B;
4. B = D-1;

What can be possible no of ~~value for~~ distinct values of B?

~~No matter in which~~

1 - 2 - 3 - 4 ⇒ C=1, B=2, D=4, (B=3)

1 - 3 - 2 - 4 ⇒ C=1, D=4, B=2, (B=3)

3 - 4 - 1 - 2 ⇒ D=4, B=3, C=2, (B=4)

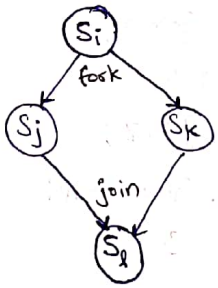
3 - 1 - 4 - 2 ⇒ D=4, C=1, B=3, (B=2)

∴ 3 distinct value

b) fork - Join Concurrency Construct:

Syntax & usage of fork:

```
Si;
fork L;
Sj;
.
.
.
L: Sk;
```



join:

```
Join count;
Sl;
```

(count contains no of ~~station~~ concurrent flows that are being joined. Here count=2)

→ join (int count)

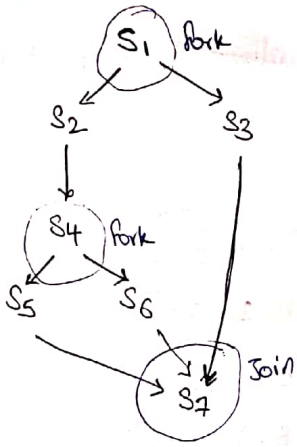
```
{
    count = count - 1;
    if(count != 0) // only when count=0 we execute Sl
        exit;
}
```

when count=0, the stmt next to the join is executed i.e., S_l

→ Now the resultant program is

```
count = 2
Si;
fork L;
Sj;
goto X;
L: Sk;
X: Join count;
Sl;
```

(Q21) Write a program using fork & join to implement below precedence graph.



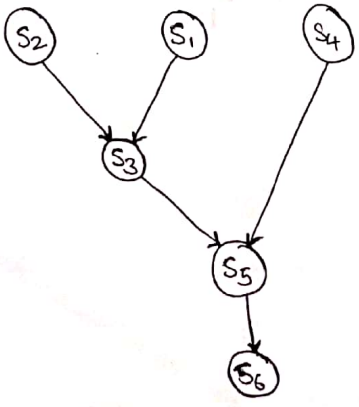

```
S1;
fork L1;
S2;
S4;
fork L2;
S5;
goto X;
L1: S6;
L2: join #2
X:
```

```
Count = 3
S1;
fork L;
S2;
S4;
fork X;
S5;
goto Z;
L: S3;
goto Z;
X: S6;
goto Z;
Z: Join count
S7;
```

→ Any graph is implementable with fork & join., but not with parbegin & parend.

However any graph is implementable if we use semaphores along with parbegin & parend.

(H2/17)



Join to S3;

∴ S3 must be joined by 2 stmts

Silly S5 must be joined by 2 stmts

H2/11

V(mutex)

V(mutex)

R != 0 || w == 1

~~Find minimum possible value of~~ count.

H2/16 test() ①

Count ~~++~~ = Count + 1

Count = Count + 1

Count = Count + 1

Count = Count + 1

Load R, Count

Inc R

test() ② ~~Store Count, R~~ Count 4

Count = Count + 1

Count = Count + 1

Count = Count + 1

Count = Count + 1

Count = Count + 1

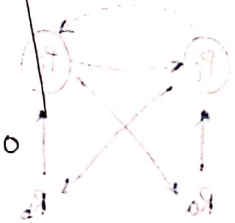
Count 9

test() ① Store Count, R Count 5

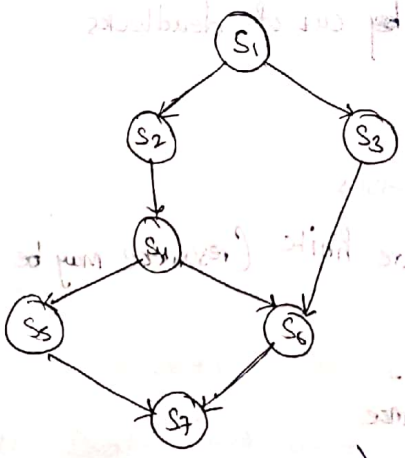
∴ min value = 5

Also here

~~max value possible~~ = 10



Q22 write a program using fork & join to implement below precedence graph.



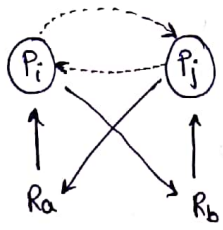
Sol:

```
c1 = 2;
c2 = 2;
S1;
fork L1;
S2;
S4;
fork L2;
S5;
goto X;
L1: S3
    goto L2;
L2: Join c1
    S6;
    goto X;
X: Join c2
   S7;
```

Deadlocks:

Two/more processes are said to be in deadlock iff they wait for happening of an event which would never happen.

Eg:



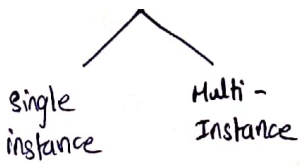
→ Deadlock decreases resource utilization drastically.

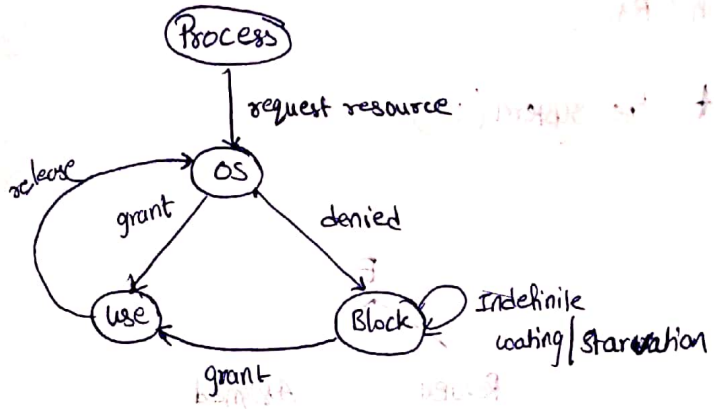
→ Also throughput is decreased by cuz of deadlocks.

System Model

→ we assume system has n-processes

→ we assume no of resources are finite (resource may be Hlw or Slw)





If the process stays in block state indefinitly then it is starvation.

" " " " " " " infinitely then it is deadlock.

*→ Every deadlock is a starvation.

Necessary Conditions for Deadlock

a) Mutual Exclusion:

i.e., presence shared resources that can be used by only one process at a time.

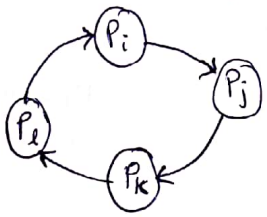
b) Hold & Wait:

There must be processes holding a resource and waiting for some other resource.

c) No Preemption of resources from processes:

i.e., every process releases resources voluntarily

d) Circular wait:



The above four condition are necessary but not sufficient.

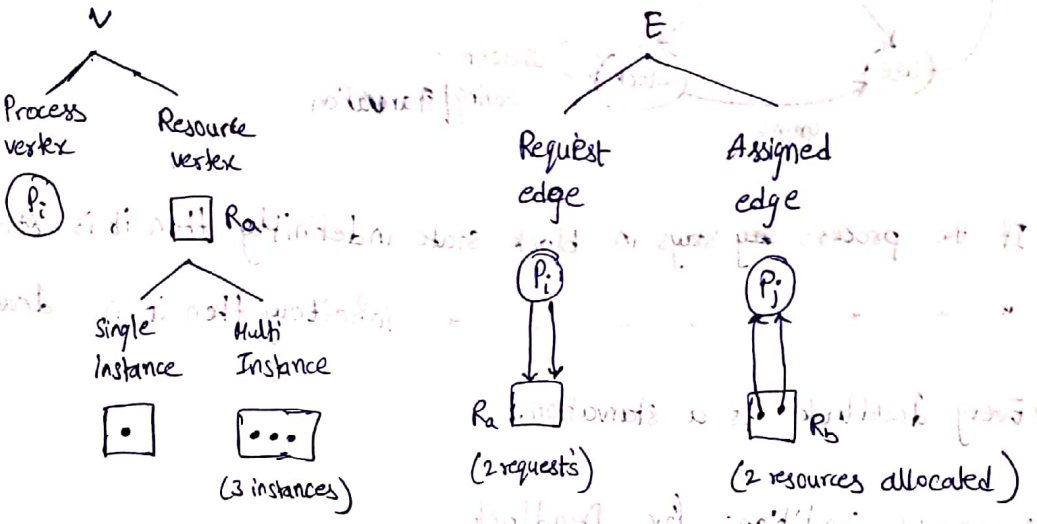
Thus ~~even~~ even if we have all the four conditions ~~dead~~ deadlock may not happen. However if there is a deadlock every condition is satisfied.

i.e., Deadlock $\Rightarrow (a \wedge b \wedge c \wedge d)$

Resource Allocation Graph (RAG)

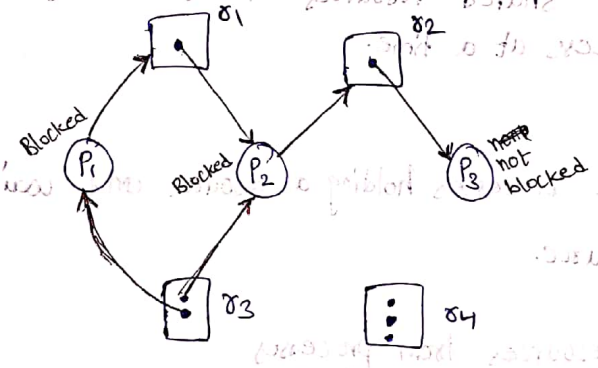
* Mechanism to represent the system (os)

$G = (V, E)$

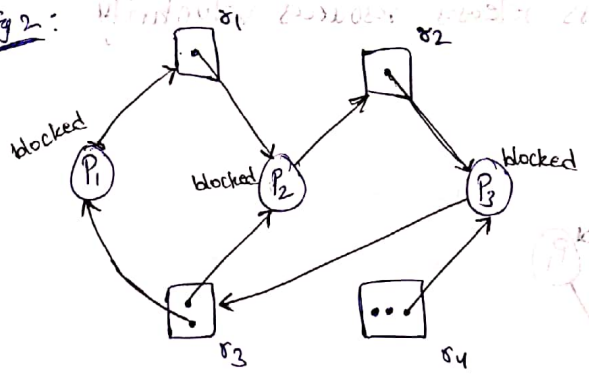


* RAG is a multigraph.

Eg 1:

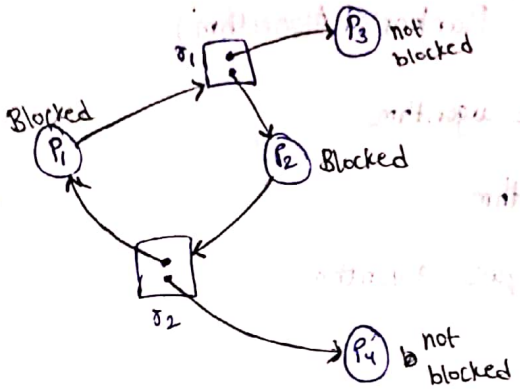


Eg 2:



Cycle: $P_1 - r_1 - P_2 - r_2 - P_3 - r_3 - P_1 \Rightarrow$ Deadlock

Eg3:



cycle: $P_1 - r_1 - P_2 - r_2 - P_1$

* Even though we have a cycle, we don't have a deadlock.

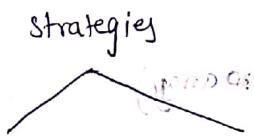
From the above examples we say

** Every deadlock has a cycle, but every cycle need not to be a deadlock.

** However, if all the resource instances are single instance type, then cycle is both necessary & sufficient condition for deadlock.

Single instance resources $\longrightarrow$ (cycle $\longleftrightarrow$ Deadlock)

Deadlock handling Strategies



Proactive	Reactive
(Deadlocks are not allowed to occur)	(Deadlocks are let to occur and handle after the occurance)
Eg: Prevention, Avoidance	Eg: Detection & Recovery, Ignorance.

1. Ignorance:

→ This is also known as ostrich algorithm.

→ No strategy is used, we just ignore deadlocks.

2.) Deadlock Avoidance (Banker's Algorithm)

Here we use two sub-algorithms

(i) Safety Algorithm

(ii) Resource-Request Algorithm

System_state:

Not all unsafe states are Deadlocks.
It is just that an unsafe state may lead to deadlock

→ The basic objective of Banker's Algo is to operate the system in safe state.

→ Safety algo is used to check whether the system is in the safe state or not.

Data structures (for Banker's Algo)

→ n: processes ($P_1 \dots P_n$)

→ m: Resources ($R_1 \dots R_m$)

→ Maximum $[1 \dots n, 1 \dots m]_{n \times m}$ (2D-array)

- $Max[i,j] = k$ means that

  P_i is requesting a maximum of k copies of resource j throughout its lifetime.

  i.e., Apriori Demand $\quad P_i \xrightarrow[req]{} k(R_j)$

  Every process has to tell how many instances of each resource it would need

→ Allocation $[1 \dots n, 1 \dots m]_{n \times m}$

$Alloc[i,j] = a$

$P_i \xleftarrow[alloc]{} a(R_j) \Rightarrow (a \le k) \quad (Alloc[i,j] \le Max[i,j])$

→ Need $[1 \cdots n, 1 \cdots m]_{n \times m}$

Need[i,j] = b

$P_i \xrightarrow[need]{} b(R_j)$

* **Need[i,j] = Max[i,j] - Alloc[i,j]**

→ Request $[1 \cdots n, 1 \cdots m]_{n \times m}$

Req[i,j] = c

$P_i \xrightarrow[req]{} c(R_j)$ at some time 't'

* **Req[i,j] ≤ Need[i,j]**

Once req is ~~a~~ processes, allocation & need are modified accordingly

→ Total $[1 \cdots m]$

total[j] = 'z';

i.e., R_j has ~~2~~ a total of 'z' instances.

→ Available $[1 \cdots m]$

avail[j] = y;

i.e., there are 'y' copies of R_j available at a given time.

* avail[j] ≤ total[j]

Safety Algorithm

Say no of process, n=5; $\langle P_1, P_2, \cdots P_5 \rangle$

m=1 ,

R=21

	Max / R	Alloc / R	Need / R	Avail / R
P_1	10	5	5	
P_2	6	3	3	
P_3	3	1	2	
P_4	8	5	3	
P_5	12	6	6	

$$Avail = total - \sum_{i=1}^{n} Alloc_i \quad @t$$

$$Avail = 21 - 20 = 1$$

→ System is said to be in safe state iff need of resources can be satisfied with available resources in some order.

So, the above state of the system is unsafe.

Now for the same above data set assume R = 22.

$$Avail = 22 - 20 = 2$$

Now we can satisfy the need of P_3

Once P_3 finishes its execution

$$avail = 1 + 2 = 3$$

P_4 : $avail = 3 + 5 = 8$

P_2 : $avail = 8 + 3 = 11$

P_1 : $avail = 11 + 5 = 16$

P_5 : $avail = 16 + 6 = 22$

$\langle P_3, P_4, P_2, P_1, P_5 \rangle$ this one of the possible order of execution.

Hence we can say the system is in safe state

P_3, P_4, P_2, P_1, P_5 is called safe sequence.

Q23

$n = 5$ $\langle P_0 \ldots P_4 \rangle$

$m = 3$ $\langle A, B, C \rangle = \langle 10, 5, 7 \rangle$ (total)

t_0 :

	Max			Alloc			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1
total alloc :				7	2	5			

Find whether the system is the safe state or not.

Sol :

total = $\langle 10, 5, 7 \rangle$

Avail = $\langle 3, 3, 2 \rangle$

P_4 : ~~avail = $\langle 3, 3, 2 \rangle$~~ P_3 : ~~avail =~~ P_3 : ~~avail =~~ $\langle 5, 4, 3 \rangle$ P_1 : avail = $\langle$

P_3 : ~~avail =~~ $\langle 5, 4, 3 \rangle$

P_1 : avail = $\langle 7, 4, 3 \rangle$

P_0 : avail = $\langle 7, 5, 3 \rangle$

P_2 : avail = $\langle 10, 5, 5 \rangle$

P_4 : avail = $\langle 10, 5, 7 \rangle$

- P_3, P_1, P_0, P_2, P_4 is a safe sequence and hence the system is in the safe state.

Now at time 't0' the system is in the safe state.

Say at time 't_i' P_1 is schedule and now let us say it requests $\langle 1,0,2 \rangle$

$t_1 : (P_1) : \longrightarrow \langle 1,0,2 \rangle : Req_1$

Resource Request Algorithm

(grant the req_i iff the resulting ~~state~~ state is safe)

Algo Res_req (P_i, Req_i, $Alloc_i$, $Need_i$, Avail)

{

1. $Req_i \leq Need_i$
2. $Req_i \leq Avail$;
3. (Assume we satisfied req_i)
   a) $Avail = Avail - req_i$
   b) $need_i = need_i - req_i$
   c) $alloc_i = alloc_i + req_i$

   } System state is changed
4. Run safety algorithm
5. If system is safe then grant req_i

   else deny req_i & block the process P_i

}

Assume the system state is

	Max			Alloc			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	3	0	2	0	2	0
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

avail : $\langle 2,3,0 \rangle$

P_1 : avail : $\langle 5,3,2 \rangle$

P_3 : avail : $\langle 7,4,3 \rangle$

P_0 : avail : $\langle 7,5,3 \rangle$

P_2 : avail : $\langle 10,5,5 \rangle$

P_4 : $\langle 10,5,7 \rangle$

The state of the system is safe

Hence the request $\langle 1,0,2 \rangle$ to P_1 can be granted.

Assume at time 't_3' P_4 has requested $\langle 3,3,0 \rangle$

t_3: P_4 : $\longrightarrow \langle 3,3,0 \rangle$

avail : $\langle 2,3,0 \rangle$

req_4 : $\langle 3,3,0 \rangle$

avail < req

∴ not granted

Now P_4 is blocked

Assume at time 't_5' P_0 requests $\langle 0,2,0 \rangle$

t_5: P_0 $\longrightarrow \langle 0,2,0 \rangle$

The system would be as shown below the req was granted

	Max			Alloc			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	3	0	7	2	3
P_1	3	2	2	3	0	2	0	2	0
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

avail : $\langle 2,1,0 \rangle$

From this avail no processes' need can be satisfied.

i.e., unsafe state.

∴ the request can't be granted

So P_0 is blocked

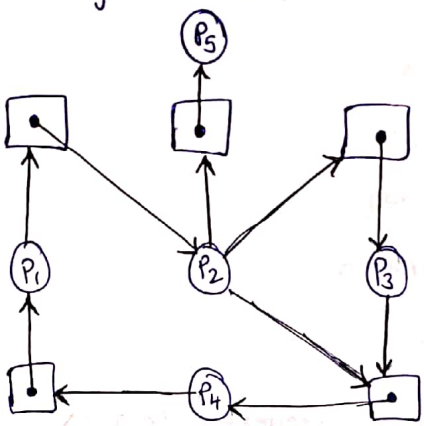
Also we go to the previous state.

3. Deadlock Detection & Recovery

a) Deadlock Detection algorithms

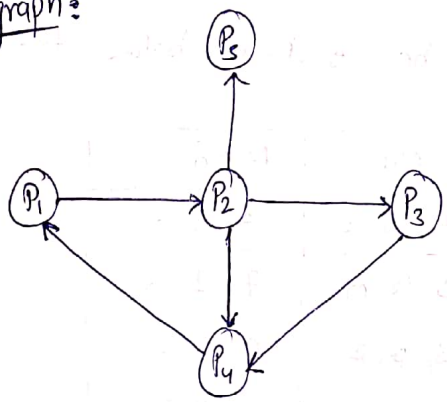
(i) Detection with single resource instances.

Consider system as shown below, at time t_3



Now we derive wait for graph from RAG

wait-for graph:



Now, we run a cycle detection algorithm on wait-for graph

Here $\langle P_1 - P_2 - P_3 - P_4 - P_1 \rangle$ is a cycle.

Since this is a single instance resource type,

Cycle $\Leftrightarrow$ deadlock.

$\therefore$ Deadlock is formed.

(ii) Deadlock Detection with multi-instance resources

$n=5$; $\langle P_0 \cdots P_4 \rangle$; $m=3$; $\langle A,B,C \rangle = \langle 7,2,6 \rangle$

At time t_0:

	Alloc			Req		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

Now, Avail : $\langle 0,0,0 \rangle$

Here at time t_0:

P_0, P_2 are unblocked

P_1, P_3, P_4 are blocked

System is said to have no deadlock (safestate) iff all processes are satisfied with available resources

to ~~P_0~~, P_0 : Avail : $\langle 0,1,0 \rangle$

P_2 : avail : $\langle 3,1,3 \rangle$

P_3 : avail : $\langle 5,2,4 \rangle$

P_4 : avail : $\langle 5,2,6 \rangle$

P_1 : avail : $\langle 7,2,6 \rangle$

∴ The state is safe

Hence no deadlock.

→ Assume at time t_1: P_2 made a request

$P_2 \longrightarrow \langle 0,0,1 \rangle$

	Alloc	Req
	A B C	A B C
P_0	0 1 0	0 0 0
P_1	2 0 0	2 0 2
P_2	3 0 3	0 0 1
P_3	2 1 1	1 0 0
P_4	0 0 2	0 0 2

avail : <0,0,0>

P_0 : avail : <0,1,0>

No other reqs can be satisfied

∴ P_1, P_2, P_3, P_4 are in deadlock.

∴ P_1, P_2, P_3, P_4 must be recovered from deadlock.

Recovery strategies:

- Process termination (Abortion)
  - kill_all
    - → All killed processes are restarted
    - → No need to apply detection algorithm again (advantage)
    - → Important processes may also get killed (dis-advantage) i.e., processes whose execution is abt to end
  - kill_one @time
    - → we kill processes one by one until deadlock is gone
    - → After killing every process (dis-advantage) deadlock detection has to be done.
    - → May be important processes may gets spared (advantage)
- Resource PreEmption

Resource PreEmption:

→ we take away latest held resources from ~~∞~~ ~~proce~~ processes and check if the deadlock is gone.

In this way we preEmpt resources until the deadlock is gone.

→ One advantage here is we don't need to restart the process.

→ Disadvantages:

* Detection algorithm has to be run again & again.
* Also when we preEmpt a resource we must rollback the execution of process to the time before it held the resource. This is an overhead.

*

* Both process termination & resource preEmption may suffer from starvation.

In process termination starvation may occur because ~~therse~~ there is a chance that killed processes may again go to deadlock.

~~Simitta~~ Similarly process preEmption also may suffer from starvation because preEmpted processes may again go to deadlock.

H2/16

test() 1:
① Load R, count
Inc R R[1]
count [0]

test() 2:
① count [1]
② count [2]
③ count [3]
④ count [4]

test() 1
Store count, R
count [1]

test() 2:
⑤ Load R, count
Inc R R[2]

test() 1:
② count [2]
③ count [3]
④ count [4]
⑤ count [5]

test() 2:
~~count~~ [5]
Store count, R count [2]

∴ min value = 2

H3/1

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

avail : $\langle 3,2,2 \rangle$

Req 1 :

$\langle 0,0,2 \rangle$

New avail : $\langle 3,2,0 \rangle$

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

avail : $\langle 3,2,0 \rangle$

P_1 : avail : $\langle 6,4,0 \rangle$

Neither P_0 nor P_2 can be schedule

i.e., unsafe state

∴ req1 is not granted

Req 2 :

	alloc			Max			Need		
	X	Y	Z	X	Y	Z	X	Y	Z
P_0	0	0	1	8	4	3	8	4	2
P_1	5	2	0	6	2	0	1	0	0
P_2	2	1	1	3	3	3	1	2	2

$P_1 \xrightarrow{req} \langle 2,0,0 \rangle$

avail : $\langle 1,2,2 \rangle$

P_1 : avail : $\langle 6,4,2 \rangle$

P_2 : avail : $\langle 8,5,3 \rangle$

P_0 : avail : $\langle 8,5,4 \rangle$

i.e., Safe State ∴ Req2 is granted.

H3/3

	Alloc			req		
	r_1	r_2	r_3	r_1	r_2	r_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

avail : <0,0,1> (find no of outgoing edges from resources and substract it from no of instances)

P_2 : avail : <0,1,1>

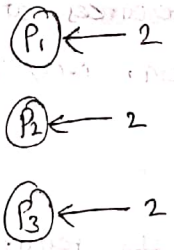
P_0 : avail : <1,1,2>

P_1 : avail : <2,2,2>

P_3 : avail : <2,3,2>

∴ no deadlock

H3/4 Here first allocate one less than no of resources required



Now adding one more tape unit would get the processes free from dead lock

∴ min required = 6+1 = 7

H3/5

	alloc	Max	Need
P_1	3	7	4
P_2	1	6	5
P_3	3	5	2

total = 9

avail = 2

P_3 : avail : 5

P_2 : avail : 6

P_1 : avail : 9

~~now req of P_1 cannot be satisfied~~

∴ Safe, not deadlocked

4. Deadlock Prevention:

Deadlock is prevented by dissatisfying one/more of the four necessary conditions.

a) Mutual Exclusion:

avoiding mutual exclusion means we should not use shared resource. i.e., no interprocess communication.
Also it is impossible to have multiprogramming as without a shared resource.

∴ ME is not avaidable.

b) Hold and wait:

Here we make sure that process either holds or waits. but not both.

→ ~~To implement this~~

(i) Process must request and be allocated all the resources prior to its commencement. Here we hold and don't wait.

disadvantage:

→ Hence this protocol may suffer starvation.

→ Also it takes all the resources at the start itself and may not use them. i.e., inefficient use of resources.

(ii) Process must release all the resources before making a fresh request

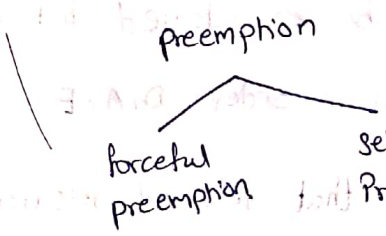
i.e., we don't hold and we wait.

drawback:

→ once process release a resource, it is not guaranteed it gets them back immediately. So here also we have starvation

c) No-PreEmption :

Here we allow preEmption of resources.



forceful preEmption

→ Here the basic idea is the running process shouldn't get blocked. So running process forcfully takes resources from other processes which are in waiting state (but not ready state)

→ If the resource are not available and not held by waiting processes

Self PreEmption then the current process must wait.

→ Here running process voluntarily releases resources and lets other processes to execute.

* However this approach may also suffer from starvation.

d) Circular wait :

(i) Number all resources uniquely

(ii) Never allow a process to req a lower numbered resource than the id of resources that are currently being held.

Eg : Resources : A B C D E F
(8) (15) (3) (5) (10) (12)

Assum Pr_i is holding C, A, E
(3) (8) (10)

Now the process can request B or F
(15) (12)

but the process cannot request D
(5)

→ Now if the processes wants resource D, then it must release the conflicting resources

i.e., A & E should be released but not C

Now process request again in order D, A, E

→ However there is no guarantee that released resources will be allocated immediately.

∴ This strategy also suffers from starvation.

(Q24) Consider a system having n-processes and single resource 'R' having 6 instances. Each process needs 2 instances to complete its execution.

(i) what is the minimum value of n to cause deadlock

(ii) what is the maximum value of n for deadlock free operation.

Sol:

(i) If we have 6 processes all the processes can get one resource each.

∴ 6

(ii) If we have 5 resources,

even if every process takes one resource we will be left with a resource and thus we don't have deadlock

∴ 5

(Q25) n=3 processes

every process need 2 instances of resource R

(i) what is max no of R for deadlock

(ii) what is min no of R for deadlock free operation

Sol:

(i) P_1 P_2 P_3

(1) (1) (1) ⟹ 3 (deadlock)

$\therefore 3$

(ii) P_1 P_2 P_3 } Add 1 resource
① ① ①

$\therefore 4$

(Q26) n: processes

R: 6 units

P_i needs 3 units of R

(i) Min(n) for deadlock

(ii) Max(n) for deadlock free

Sol:

$P_1 : 3$	$P_1 : 3$	$P_1 : 2$
	$P_2 : 3$	$P_2 : 2$
	6	$P_3 : 2$
		6

$\therefore$ deadlock

$\therefore$ (i) 3

(ii) 2

(Q27) n=5 processes

Resource R

Peak(max) demand of each process for resource R is as given below

P.No	Max(R)
1	10
2	3
3	20
4	15
5	12

(i) Max(R) for dead lock

(ii) Min(R) for dead lock free operation.

(i)

①	②	③	④	⑤	
↓	↓	↓	↓	↓	
9	2	19	14	11	⇒ Deadlock

55

(ii) ∴ 56 will be sufficient for deadlock free operation.

H3/2 n-process

Pid	Alloc	Req
P_1	x_1	y_1
P_2	x_2	y_2
P_3	x_3	y_3
⋮	⋮	⋮
P_A	x_A	0
P_B	x_B	0
⋮	⋮	⋮
P_n	x_n	y_n

a) Right now P_A & P_B can be computed

∴ avail = $k + x_A + x_B$

∴ to not approach deadlock

$\min(y_i) \leq (k + x_A + x_B)$ $_{(i \neq A \neq B)}$ (not approaching deadlock, but deadlock make occur in future)

b) total(R) = $k + \sum_{i=1}^{n} x_i$

c) if $(k + x_A + x_B) \geq \max(y_i)$

then deadlock free

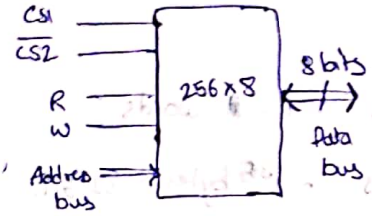
15/08/20

Memory Management:

(Primary memory or Physical memory or RAM)

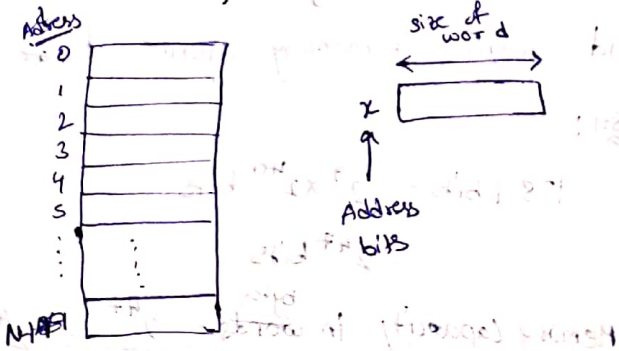
→ There are two views of memory

i) Physical view:



ii) Abstract view (programmer's view):

viewed as 1d array of words



Memory[0...N] (N words)

If size of word is 'm' bits and no of bits in address is 'n'

then Memory Capacity (total words) $= 2^n = N$

→ If word $N = 8$ words

$\Rightarrow n = 3$ bits

i.e., $N = 2^n$ words

$n = \log_2 N$ bits

$\Rightarrow N = 16$ GB

$\Rightarrow$ i.e., $N = 16 \times 2^{30} = 2^{34}$

$n = 34$

→ If n=28 bit

a max of 2^{28} Bytes can be addressed

i.e., 256 MB

→ If n=34 bits and word length = 16 bits

find capacity of memory interms of words, bytes, bits.

Sol:

$2^n = 2^{34}$ words = 16G words

$2^{34} \times 2$ bytes $= 2^{35}$ bytes = 32 GB

$2^{34} \times 16$ bits $= 2^{38}$ bits = 256 G.bits

→ let capacity of memory be 128 Tbits and word size be 64 bits

find capacity of memory interms of bytes and words

Sol:

128 Tbits = $2^7 \times 2^{40}$ bits

$= 2^{47}$ bits

Memory capacity in bytes = $\frac{2^{47}}{8} = 2^{44}$ bytes

= 16TB

Memory capacity in words = $\frac{2^{47}}{64} = 2^{41}$ words

= 2 T. words.

Loading vs Linking

↳ Loading of executable program from disk to memory.

Loading:

Loading is of two types:

i) Static Loading:

* Entire program is load before execution

(ii) Dynamic Loading:

Modules can be loaded during runtime.

Eg: Consider

```
main()
{
    if(...)
        f1();
    else
        f2();
}
```

In static loading we load main(), f1(), f2()' into memory. However we use only one out of f1 & f2. This is space inefficient.

Thus we can use dynamic loading and load only main() initially and load either f1 or f2 based on the requirement. at the time of execution.

→ However execution time is more if we use dynamic loading.

→ Static loading: Space inefficient, time efficient

Dynamic loading: Space efficient, time inefficient

Linking:

Linking is the process of resolving external references

↳ function, variable (global) or any objects.

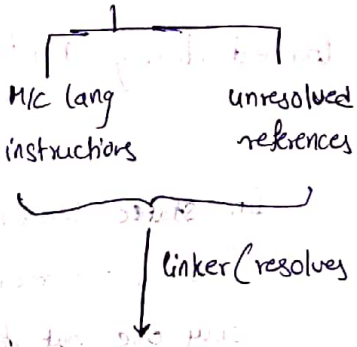
kk.c

```
extern int x; // external reference
main()
{
    :
    f() // since f() is defined after
    :   main() the address of f()
}       is not known earlier. This
f()     is an unresolved reference.
{
    :
    g() // BSA ——→ unresolved reference
    :
}
g()
{
    :
    h() // external reference
    :
}
```

ak.c

```
int x;
h()
{
    :
    scanf()
    :
}
```

Compiler generates objects code



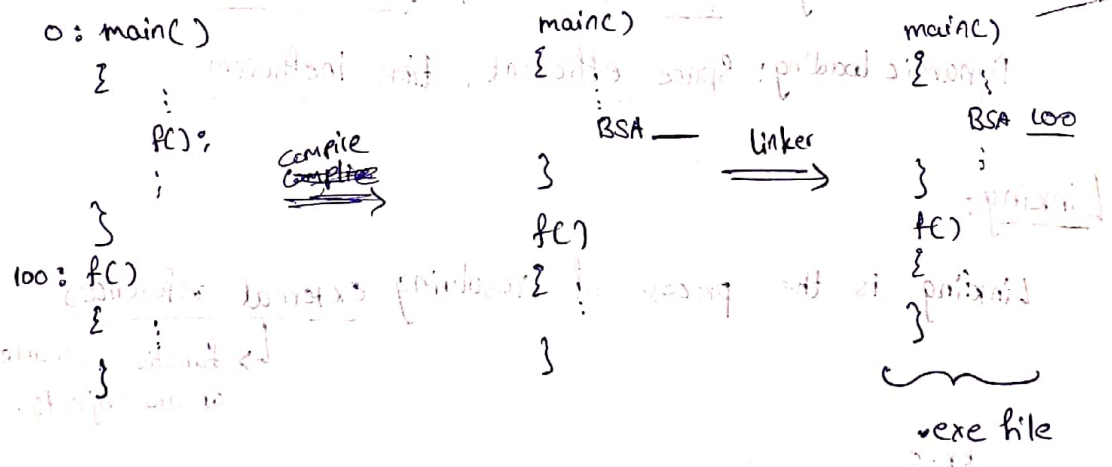
The linker resolves addresses through logical addresses

Linking is of two types

(i) Static Linking
(ii) Dynamic Linking

(i) Static Linking:

* Linking is done before runtime.



Drawback:

* Sometimes unnecessary modules may be linked and this is inefficient. for which we sometime may postpone linking till runtime (i.e., late binding or dynamic linking)

* However execution of program is faster in the case of static linking.

(ii) Dynamic Linking:

For every unresolved reference compiles associates a piece of code called Stub. This stub when execute at runtime calls linker.

→ The libraries which are linked at runtime are called
· Dynamic Link Libraries (DLL)

→ It is more space efficient.

Benefits of dynamic linking

* space efficiency
* Reusability of libraries
* flexibility of modification (debugging becomes easy)

Drawback

* time inefficient
* Security threat

→ Static linking is more secure.

Modules needed by linker:

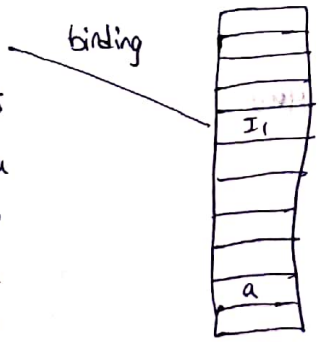
→ object code
→ external references
→ Relocation information

* Linker does not need physical address of memory.

Address Binding

Association of program instructions and data units to memory locations is called address binding.

```
I1 : Load  a,#1
I2 : Load  b,#5
I3 : Load  R1,a
I4 : Load  R2,b
I5 : Add   R1,R2
I6 : Store c,R1
```



The time at which binding takes place is called binding time.

Binding can take place at three times:

- (i) Compile time } Static
- (ii) Load time }
- (iii) Run time } Dynamic

Compile time Binding:

* Compiler associates physical addresses.
* The program is loaded into fixed memory locations.

Load time Binding:

* Here loader does the work of binding.
* Based on the value in base register, the instruction are loaded into memory.

Runtime Binding (Dynamic Relocation)

* Here loader does the work of binding at runtime.

→ If a process is swapped out

* it must be swapped in into the same memory location if we use compiletime binding or load time binding.
* it can be swapped in into different memory location if we use runtime binding.

Functions & Goals of Memory Manager:

Functions:

i) Allocation

ii) Protection

iii) Address Translation (done by MMU).

iv) Free Space Management

v) Deallocation

Goals:

→ Effective utilization of memory

i.e., Minimize Fragmentation

→ Manage the execution of larger program in a small memory area.

→ Memory management Techniques

Contiguous

i.e., program is centralized

i.e., program is present at one place

Eg: Partitions < fixed / dynamic

Overlays

Swapping

Non-Contiguous

→ Here program may be divided into modules and different modules may be at different places

Eg: Paging

~~Semenat~~

Segmentation

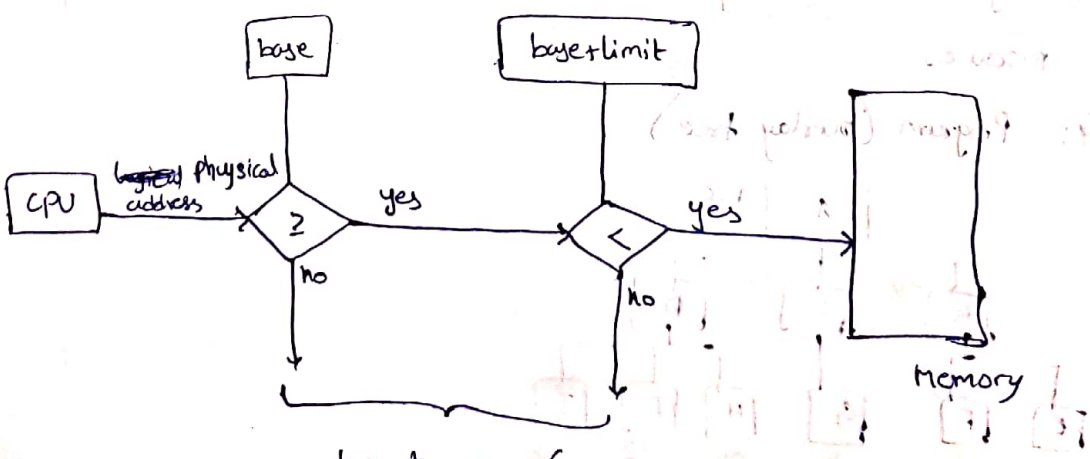
Segmented paging

~~Vir~~ Virtual Memory

Protection:

→ (Relocation register)

* base register contains starting address of the process.
* limit register contains size of process.



1. Contiguous Memory Management Techniques:

a) Overlays:

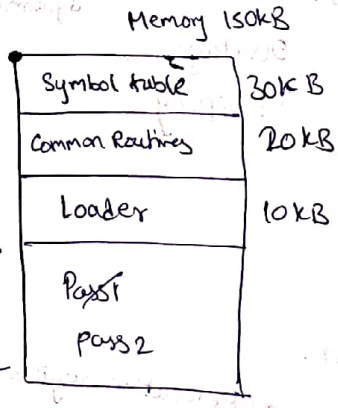
Assume we have a 2-pass Assembler.

Below are sizes of each module.

pass1 : 70 kB
pass 2 : 80 kB
Symbol table : 30 kB
Common Routines : 20 kB
Overlay loader (or) overlay driver : 10 kB

} total memory : 210 kB

Assume we have only 150kB of free memory



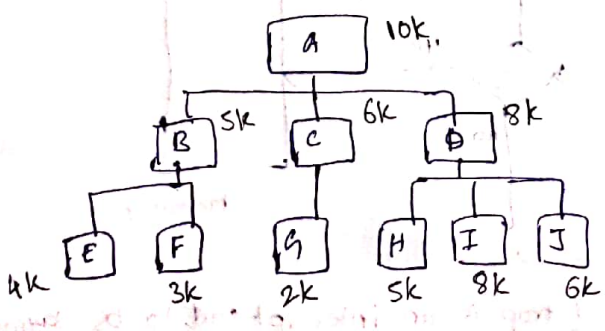
→ Pass1 and Pass2 are independent modules. i.e., pass1 can run without any use of pass2 and vice versa

→ Now 1st overlay loader loads pass1. Later pass1 is overlayed (replaced) by pass2

Limitation:

→ Program must be possible to be divided into independent modules

Eg: Program (overlay tree)



For the above overlay tree, what is the minimum amount of memory required for executing? ~~the above?~~

Sol:

During pass1 we may need A,B,E (i.e., 19KB)

During pass2 we may need A,B,F (i.e., 18k)

In that way at some pass we may need A,D,I

i.e., 10+8+8 = 26k

∴ Size of max pass = 26k

∴ Min memory required = 26k

* → Minimum memory required = Max{path lengths from root to the leaves}

b) Partitions:

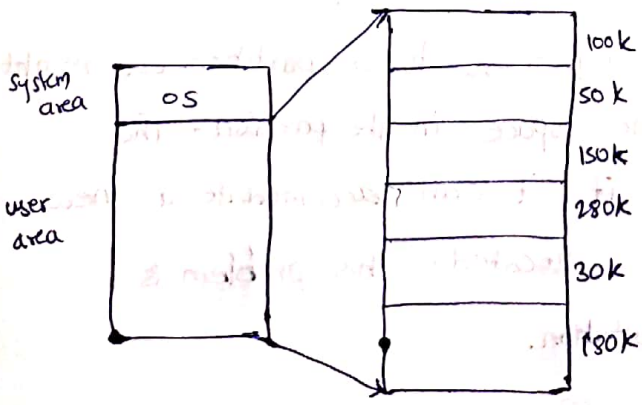
Partitions is of 2 types:

(i) Fixed Partitioning (MFT: Multiprogramming with Fixed no of Tasks)

(ii) Variable Partitioning (MVT: Multiprogramming with Variable no of Tasks)

(i) Fixed Partitions: (or) Static Partitioning

User area: It is divided into fixed no of partitions.



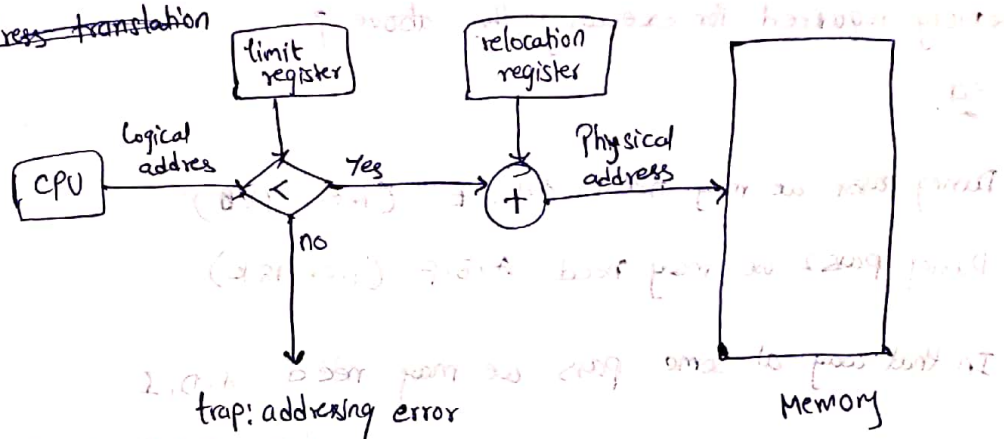
Partitions may be of different sizes.

Each partition holds exactly one process.

Address translation & Protection:

Protection

address translation



Allocation Strategies:

First fit:

→ search from first partition and store it wherever enough sized partition is found

Best fit:

Smallest free and big enough partition is choosen

Worst fit:

Largest free and big enough partition is choosen

Next fit:

* Next fit works like fit except that search for free partition starts from last allocation.

Performance issues:

i) Internal Fragmentation:

After allocating a process to a partition we might be left out with some space in the partion. The left out space even if it can accomodate a new process shouldn't be allocated. This problem is called internal fragmentation.

2) External Fragmentation:

No external fragmentation

3) Degree of Multiprogramming is restricted to no of

4) ~~size of max~~ Maximum process size that is runnable is limited

5) Best fit is the most suitable allocation technique. cuz it has least amount of internal fragmentation.

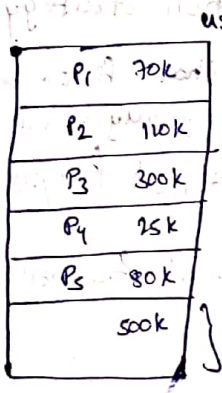
(ii) Variable Partitions (or) Dynamic Partitioning

Partitions are created dynamically at runtime

At time t assume we have below requests

70k ; 120k ; 300k ; 25k ; 80k ;
P_1 P_2 P_3 P_4 P_5

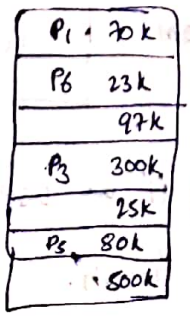
Here OS maintains a table to keep track of which parts of memory are available



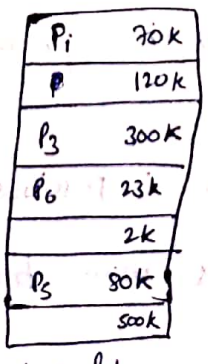
→ If P_5 finishes, its execution is we have two holes of 80k and 500k adjacent to each other. Now these two will be joined. This joining of adjacent holes is called coalescing.

Assume after some time P_2 and P_4 has finished execution and we have a new process request by P_6 of 23k.

Now we may use first fit, best fit, worst fit, next fit :



first fit



best fit

Performance issues

1. No internal fragmentation
2. We may have External fragmentation

Eg: for previous example,

assume we have a process req of 550k, we can't place in the memory. However total available space is more than 550k. Thus we say that we may have external fragmentation.

> we say we have EF iff
> * We can't allocate memory for process even if total available space is more than the size of process

3. Deg. of M.Pr is flexible.
4. Maximum size of runnable process is also flexible.
5. Using best fit creates small free holes. These holes are wasted in the form of external fragmentation.

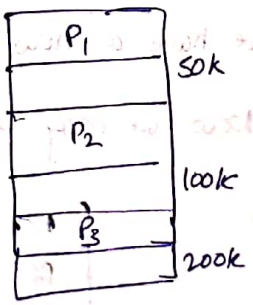
- Hence worst fit is best allocation strategy for dynamic partitioning. (In TB it is given that first & best are better than worst fit: first & best are equally good in storage utilization, however first fit is faster)

Overcoming the problem of External fragmentation:

we have 2 way for overcoming external fragmentation.

i) Compaction (or) Defragmentation:

At time t, assume memory is as shown below



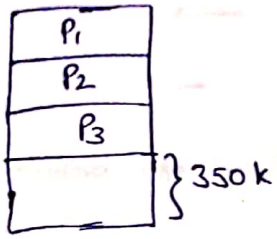
Now if we have a process (P_4) request with size 250k we cannot allocate it.

→ Compaction moves all the processes to one place.

* → However compaction is possible only if we use runtime binding.

→ Also compaction takes more time to finish.

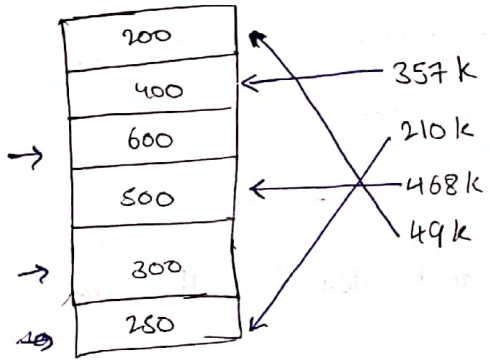
Memory after compaction is



(b) Non-Contiguous Allocation (Paging)

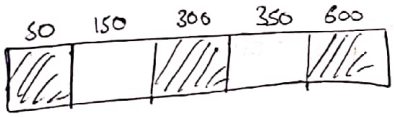
If we don't use contiguous allocation we can now ~~place~~ divide the program and place it in different holes.

H4/1

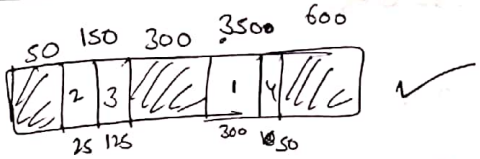


~~∴ 250 & 600~~ ∴ 600k & 300k

H4/2

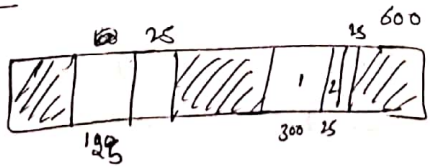


first fit



✓

Best fit



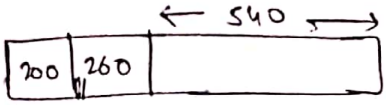
✗

only first fit

∴ opt (b)

H4/3

(i)



To get denied always we provide maximum hole possible

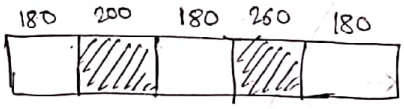
∴ 541k

(ii) To find smallest req to be denied we must create maximum possible holes of equal sizes

max possible holes = 3

total free space = 540

size of each hole = $\frac{540}{3} = 180$



∴ 181k can be denied in this case

H4/4

Memory size = 2^{46} bytes

no of partitions = $\frac{2^{46}}{2^{24}} = 2^{22}$ partitions

$= 4M$ partitions

partition address (Ptr) = 22 bits

(i) size of pointer to the nearest byte

nearest byte to 22 bits is 3 bytes (24 bits)

(ii) size of each entry = size (Ptr) + size (PID)

= 3 + 4 bytes

= 7 bytes

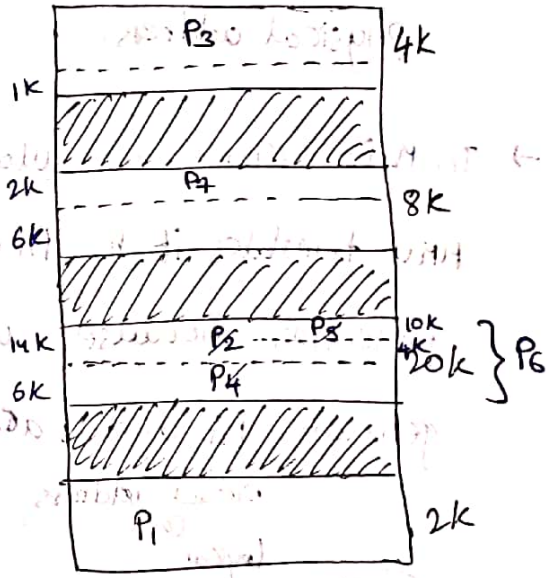
i.e., $7 \times 500 = 3500$ bytes

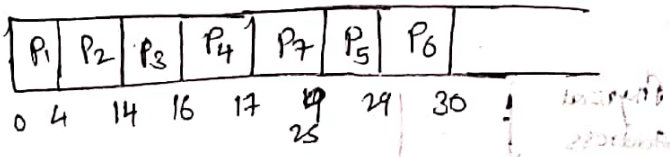
(H5 15) Load time is the time when they are loaded into memory.

The load time of

P_1, P_2, P_3, P_4, P_7 is 0

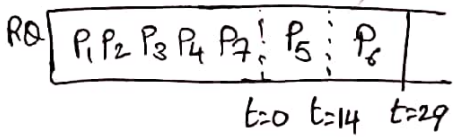
P_5 & P_6 must wait until 20k hole is freed





Load time of $P_5 = 14$

Load time of $P_6 = 29$



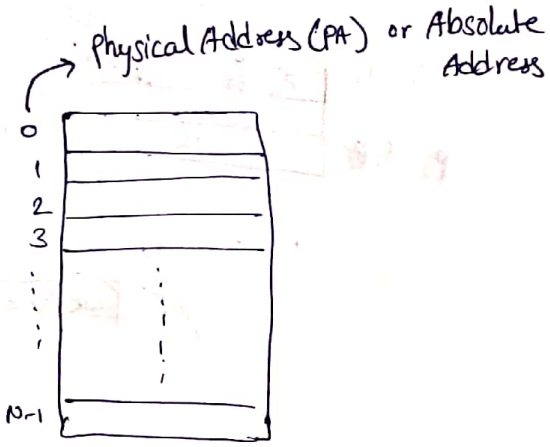
Completion times can be seen in the gantt chart.

2. Non-Contiguous Memory Allocation Techniques

Logical Address Space (LAS) vs Physical Address Space (PAS)

(or)

(Virtual Address Space (VAS))

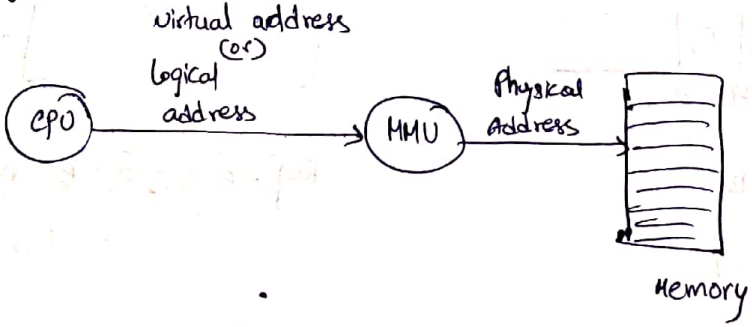


* → In all uniprogrammed OS, the address generated by the program is physical address

* → Also in compile time binding & load time binding, the address generated ~~is phy~~ by program is physical Address

* → In runtime binding, the address generated by the program is logical address which is not equal to the corresponding physical address.

* → In M.Pr systems we allow CPU to generate logical address and MMU translates it to PA. ~~This way~~ This approach provides protection because we are not letting executing program generate PA and access memory directly.



→ Based on the technique we use the hardware of MMU changes. It may be page table (or) segment table

→ If size of PA is 'n' bits, then PAS is 2^n words.
also size of MAR is n bits.

→ If size of VA is 'm' bits, then LAS (or) VAS is 2^m words
also size of address bus of CPU is m bits.

→ The address binding used here is runtime binding.

→ LAS = 2^{LA} words

LA = $\log_2$ LAS bits

→ PAS = 2^{PA} words

PA = $\log_2$ PAS bits

→ theoritically

LAS ≥ PAS

LAS > PAS

LAS < PAS

LAS = PAS

→ But pratically

LAS > PAS

(i) Paging: (simple paging)

Assume

LAS = 8kB; PAS = 4kB Page size = 1kB (PS)

⇒ LA = 13 bits PA = 12 bits

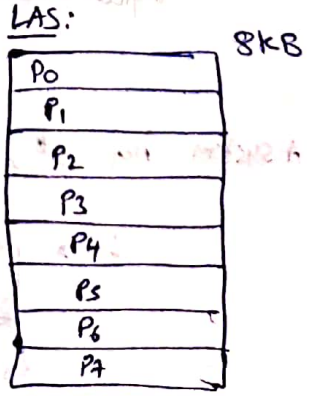
Step 1: Organization of LAS on PAS

→ LAS is divided into equal size units called pages.

→ Generally page size is power of 2 (but not necessary though)

→ no of pages = $\frac{8kB}{1kB}$ = 8 pages

i.e., $\boxed{\text{no of pages}(N) = \frac{LAS}{PS}}$



→ Every page is given page number or page id.

→ Page No (P) = $\log_2 8 = \log_2 N$ i.e., $\log_2 N = \log_2 8 = 3$ bit

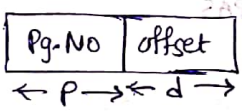
$N = 2^P$

→ Page offset (d) = $\log_2$ PS (bits) i.e., pg. off = $\log 1KB = \log 2^{10} = 10$

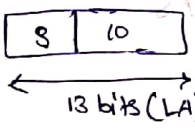
$PS = 2^d$

→ Here we generally select req page first, then using page offset then we go to required word.

→ Logical Address format:



i.e.,



Eg: LA = 24 bits; & PS = 4KB

then

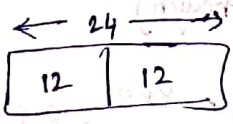
No of pages, $N = \frac{2^{24}}{2^{12}} = 2^{12}$ pages

$LAS = 2^{24} = 16$ MB

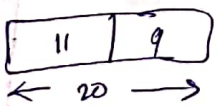
$P = \log_2 2^{12} = 12$ bits

$d = \log_2 PS = 12$ bits

Logical address format



Eg: A system has 2k pages with page offset of 9 bits then



$P = \log 2^{11} = 11$

⇒ LA = 20

$LAS = 2^{20} = 1$ MB

P = 11

d = 9

Eg: System supports a LA of 32 bits if page size is 16kB. Calculate no of pages in System

Sol:

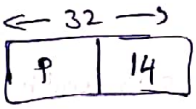
LAS = 2^{32}

PS = 2^{14}

no of pages = $\frac{2^{32}}{2^{14}} = 2^{18}$ pages

$\therefore N = 2^{18}$

Method 2:



$\Rightarrow p = 32 - 14 = 18$

$\Rightarrow$ no of pages = $N = 2^{18}$

Step 2: Organization of PAS:

→ PAS is divided into equal size units known as frames (page frames)

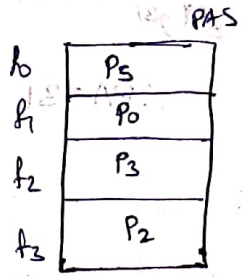
→ Frame-size (FS) = Page size

→ Any page can be stored in any frame (Non contiguous allocation)

In the example,

PAS = 4kB PS = 1kB

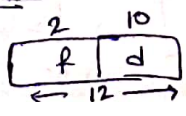
no of frames (M) = $\frac{PAS}{PS}$



→ frame No(f) = $\log_2 M$ bits

→ frame offset = page offset = d

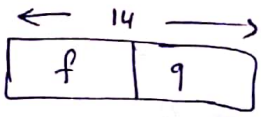
→ physical address format:



Eg: PAS = 16KB ; d = 9 bits

then

~~Mc PAS~~ Phy address format



⇒ f = 5

⇒ no of frames (M) = 2^5

⇒ page size = 2^9 = 512 Bytes

Eg: System has a LA of 35 bits with 4k pages. If it has 256 frames then calculate size of PA

Sol:

LAS = 2^{35} = 32 GB

no of pages (N) = 4K = 2^{12}

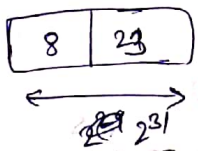
⇒ page size = $\frac{LAS}{N} = \frac{2^{35}}{2^{12}} = 2^{23}$ bytes

⇒ page offset (d) = 23

no of frames (M) = 256

∴ frame number (f) = $\log_2 256 = 8$

⇒

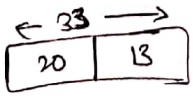


∴ ~~PA = 29~~ ∴ PA = 31

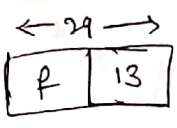
(Q28) System has LA of 33 bits with a page size of 8 kB. If PA is 29 bits. Calculate no of frames.

Sol:

LA



PA



⇒ f = 16

⇒ no of ~~pages~~ frames = 2^{16} = 64k frames.

Process Management Volume I:

(P/3) System calls are invoked by a privilaged instruction which generates a software interrupt.

Not all privilaged instructions generate s/w interrupt.

∴ opt (a)

(P/7) Scheduler process comes only after a process terminates.

Power failure also causes interrupt.

IO Redirection

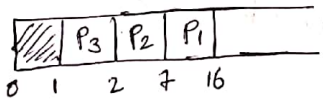
(P/8) Every process is associated with 3 standard files (by default)

Stdin: keyboard
Stdout: monitor
stderr: monitor

IO redirection is a concept in which some other file may be used in the place of any of the 3 standard files.

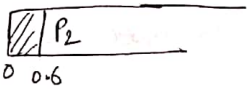
(P/10) Option verification:

a)



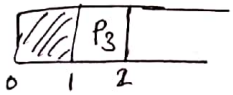
Schedule length = 16 - 0 = 16

b)



↳ not possible cuz idle time must be 0.6

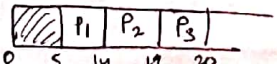
c)



idle time = 1

∴ not possible

d) This sequence is possible cuz we can keep the processor idle if we need.



Here both option a & d are possible

Now we need to decide which is optimal.

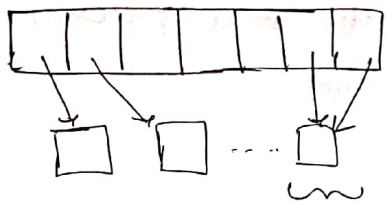
Optimality depends on avg. TAT, avg wt., CPU utilization etc.

Also opt @ follows SJF which is optimal

∴ opt @

P/25

RQ:



This process gets more time in one round.

This action results in giving more priority to the process.

If this prioritizing is more, it leads to starvation.

P/26

(i) low B.T are given high priority

(ii) The last queue of multilevel feed back uses FCFS.

(iii) High priority is given to low arrival time processes.

(iv) RR & SJF don't have any relation

P/27

Here we take response time as waiting time.

Thus to get least waiting time is given by SJF

Scheduling seq: is

∴ if (x < 3)

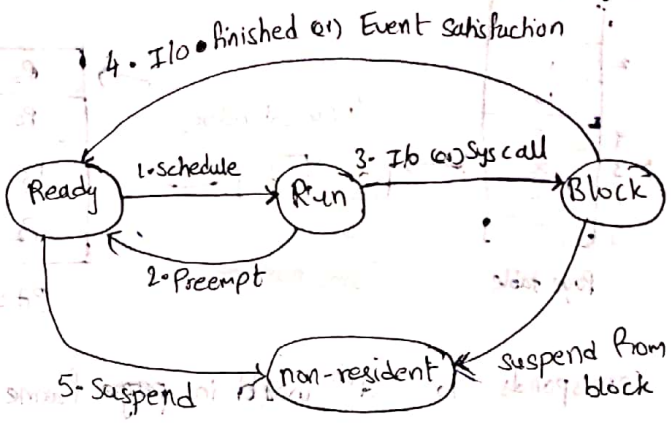
<x, 3, 5, 6, 9>

else if (x < 5)

<3, x, 5, 6, 9>

.....

P/28 Non-resident means outside on disk.
i.e., suspended process.



Step3: Organization of MMU

MMU in paging is page table or page map table or address translation table.

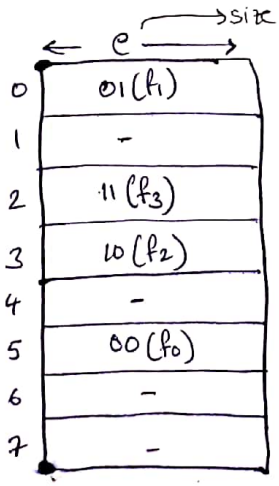
→ Each processes has its own page table

→ Page tables are generally stored in memory.

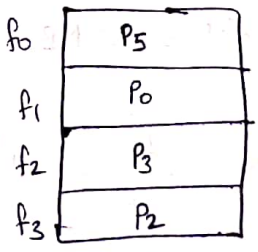
→ page tables is organized as a set of entries and each entry is known as page table entry.

→ No of entries in PT = no of pages in LAS

→ PT entries contains the frame no (f) in which the page is present.



Page table (MMU)

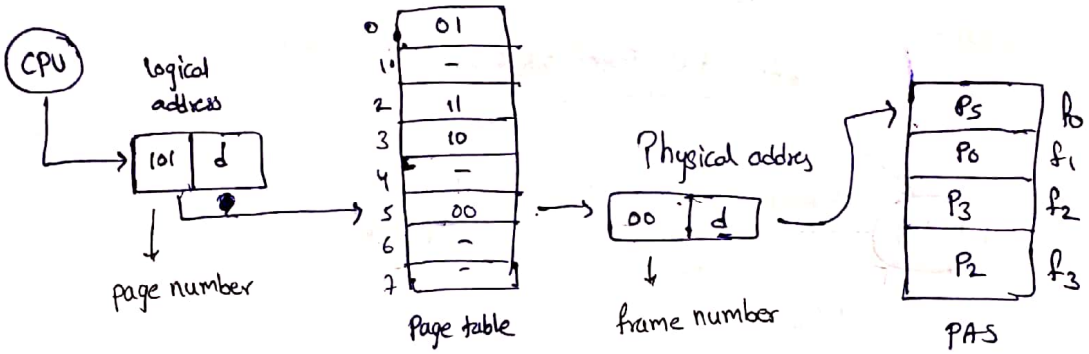


PAS

→ PT entries are generally in bytes.

→ Page table size = No of pages in LAS x size of each entry = N x e

address translation (logical to physical)



∴ 101 | d corresponds to d^{th} word in frame 0 of memory.

Q29 LA = 34 bits; PA = 27 bits;
PS = 8 kB;

LAS = ? N_{pages} = ? P.A.S = ?

M_{frames} = ? p = ? f = ?

d = ?

Find format of logical address and physical address

Find size of page table (approx).

Sol:

$LAS = 2^{34} = 16\ GB$

$PA = 2^{27} = 128\ MB$

$N_{pages} = \frac{2^{34}}{2^{13}} = 2^{21}$ pages

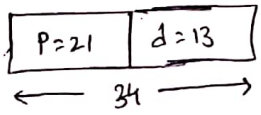
$M_{frames} = \frac{2^{27}}{2^{13}} = 2^{14}$ frames

$d = \log_2 PS = \log_2 2^{13} = 13$

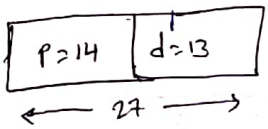
$p = \log_2 N = 21$

$f = \log_2 M = 14$

logical address format



physical address format



size of page table = no of pages (entries) × size of each entry

$= 2^{21} \times 16$ bits

$= 2^{22}$ bytes (size entry = 16)

$\cong 4$ MB

Q30 Consider a ~~regis~~ system supporting a 32 bit VAS with a page size of 4kB. If PAS is 64 MB calculate the approximate PT size in bytes:

Sol:

$PS = 2^{12}$

~~VAS~~ $VAS = 2^{32}$

$PAS = 2^{26}$

$M_{pages} = \frac{2^{32}}{2^{12}} = 2^{20}$ pages

$M_{frames} = \frac{2^{26}}{2^{12}} = 2^{14}$ frames

size of PT entry = 14 bits $\cong$ 2 bytes

size of PT = $2^{20} \times 2 = 2$ MB

H4/6

$PS = 2^{13}$ bytes

$PTS = 24 \times 2^{20}$ bytes

~~$= 3 \times 2^{23}$ bytes~~

$e = 24$ bits $= 3$ bytes

no of pages $= \frac{24 \times 2^{20}}{3} = 8 \times 2^{20} = 2^{23}$ pages

logical address:

23	13

$LA = 23 + 13 = 36$ bits

$LAS = 2^{36} = 64$ GB

H4/7

$VA = l$

$N = Z \Rightarrow p = \log_2 Z$

$M = H \Rightarrow f = \log_2 H$

LA:

p	d

(l spans both fields)

$l = p + d \Rightarrow d = l - p$

$$d = l - \log_2 Z$$

PA:

f	d

$PAS = 2^{f+d} = 2^{f+l-\log_2 Z} = \frac{2^{f+l}}{Z} = \frac{2^{\log_2 H + l}}{Z} = \frac{H \cdot 2^l}{Z}$

$\therefore d = l - \log_2 Z$, $PAS = \frac{2^{f+l}}{Z} = \frac{H \cdot 2^l}{Z}$

H4/8

$LA = 32$ bits $\Rightarrow LAS = 2^{32}$

$PTES = 4$ bytes

$PTS = 1$ frame $=$ page size i.e., $PTS = PS$

$$N \times e = PS$$

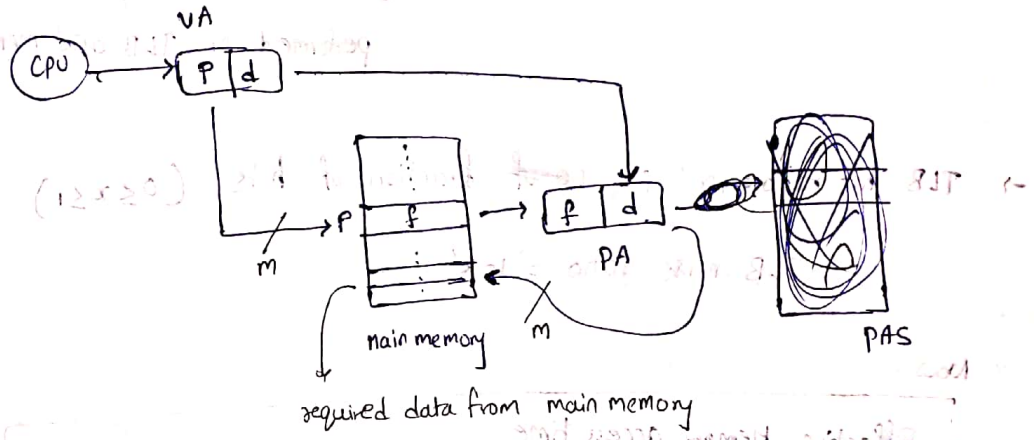
$$\frac{2^{32}}{PS} \times 4 = PS$$

$$\Rightarrow PS^2 = 2^{34}$$

$$\Rightarrow PS = 2^{17} \text{ bytes} = 128\text{kB}$$

Performance of Paging:

1) Temporal issue (Impact on time):



↳ Let main memory access time be 'm'.

so address translation time = m (overhead due to page table)

data access time = m

Due this throughput decreases

∴ Effective memory access time = 2m

Now we reduce this overhead 'm' using the concept of TLB.

TLB: Translation Look aside Buffer. (Logical address cache)

TLB is similar to cache.

→ TLB contains previously resolved VA and its corresponding PA

→ Let TLB access time be 'c' (c <<< m)

→ Now if corresponding VA is found in TLB, which takes less access time, we directly use the corresponding PA to access required data in the memory.

→ The event of finding required address in TLB is called TLB hit.

Here access time = c+m

→ The event of missing required address in TLB is called TLB miss

Here access time = c+2m (Also '2m' if a parallel search is performed on TLB and memory)

→ TLB hit ratio 'x' is fraction of hits $(0 \le x \le 1)$

=) TLB miss ratio = '1-x'

Now,

Effective Memory access time with TLB	$= x(c+m) + (1-x)(c+2m)$

→ Hit ratio = $\frac{\text{no of hits}}{\text{no of references}}$

Eg: m = 100ns

c = 20ns

x = 90%

Sol:

without TLB;

Effec. Mem. AT = 2(100) = 200

With TLB

Effe Mem AT = .0.9(100+20) + 0.1(20+200)

= 108+22 = 130ns (less than without TLB)

If 'x' were 10% then

Effe. Mem. AT = 0.1(120) + 0.9(220) = 210ns (more than without TLB with TLB)

→ The TLB we used is Logical address cache.

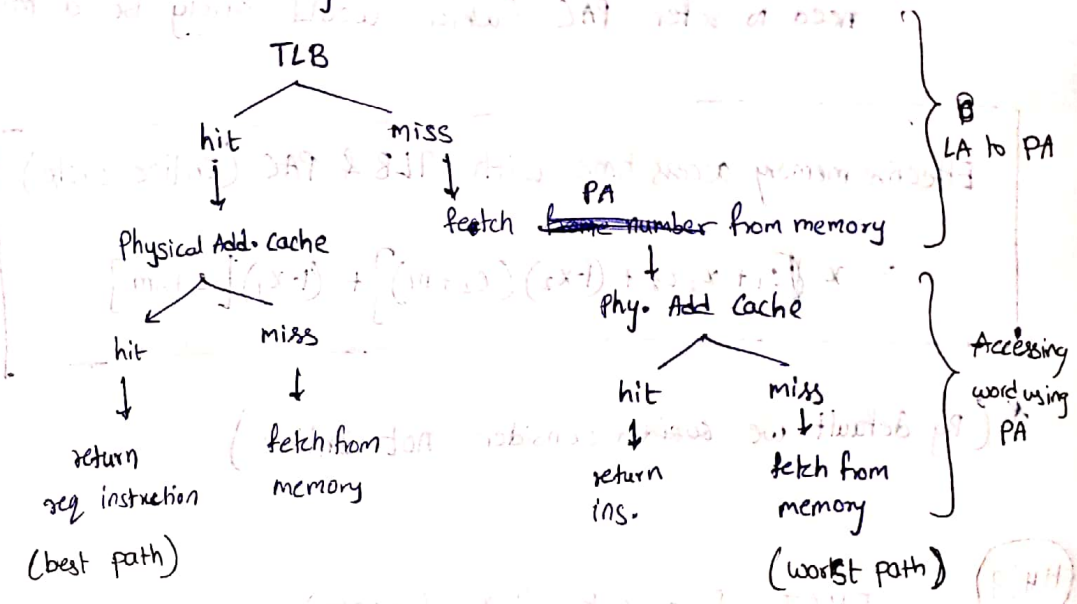
~~If we use physical address~~

After obtaining required PA we need to access memory again.

To reduce this we can use physical address cache. (PAC)

Physical address cache contains ~~both~~ PA and corresponding data.

→ with this now accessing instruction is



→ Let hit ratio of TLB = x_1

hit ratio of PAC = x_2

access time of TLB = c_1

acess time of PAC = c_2

Memory access time = m

Effective memory access time with both TLB & PAC (This valid if cache is not In-line)

$$= x_1 \left[c_1 + x_2 c_2 + (1-x_2)(c_2+m) \right] + (1-x_1) \left[\underbrace{c_1+m}_{\text{for PA}} + x_2 c_2 + (1-x_2)(c_2+m) \right]$$

* → ~~not~~ not In-line cache means both TLB & PAC are independent

* → In-line cache means TLB & PAC are dependent...

~~For example~~

i.e., whenever we remove an entry from TLB (VA & PA) the corresponding entry in PAC (PA & instruction or data) are also removed.

In this case if TLB miss occurs then there is no need to refer PAC (which would surely be a miss)

Effective memory access time with TLB & PAC (Inline cache)

$$= x_1\left[c_1 + x_2 c_2 + (1-x_2)(c_2+m)\right] + (1-x_1)\left[c+2m\right]$$

* (By default we ~~consider~~ consider not Inline)

H4|9

$$EMAT_1 = D = x_1 k + (1-x_1)(k+2m) \quad -①$$

$$EMAT_2 = Z = x_2 k + (1-x_2)(k+2m) \quad -②$$

$$\Rightarrow ① - ② \Rightarrow D - Z = (x_1 - x_2)k + (1 - x_1 - 1 + x_2)(k+2m)$$

$$D - Z = (x_1 - x_2)k - (x_1 - x_2)(k+2m)$$

$$D - Z = (x_1 - x_2)\left[k - k - 2m\right]$$

H4|9

without TLB

$$EMAT = D = 2m \Rightarrow m = \frac{D}{2}$$

with TLB

$$EMAT = Z$$

$$x\left(k + \frac{D}{2}\right) + (1-x)(k+D) = Z$$

$$xk + \frac{Dx}{2} + k + D - xk - xD = Z.$$

$$\frac{Dx}{2} = k+D-z$$

$$x = \frac{2(k+D-z)}{D}$$

(14/10) LA = 32 $\Rightarrow$ LAS = 2^{32}

PS = 2^{13}

word size = 4 bytes

e = Page table entry size = 4 bytes (1 word)

$$N = \frac{LAS}{PS} = 2^{19} \text{ pages}$$

Size of page table = N x e = 2^{19} words

time req to load page table = $2^{19} \times 100$ nsec

total process runtime = 100 msec = 100×10^6 nsec

$$\text{req. fraction} = \frac{2^{19} \times 100}{100 \times 10^6} = \frac{2^{19}}{10^6} \cong 0.5242$$

i.e., 52.42%

2) Spatial Issue (space optimization)

Assume

LA = 32 bits ; PS = 4KB

$\Rightarrow$ $N_{pages} = 2^{20}$

Page table size, PTS $\propto$ N_{pages}

PTS = 1M entries

If e(PTES) is 4 bytes

then PTS = 4MB

i.e., Every process will have a page table of 4MB

* It is not desirable to have larger page tables. So we need to reduce page table size of a process.

Reducing PT size of a process

i) Increase page size:

$PTS = N * e$

$$PTS \propto N \propto \frac{1}{PS} \Rightarrow PTS \propto \frac{1}{PS}$$

However increasing PS has its own draw back.

Eg: Assume we have a program of size 1026 bytes

if PS = 1024 bytes

here we require 2 page in which we use only 2 bytes of 2nd page

i.e., Internal fragmentation = 1022 bytes

if PS = 2 bytes

then we require 513 pages.

i.e., Internal fragmentation

* ⇒ Increasing page size increases internal fragmentation

So, now, we need to come up with an optimal page size

Optimal page size:

Let VAS = S bytes

PS = P bytes

PTE = 'e' bytes

$\Rightarrow PTS = \frac{VAS}{PS} \times PTE$

$$\Rightarrow \boxed{PTS = \frac{S}{P} \times e}$$

Internal fragmentation can happen only in the last page

In the worst case IF could be almost equal to page size

$\therefore$ Avg. IF = P/2

→ page table size & Avg IF are clearly overheads

$\therefore$ optimal page size is such that total overhead is minimum

i.e., $\frac{se}{p} + \frac{p}{2}$ is minimum

let $f = \frac{se}{p} + \frac{p}{2}$

$$\frac{df}{dp} = \frac{-se}{p^2} + \frac{1}{2} = 0$$

$$\Rightarrow \frac{se}{p^2} = \frac{1}{2} \Rightarrow P = \sqrt{2se}$$

$\therefore$ optimal page size, $P = \sqrt{2se}$
w.r.to overhead of
PTs & IF

H4/11

$VAS = 2^{16}$

$PS = 2^{12}$

$N = 16$ pages

$N_{text} = 8$ pages

$N_{data} = 5$ pages

$N_{stack} = 4$ pages

} 17 pages

a) ∴ does not fit

a) Assume PS = 4 bytes

$N_{pages} = \frac{2^{16}}{2^2} = 2^{14}$ pages $= 16384$

text : $\frac{2^{15}}{2^2} = 2^{13}$ pages $= 8912$

data = $\frac{16386}{4} = 4096.5 \Rightarrow 4097$ pages

Stack = $\frac{15870}{4} = 3967.5 \Rightarrow 3968$ pages

} 16257

Thus for PS = 4 bytes, the program fits

b) Assume max possible page size = 2^x bytes

$\Rightarrow N_{pages} = \frac{2^{16}}{2^x} = 2^{16-x}$ pages

text : $\frac{2^{15}}{2^x} = 2^{15-x}$ pages

data : $\left\lceil \frac{16386}{2^x} \right\rceil$ pages

stack : $\left\lceil \frac{15870}{2^x} \right\rceil$ pages

$\Rightarrow 2^{15-x} + \left\lceil \frac{16386}{2^x} \right\rceil + \left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{16-x}$

$2^{15-x} + \left(2^{14-x} + 1\right) + \left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{16-x} \quad (\forall x \geq 2)$

$\Rightarrow \left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{16} - 2^{15} - 2^{14}}{2^x} - 1$

$\Rightarrow \left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{14}(4-2-1)}{2^x} - 1$

$$\left\lceil \frac{15870}{2^x} \right\rceil \leq \frac{2^{14} - 2^x}{2^x}$$

$$\left\lceil \frac{15870}{2^x} \right\rceil \leq 2^{14-x} - 1$$

put $x = 5$

$$496 \leq 512 - 1 \checkmark$$

put $x = 7$

$$124 \leq 128 - 1 \checkmark$$

put $x = 8$

$$62 \leq 64 - 1 \checkmark$$

put $x = 9$

$$31 \leq 32 - 1 \checkmark$$

put $x = 10$

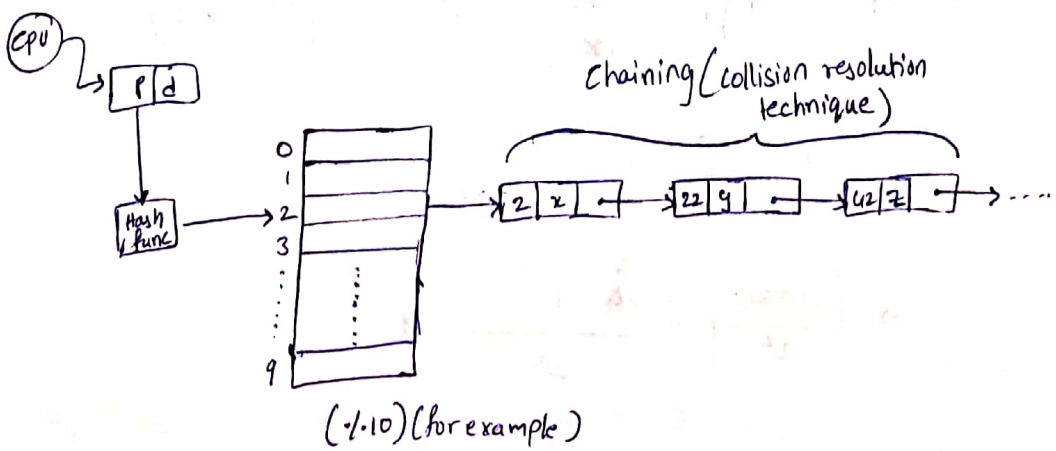
$$16 \leq 16 - 1 \times$$

∴ Max page size such that the program could fit is

$$2^x = 2^9 = 512 \text{ bytes.}$$

(ii) Hashed Paging | paging with Hashing

* This techniques helps reduce size of page table.



→ LAS says max possible process size. However we may have smaller processes. In this case size of hash page table is very less than normal page table.

For example,

LAS = 2^{32} = 4GB (max possible process size)
PS = 4KB

In normal page table size of page table for any process is same.

Eg Assume we have process of 128KB

$$\Rightarrow \text{no of pages} = \frac{128}{4} = 32$$

∴ Hash page table will have only 32 entries

However normal page table has 2^{20} entries.

→ However hash page table may require extra time for searching through the list. So hash page table is space efficient but not time efficient.

(iii) Inverted Paging / Reverse paging:

(This topic will be discussed in the end)

2. Multi-Level Paging / Hierarchical Paging:

* The basic objective of multi-level paging is to reduce page table size overhead.

* Multilevel paging is paging on page table.

* Paging generally involves 3 steps:

(i) divide the address space in pages

(ii) store the pages into frames of memory

(iii) Access the pages of the address space through page table.

For example

Consider

LA = 32 bits , PS = 4kB

Now $N_{pages} = 2^{20} = 1M$ pages

Assume size of page table entry = 4 bytes

∴ PTS = 4MB

Now the page table itself is more than size of one page

PT must be stored in $\frac{4MB}{4kB}$ = 1K pages

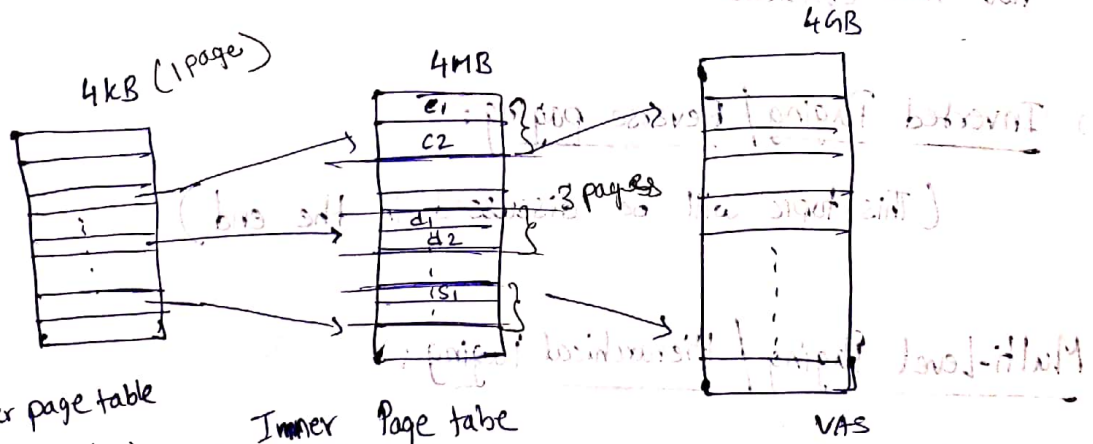
> Without multilevel paging, the page table must be stored contiguously if it spans across several pages

Now we page the page table using another page table (Outer page table)

Size of outer page table = 1K × 4 bytes (→ no of pages in inner page table)

= 4kB

i.e., 1 Page exactly



→ Now in the above scenario, assume we have a process of size 20 kB.

i.e., $\frac{20kB}{4kB}$ = 5 pages

~~a normal page~~

Assume these 5 pages are

(2 code + 2 data + 1 stack)

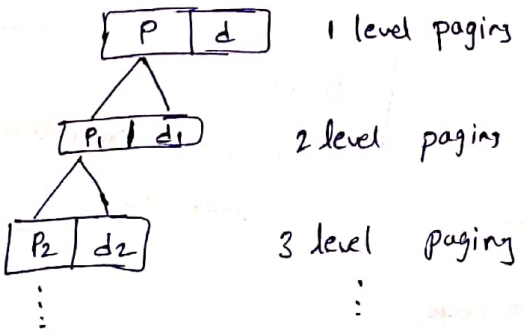
→ Assume these 3 parts are present in 3 different location such that code is in 2 contigious pages, data is in 2 contigious pages and stack in one page such that we need 3 pages of page table. (As shown in the figure)

→ Now in the case of multilevel paging we need outer pagetable, and the 3 pages of inner page table for the execution of the processes. (i.e., 4KB).

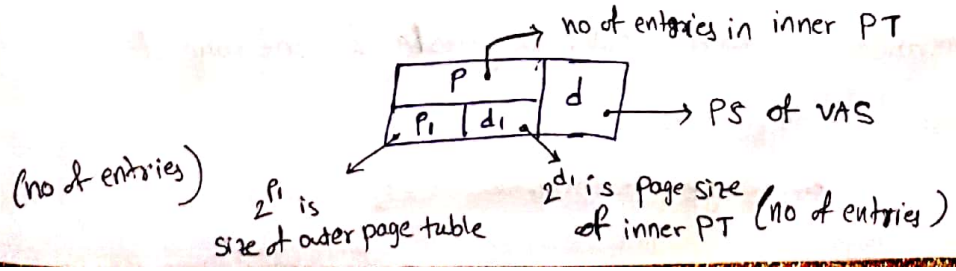
→ If we don't use multilevel paging then we must have all 1M entries of page table (i.e., 4MB) in main memory.

→ Thus multilevel paging reduces the overhead of page table size that is to be stored in main memory.

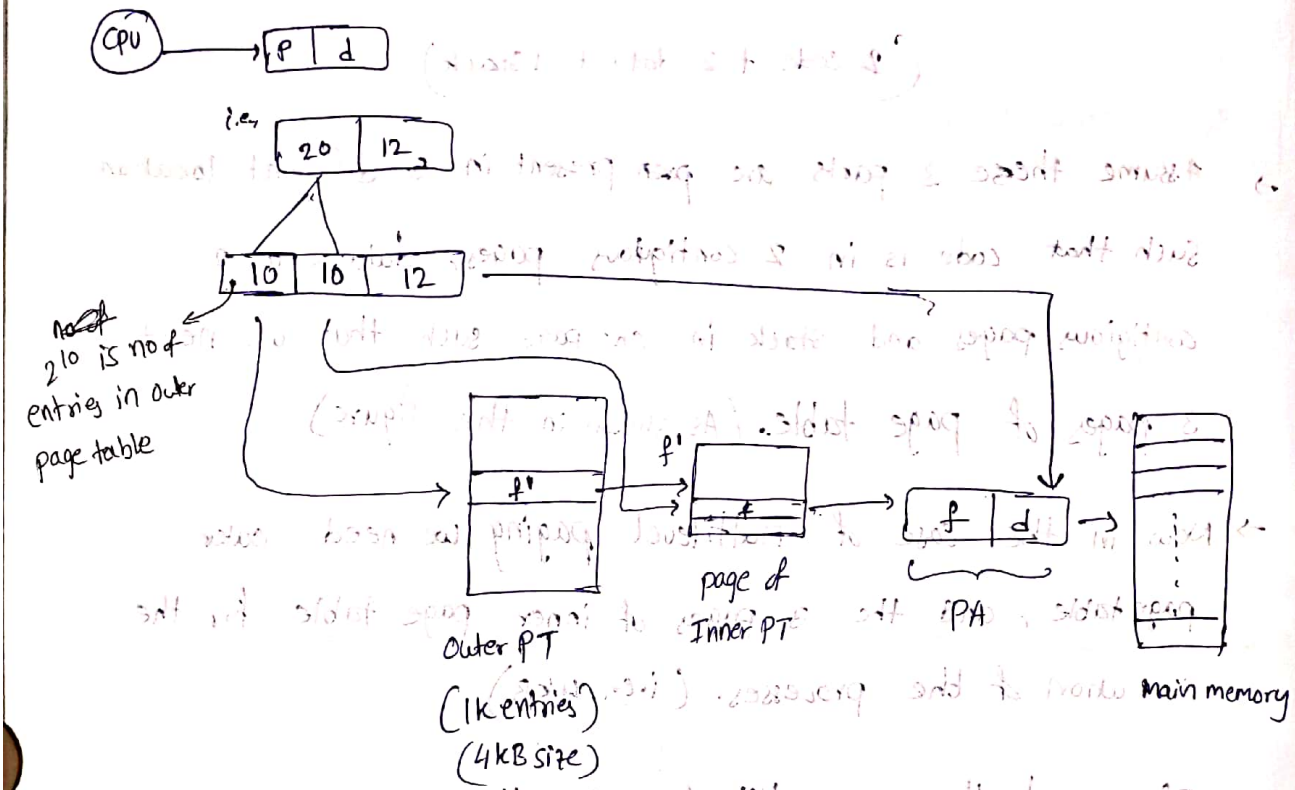
→ **Addressing:**



∴ Final logical address format for 2 level paging is



VA = 32 bits PS = 4kB

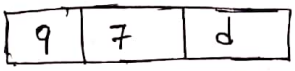


→ Here f' is frame of main memory in which required page of inner page table is located.

→ f is frame of main memory in which required page of a process is stored

→ For there offset 'd' is used to obtain required word.

H4 12



In this problem page size of outer table & inner table are different

$\Rightarrow d = 28 - 16 = 12$

(i) $\Rightarrow$ pg size $= 2^{12} = 4$ KB,

$N_{pages} = \frac{2^{28}}{2^{12}} = 2^{16}$ pages

(ii) for translation we need outer pagetable & one page of inner page table in memory

~~i.e. 2 page = 2 * 4kB = 8kB~~

$2^9 * 4B + 2^7 \times 4B = 2^{11} + 2^9$

$= 2^9(5)$

$= 512(5)$

$= 2560$ bytes

$= 2.5$ KB

18/08/20

* If we use 2-level paging, we require 3 memory accesses in total.

~~i Memory~~

∴ Effective memory access time = 3m without TLB

* Silly for n-level paging without TLB

EMAT = (n+1)m

(H4/13) VA = 46 bits ⇒ VAS = 2^{46} = 64 TB

PTES = 4 bytes = 2^2

let PS = 2^x

$N_{pages} = \frac{2^{46}}{2^x} = 2^{46-x}$

Size of 3rd level page table = $2^{46-x} \times 2^2 = 2^{48-x}$ bytes

no of pages in 3rd level page table = $\frac{2^{48-x}}{2^x} = 2^{48-2x}$

~~Size~~ size of 2nd level page table = $2^{48-2x} * 2^2 = 2^{50-2x}$

no of pages in 2nd lvl PT = $\frac{2^{50-2x}}{2^x} = 2^{50-3x}$

Size of 1st lvl page table = $2^{50-3x} \times 2^2 = 2^{52-3x}$

Given that

Size of 1st lvl page table = size of frame

i.e., $2^{52-3x} = 2^x$

$\Rightarrow 52-3x = x$

$\Rightarrow x = 13$

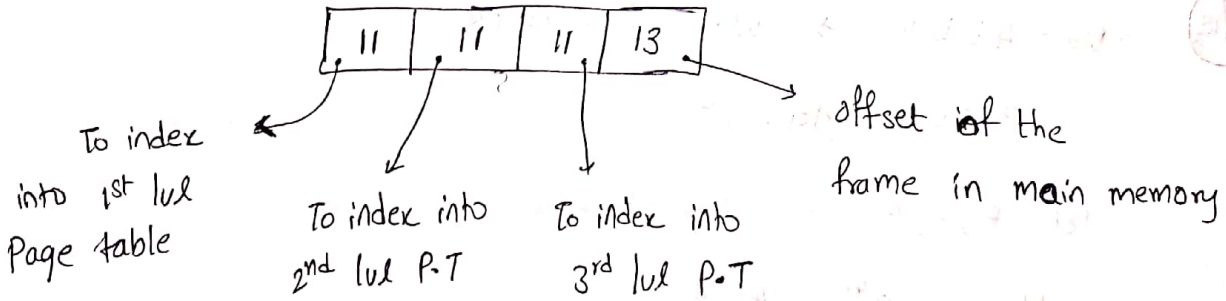
$\therefore$ page size = $2^{13} = 8KB$

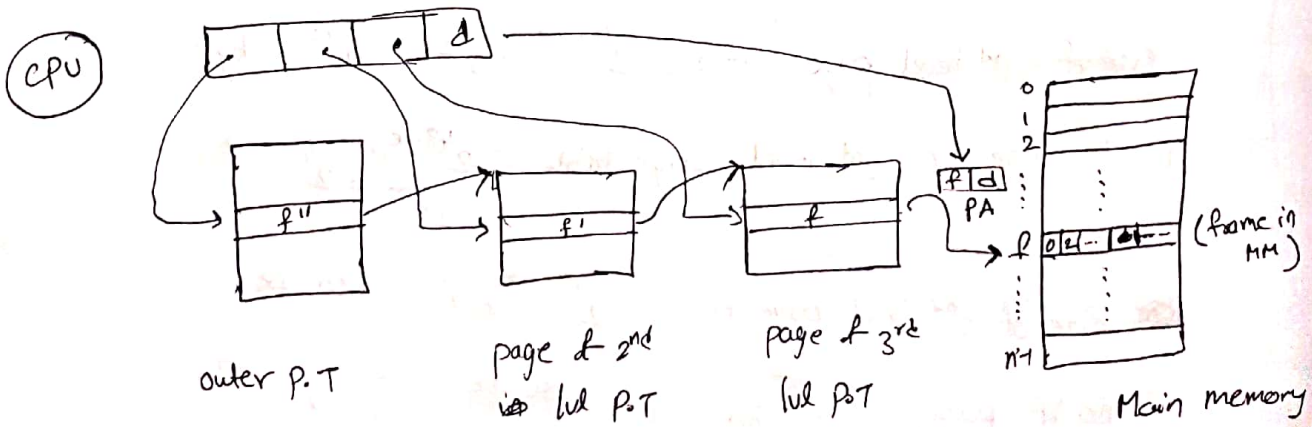
Page table entrie size is same at all levels of page tables.

$\therefore$ no of PT entries in one page of page table

$= \frac{2^{13}}{4} = 2^{11}$

$\therefore$ logical address format





Method 2:

Let 2^x be req. PS

Size of outer (1st lvl) P.T = 2^x

no of entries = $\frac{2^x}{PTES} = \frac{2^x}{4} = 2^{x-2}$

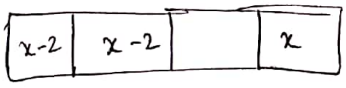
LA:

Silly after indexing to outer P.T, we go to a page of 2nd lvl

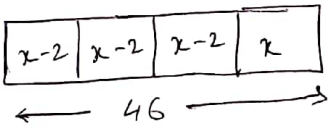
~~pag~~ P.T

no of entries in ~~second~~ a page 2nd lvl PT = 2^{x-2}

∴ LA



Silly we get



∴ ~~x=2~~ $3(x-2)+x = 46$

$\Rightarrow 4x = 52 \Rightarrow x = 13$

∴ Page size = $2^x = 2^{13} = 8$ kB

From the above question we can come up with a general formula for P.T with n level paging

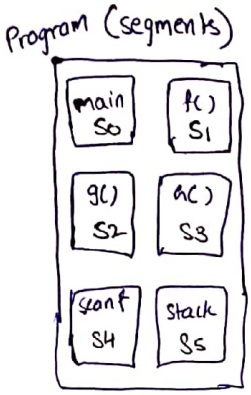
Assum VAS = 2^s bytes ; PS = 2^x bytes ; PTE = 2^c bytes

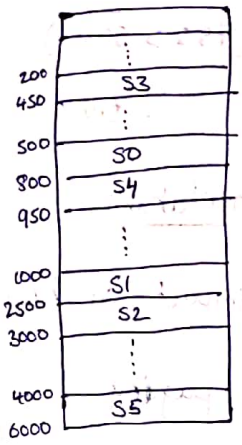
Size of OPT n-level paging = $\left\lceil 2^{s-nx+nc} \right\rceil$ bytes

(with uniform pages at all levels)

II. Segmentation

Paging does not preserve users view of program





→ At user view program is divided into segments (each segment may be of diff sizes).

→ Each segment has segment id.

→ Different segments a stored non-contiguously. However a segment itself is store totally at one location (contiguously)

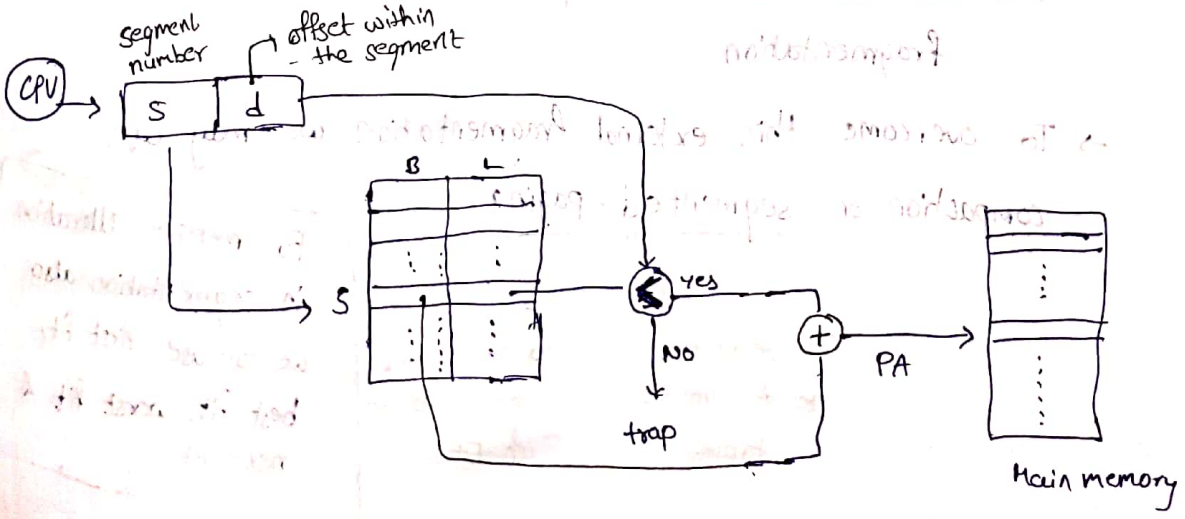
→ Here MMU is segment table.

	B	L
0	500	300
1	1000	1500
2	2500	500
3	200	250
4	800	150
5	4000	2000

B - Base address of segment

L - length of segment

→ Every process has its own segment table



Eg: For above example

LA<3,50> ⟶ PA<200+50> = 250

Eg: LA<2,755>

└ (755 > 500)

∴ trap

Performance of Segmentation

(i) Temporal issue (impact on time):

* Exactly same as paging

i.e., EMAT = 2m without TLB

Silly we can add TLB & PAC (same equation as in the case of paging)

(ii) Spatial issue

* Large segment table is undesirable

* One of ways to reduce size of segment table is to use paging on segment table.

Note:

* In case of segmentation, we don't divide PAS.

∴ PAS is organized with variable partitioning (MVT)

we know that variable partitioning suffers from external fragmentation

→ To overcome this external fragmentation we may use compaction or segmented-paging

- reduces size of segment table
- to overcomes the problem of EF

For memory allocation in segmentation also we can use first fit, best fit, worst fit & next fit

Note:

	IF	EF
Paging	✓	X
Segmentation	X	✓

IF - Internal Fragmentation

EF - External Fragmentation

Q31) Consider below segment table

segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

what the physical addresses for following logical addresses

a) 0,430

219 + (430 - 0) = 219 + 430 = 649

b) 1,10

2300 + 9 = 2309 2300 + 10 = 2310

c) 2,500

500 > 100

∴ trap

d) 3,400

1327 + 400 = 1727

e) 4,112

112 > 96

∴ trap

Segmented - Paging:

Assume LA = 34 bits

<s,d> = <18, 16> bits

⇒ Maximum segment size = 2^{16} = 64kB

total no of segments possible = 2^{18} = 256k segments.

→ To avoid EF we apply paging on segment.

(i) Divide address space (segment) into pages

(ii) Store these pages in frames.

(iii) Access the frames through page table of a segment

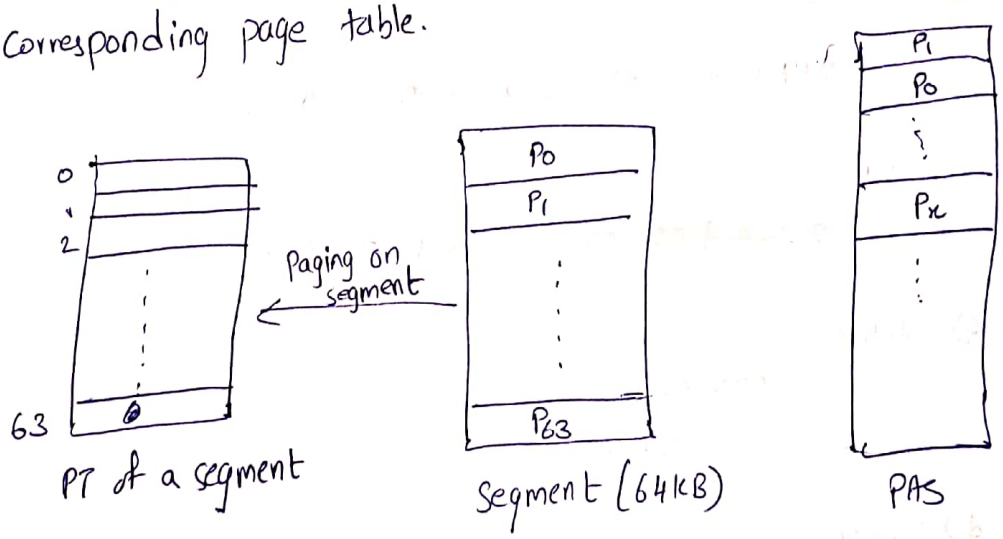
Let page size = 1KB

no of pages in a segment = $\frac{64\ KB}{1KB}$ = 64 pages

→ Now PAS contains frames but not segments.

→ Now the page table of segment contains an entry for each page of the segment

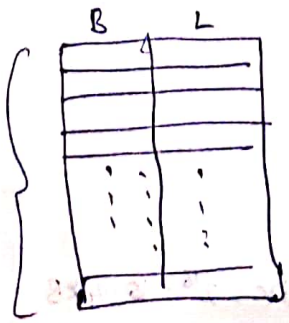
→ Now every segment must be accessed through its corresponding page table.



no of Page tables = no of segments.

Thus we have 256k page tables.

Structure of segment table:

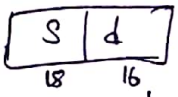


256k entries
(i.e., no of segments
= no of page tables)

B - base address of corresponding page table of the segment

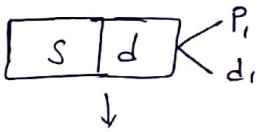
L - length of segment

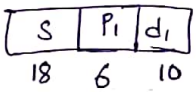
Logical address format



we have applied paging on segment
i.e., on 'd'

∴



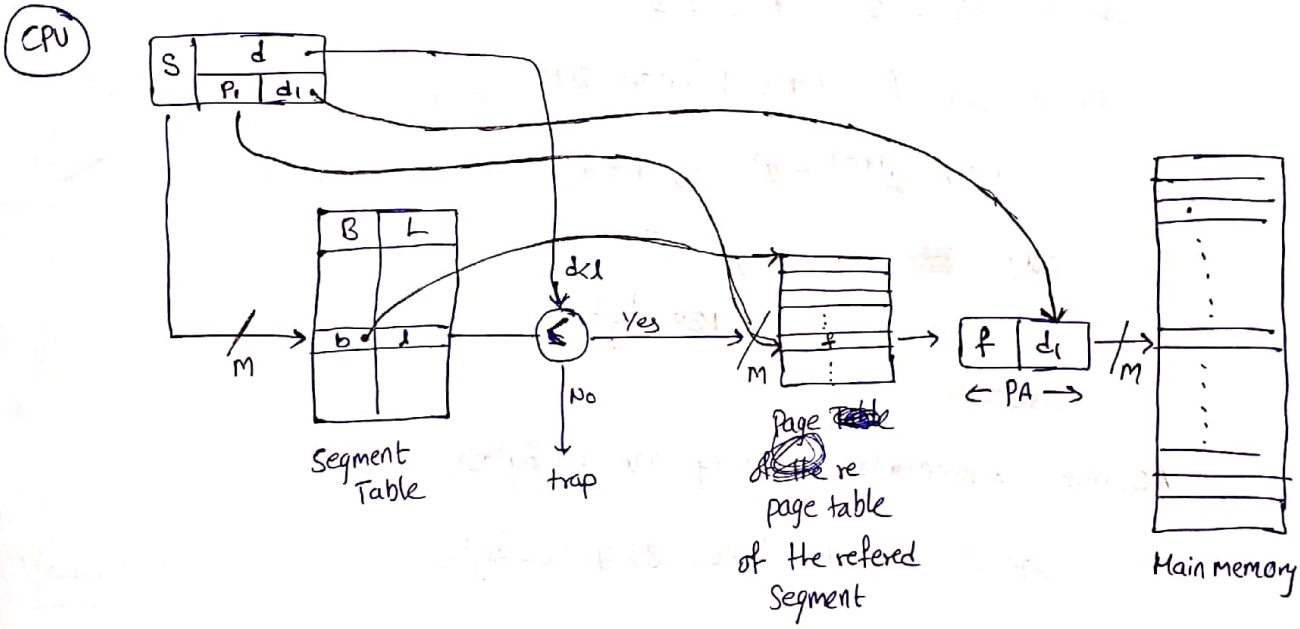


2^{P_1} - no of entries in each page table

2^{d_1} - size of page

d_1 say offset within the page

Address Translation



* Here we need to access segment table, then corresponding page table then the required data.

i.e., 3 memory accesses

∴ EMAT = 3M

Like in the case of paging we can use TLB to reduce EMAT.

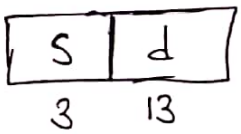
Here also the TLB contains VA & corresponding PA.

(14/14) VAS = PAS = 2^{16} bytes = 64 kB

no of segments in VAS = 8

size of each segment = $\frac{2^{16}}{2^3} = 2^{13}$ = 8 kB

VA:



Pg table entry size = 2 bytes

Let page size = 2^x

no of pages in a segment = $\frac{2^{13}}{2^x} = 2^{13-x}$

i.e., no of entries in a PT = 2^{13-x}

Size of PT = $2^{13-x} * 2 = 2^{14-x}$

given size of PT = 1 frame size

i.e., $2^{14-x} = 2^x \Rightarrow x = 7$

∴ page size = $2^x = 2^7$

= 128 bytes

Size of

Assume segment table entry size = 4 bytes

size of segment table = 8 × 4 = 32 bytes

Virtual Memory (VM):

VM given an illusion to the programmer that a huge amount of memory is available for executing larger programs in small memory area.

→ It is a technique that supports the goal of managing execution of larger programs in small memory area.

→ CPU assumes that it can executes a program as big as VAS.

→ However PAS may not be as big as VAS.
So we need to store the program on disk.
From disk we load required pages into memory.

→ Sometimes required page may not be found in main memory. In that case required page is loaded into memory from the disk. This is known demand paging.

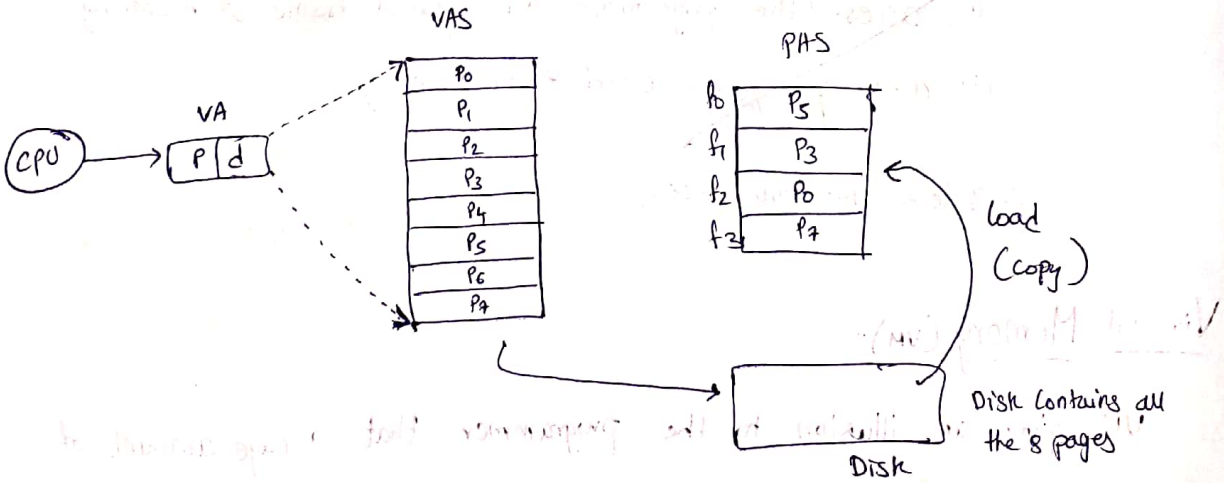
Demand paging:

Loading page from disk to memory at runtime based on demand is known as demand paging.
i.e., Virtual memory is implemented through demand paging.

VM implementation through Demand Paging:

Let VAS = 8KB ; PAS = 4KB ; PS = 1KB

⇒ N = 8 M = 4 d = 10



→ Whether the req page is present in the memory or not is known by valid/invalid (V/I) bit which is in the page table.

→ If required page is found in the memory then it is called page hit.

→ If required page is not found in the memory then it is called page miss or page fault.

0	10	1
1	–	0
2	–	0
3	01	1
4	00	0
5	00	1
6	–	0
7	11	1

Page table

Page fault service:

1. Process that caused page fault gets blocked
2. Virtual memory manager now starts running (in kernal mode). Virtual memory manager tells disk manager about which page is missing.
3. Disk Managers communicates with device driver.
4. Device driver reads the page (through disk mechanism) and transfers required page to OS using DMA.
5. OS will try to load the page in one of the frames of user process.
6. If there is no empty frame to load new page then virtual memory manager has to initiate page replacement through which one of the pages in memory is replaced with required page. Here we have 2 cases:
   (i) page that is selected for replacement is not modified then we can directly overwrite it with new page. (1 disk operation)
   (ii) If it is modified (dirty page) then the modified page has to be saved in the disk. Later the new paged is loaded (i.e., 2 disk operations)
7. After the new page is loaded we need to update page table (Eg: changing V/I bit etc.)
8. Unblock the user process

   Now when the process is scheduled again, it generates the same VA that it had generated previously and continues its execution.

This is VM implementation through demand paging.

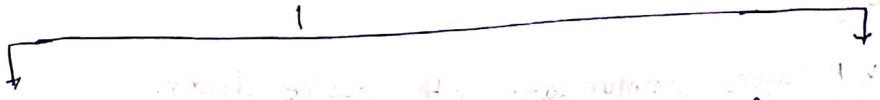
→ page fault service time is time taken to service a page fault.

~~Hence we need to reduce~~

This service time reduces throughput.

So we need to reduce page fault rate.

→ Demand paging is of 2 types:



~~Pure Demand~~

Pure Demand paging

→ Execution starts with no ~~pages~~ pages in the main memory

Prefetched Demand paging

→ Execution starts with loading a few pages

(By default we consider pure demand paging while solving questions)

(Q32) The size of virtual memory supported is eventually limited by size _____

Ans:

size of disk (secondary storage)

~~(Q33) What~~

Note:

For real time implementation of virtual memory

PAS ≤ VAS ≤ Disk AS

Performance of Virtual Memory

(i) Temporal Issue: (time)

Assume

main memory access time = 'm'

page fault service time = 's' (s >>> m)

page fault rate = 'p' ($0 \le p < 1$)

page hit rate = 1-p

> EMAT with Demand paging = $(1-p)m + p*s$

(∵ s >>> m we don't take s+m)

In general pg. fault service time may range in milliseconds.

(H4/16)

PFST = 10ms

Mem. AT = 1μs

page hit rate, $P = \frac{99.99}{100} = 0.9999$

$$EMAT = \frac{99.99}{100}(1) + \frac{0.01}{100}(10 \times 10^3) \ \mu s$$

$$= \frac{99.99}{100} + \cdots$$

$$= \frac{99.99}{100} + 1$$

$$= 1.9999 \ \mu s \cong 2 \mu s$$

(H4/17)

page fault rate = $\frac{1}{k}$

page hit rate = $\frac{k-1}{k}$

Effective instruction time = $\frac{k-1}{k} i + \frac{1}{k}(i+j) = \frac{(k-1)i}{k} + \frac{i}{k} + \frac{j}{k}$

$$= i + \frac{j}{k}$$

H4/18

PFST = 8 ms (empty frame or unmodified frame)

= 20 ms (modified frame)

Mem. AT = 100 ns

p page to be replace is modified 70% of the time.

required EMAT ≤ 2000 ns

Let p be page fault rate

$EMAT \geq (1-P)(M) + P(0.7(20) + 0.3(8))$

$= (1-P)(100) + P(14 + 2.4)$

$= 100 - 100P + 16.4P$

$\Rightarrow 100 - 83.6P \leq 2000$ ns

$EMAT = (1-P)(100) + P(0.7(20\times10^6) + 0.3(8\times10^6))$

$= 100 - 100P + P(14\times10^6 + 2.4\times10^6)$

$= 100 - 100P + 16.4P\times10^6$

$\Rightarrow (164\times10^5 - 100)P \leq 2000 - 100$

$\Rightarrow (164000 - 1)(100P) \leq 1900$

$P \leq \frac{19}{163999}$

max acceptable page fault rate = $\frac{19}{163999}$

H4/19

Mem. AT = M (page hit)

= D (page fault)

for some process . EMAT = X units.

page fault rate, P = ?

$$X = (1-P)M + P(D)$$

$$X = M - MP + PD$$

$$\Rightarrow P(D-M) = X - M$$

$$P = \frac{X-M}{D-M}$$

Note:

Considering TLB & page fault rate the effective memory access time is

$$EMAT = x(c+m) + (1-x)\left[(1-P) * m + P * s\right] + c$$

↳ (negligible)

x - hit ratio of TLB

p - page fault rate

(ii) page replacement policies:

Page reference string or reference string:

* set of successively unique pages refered in the given list of virtual addresses.

Eg: VAs: ⟨702; 755; 864; 084; 122; 360; 768; 934; 015; 125; 654⟩

Let page size = 100

Also all the above addresses are in ~~dec~~ decimal

page P_0 contains addresses 0 to 99

P_1 " " 100 to 199

⋮

page number = $\left\lfloor \frac{VA}{PS} \right\rfloor$

page offset = VA % PS

∴ Reference string for above VA's is

⟨7,8,0,1,5,7,9,0,1,6⟩

(successive page numbers are unique)

Every reference string has 2 parameters:

(i) length (i.e., 10 in above example)

(ii) no of unique pages (n) (i.e., 7 in above example)

Frame Allocation Policies

Assume we have

'n' no of process $(P_1 \dots P_n)$

demand of each process P_i be S_i no of frames

total demand for frames, $D = \sum_{i=1}^{n} S_i$

total frame available be: M $(D > M)$

frames allocated to process P_i be a_i

Eg: Let $n=5$, $M=50$

	S_i	a_i ①	a_i ②
P_1	6	10	3
P_2	20	10	10
P_3	44	10	22
P_4	18	10	9
P_5	12	10	6
D:	100		

① Equal allocation policy

Every process gets equal no of frame

$\therefore \frac{50}{5} = 10$ frames

② Proportional allocation

no of frames allocated is proportional to size

i.e., $\frac{S_i}{D} \times M$ no of frames for process P_i

③ x% rule

Every process is given x% of the frames it has demanded.

This x is choosen based on main memory size and other factors.

Page Replacement Techniques (pure demand paging by default)

Consider for a process

reference string I: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Length of reference string l = 20

no of unique pages, n = 6

1) FIFO:

Assume no of frames allocate for the process = 3

based on the time we loaded we replace

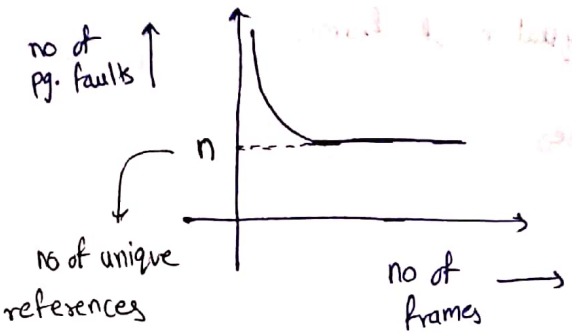
7	7	7	2	2	2	4	4	4	0	0	0	3	3	3	2	2	2	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0	0	1	1	1	0
		1	1	1	0	0	0	3	3	3	2	2	2	1	1	1	7	7	7

} frames

from above figure we have 15 page faults

$\therefore$ page fault rate $= \frac{15}{20} = 0.75$

If we increase no of frame allocate page faults decreases



If no of frames allocated = 4

7	0	1	2	0	3	0	4	2	3	0	3
	0	1	2	0	3	0	4	2	3	0	3
		1	2	0	3	0	4	2	3	0	3
			2	0	3	0	4	2	3	0	3

∴ 10 page faults

page fault rate = $\frac{10}{20} = 0.5$

Ref-String II : ⟨1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5⟩

l = 12

n = 5 (uniq. ref)

FIFO

no of frame allocate = 3

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3

∴ 9 page faults

Pg fault rate = $\frac{9}{12} = 0.75$

if no of frames allocated = 4

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3
			1	2	3	4	1	2	5	1	2

∴ page faults = 10

page fault rate = $\frac{10}{12} = 0.83$

In this example we got more page faults with more no of frames which is an anamoly.

This is known as Belady's anamoly

↳ with increase in no of frames to the process, the page fault rate increases

→ Beladys Anamoly is observed only in FIFO & FIFO based replacement.

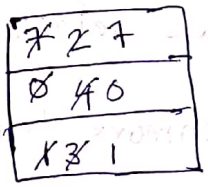
19/08/20

2) Optimal replacement:

Optimal replacement says that replace that page which will not be used for longest duration of time in ~~future~~ future references.

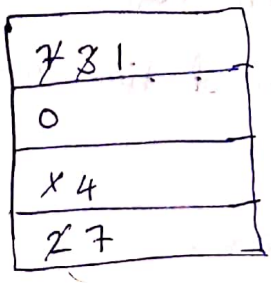
ref string: < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

- 3 frames:



∴ 9 page faults

4 frames:



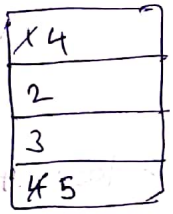
∴ 8 page faults

Ref string : <1,2,3,4,1,2,5,1,2,3,4,5>

3 frames :

∴ page faults = 7

4 frames :



∴ page faults = 6

→ Optimal replacement ~~gener~~ has the least page fault rate.

Drawback :

→ Optimal page replacement is not practically implementable since we can never know the future references.

→ However this can be used as benchmark to measure the performance of other algorithms.

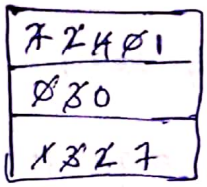
3) Least Recently Used (LRU) :

→ LRU replaces that page which hasn't been refered for the longest duration of time in the past.

∴ selection criteria : Time of reference

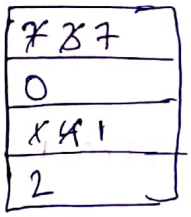
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 12

4 frames:



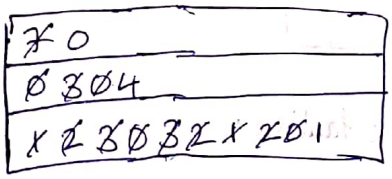
∴ no of page faults = 8

4) Most Recently Used (MRU)

* Most recently used page is replaced

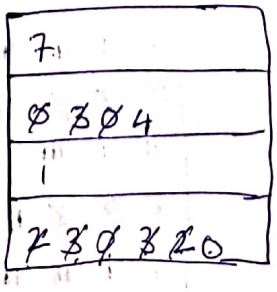
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 16

4 frames:



∴ no of page faults = 12

5) Counting Algorithms

(Least Frequently Used) (Most Frequently Used)

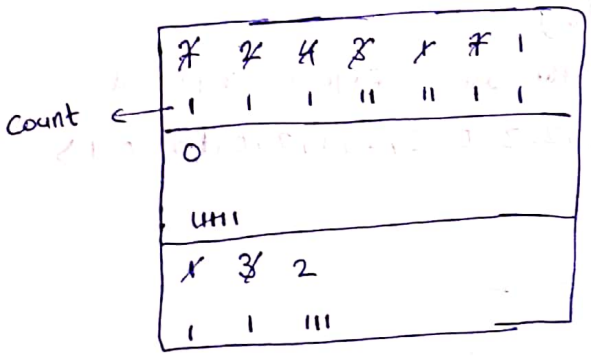
* Frequency means no of times the page is ~~refren~~ refered since it has been brought into memory. (i.e., if we load a page that has been loaded previously we start the count from 0)
* FIFO may be used ~~for~~ to resolve a conflict

3 ~~frames~~ :

Ref string : < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

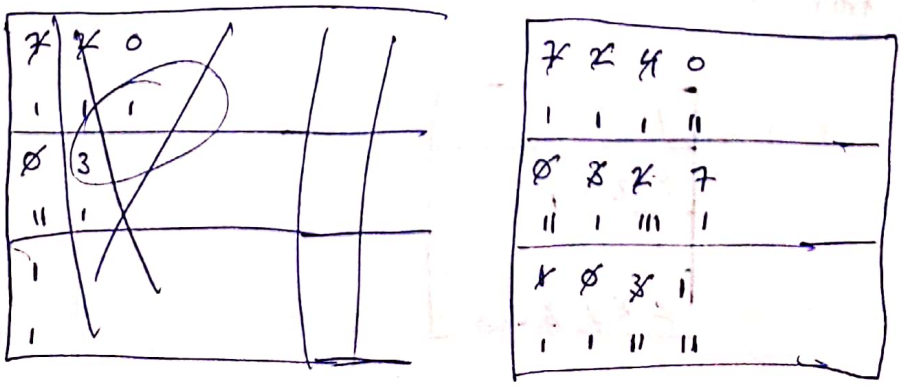
3 frames

LFU :



no of page faults = 11

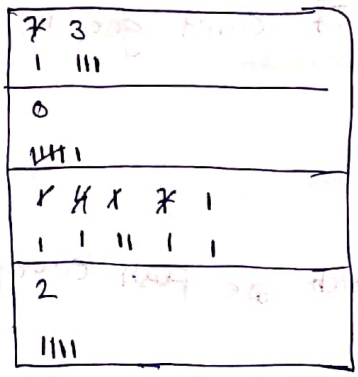
MFU :



∴ no of page faults = 12

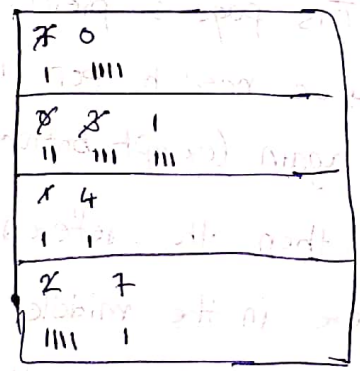
4 frames :

LFU :



no of page faults = 9

MFU :



no of page faults = 9

Note :

→ Practically LRU has been found to give less page faults (i.e., best performance & close to optimal)

Implementation of LRU :

* Here we need to monitor time of reference (TOR)

There are two methods to monitor time of reference

i) Counter method :

we maintain a counter register which is incremented at every clock pulse. where whenever we refer a page we assign the value of counter as time of reference for that page. Initial value of counter is 0.

Limitation :

* An n-bit counter can count only 2^n values after which count starts from 0. If count goes to '0' then the algorithm fails.

(ii) Stack method

→ Here we use a stack into which we push every new page we replace.

→ At the time of replacement, we need to replace with the least recently used page. This page is present at the bottom of the stack. Thus we need to pop out all the element and push them again (except bottom element)

→ If a page hit occurs, then the refered place might be present somewhere in the middle of the stack. This element has to be onto to the top of the stack for which we need to perform push & pop operations again.

Advantage :

* Unlike counter method, it works good in any case.

Disadvantage :

* push & pop operation requires more time.

LRU Approximations :

* LRU approximations are the class of replacement techniques that pretend to work like LRU. i.e., they approximate to the behaviour of LRU.

we have 4 types of LRU approximations

i) Reference bit (R):

R < 0 : page not refered so far during present epoch
R < 1 : page is refered ~~so far~~ atleast once so far during present epoch.

Consider below page table

Pg No	frame No.	V/I	TOL	R
0	a	1	2	1
1	b	1	4	0
2	–	0	–	–
3	c	1	0	1
4	d	1	3	1
5	e	1	1	0

while implementing this algorithm we may not have TOL column

(TOL - Time of ~~reference~~ loading)

(Here in the Pg table Pg P_1 & P_5 are not refered in the present epoch)

* Time is divided into equal interval and each interval is called an epoch.

* when an epoch is finished all the 'R' values are cleared.

Algorithm:

→ * If page hit occurs, then set 'R' of the corresponding page table entry.

→ If page fault occurs,

Start searching ~~for R=0~~ from the first entry for R=0 are replace the first encountered entry's corresponding page.

→ If 'R' value is one for all the page then the algorithm fails.

* To overcome this limitation we use next LRU approximation which is additional reference bits.

~~(ii) Additional~~

(ii) Additional Reference bits:

Here we have more than one reference bits, say 'n' indicating past 'n' references of past 'n' epochs

For example take n=8

	R_1 R_2 ..... R_7 R_8
P_i :	10011111
P_j :	01101111
R_k :	10111111

* Least significant bit corresponds to current epoch
* Most significant bit corresponds to most previous epoch

For above example if a page fault occurs P_j is replaced.

(∵ the 4 least significants of the 3 processes are '1'

however R_4 of P_j is 0 and R_4 of P_i & P_k are 1

∴ Replace P_j)

* After every epoch we need to left shift the reference bits by one bit.

* However this method also has a chance to fail when all the reference bits of all the processes are 1's. Anyway the probability for this is very less.

(iii) Second Chance (clock Algorithm)

Selection criteria: (Time of loding & Reference bit)

→ The page with R value as 0 and least TOL (early loaded page among R=0 pages) is replaced.

The pages whose TOL is less than replaced page, their R value is set to '0'.

→ when all R values are 1's, the earliest loaded page is replaced.

i.e., The algorithm, when all R values are 1, degenerates to FIFO

* Hence this algorithm (FIFO based) also suffers from Belady's anomaly.

(iv) Enhanced Second chance / Not recently used (NRU):

Selection criteria: (Reference bit (R) & modified bit (M))

Modified bit (or) Dirty bit
- 0: page is not modified since its loading
- 1: page is modified since the time of its loading

R	M	
0	0	(I)
0	1	(II)
1	0	(III)
1	1	(IV)

← this means the page is not refered in this epoch but is refered and modified in some previous epoch

* If a page fault occurs

then page with RM = 00 is replaced

if no such page RM = 01 is replaced

if no such page RM = 10 is replaced

if no such page RM = 11 is replaced

* The above search starts from the first entry.

(14/20)

length of ref string: L

no of unique pages: k

allocated frames: Z

* In the worst case every reference can be a page fault

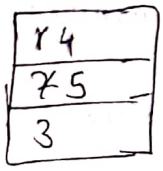
i.e., L

* Since it is pure demand page we must have atleast k page faults

~~In the best case we can have all page hits after loading the servicing the last page fault~~

Eg: Let z=3, k=5

Ref string: 1, 2, 3, 4, 5, 4, 5



∴ k

In the best case the above could happen.

H4/21

Ref ~~string~~: $P_1, P_2, P_3, P_4 \cdots P_{n-1}, P_n, P_n, P_{n-1}, \cdots P_3, P_2, P_1$

↳ (This is not reference string since we have Pn twice successively)

After refering till Pn we will have

$P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)}$ in the frames

∴ The next k references $(P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)})$ will be hits

total references = 2n

no of hits = k

∴ no of page faults = 2n-k

H4/22 Ref string: ~~1, 2, 4, 5, 1, 2, 2~~, 2,

1, 2, 4, 5, 1, 2, 3.

∴ length of refstring = 7

no of frames allocated = 1

∴ 7 page faults

(H4/23) VAS = PAS = 2^{16} bytes = 64 kB

PS = 512 bytes = 2^9 bytes

PTES = 4 bytes = 32 bits

no of frames = $\frac{2^{16}}{2^9} = 2^7$

no of bits to store frame number = 7
V/I = 1
R = 1
Modified = 1
Page Protection = 3
} 13 bits

$\therefore$ 32 - 13 = 19 bits

19 bits can be used to store other information

no of pages = $\frac{2^{16}}{2^9} = 2^7$

size of page table = $2^7 \times 4 = 2^9$ = 512 bytes

Thrashing:

* Excessive [high paging activity
+
act of servicing page fault

i.e., High page fault rate

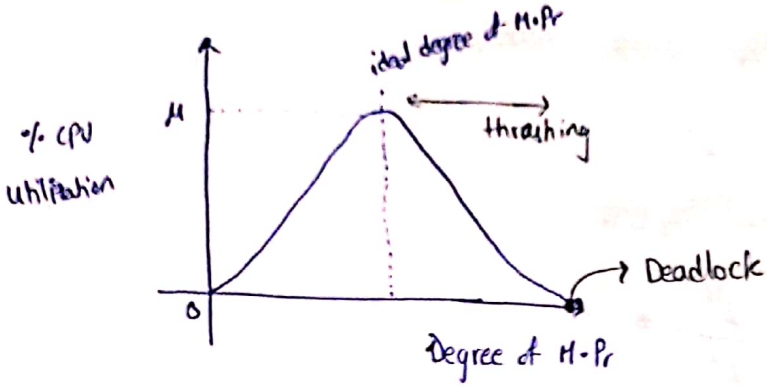
* Thrashing is undesirable

* Thrashing cause more processes to block

Cause of Thrashing

→ Lack of frame (i.e., Lack of memory)
→ High degree of multiprogramming
} Primary reasons for thrashing

(We also have some secondary reasons which we will explore later)



Thrashing Control strategies:

Prevention

* Prevention is done by controlling degree of multiprogramming. (controlled by long term scheduler)

Detect & Recover

Detection : * High degree M.Pr & low CPU utilization
=> thrashing
* High paging (high disk utilization)

The above 3 conditions confirms the presence of thrashing

Recovery: Decrease the degree of M.Pr
(This done by suspending the processes)

Secondary reasons for thrashing:

→ Page replacement policy

→ page size:

page size must be large to reduce page faults.
It is because with large PS we have less pages.
(Large PS => less pages => less page faults)

→ Programming techniques & Data structures used by programmers. This fact is explained through below case studies.

Case study I:

```
integer A[1...128][1...128];
```

1) for i←1 to 128
       for j←1 to 128
           A[j][i]=1;

2) for i←1 to 128
       for j←1 to 128
           A[i][j]=1;

Assume we are store the array in row major order

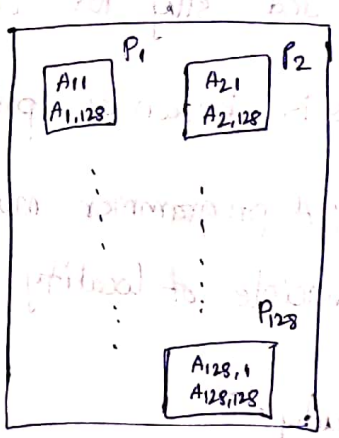
also let page size = 128 words

and each element of array is one word

⇒ each row is stored in one page exactly

Assume we are using pure demand paging and replacement technique FIFO.

→ Assume the OS has allocated 127 frames to the process.



page faults with 1st program:

→ A[0][0], A[1][1] .... A[128][1]

Here every reference cause a page fault

i.e. 128 page faults.

Sily for 2nd column we get 128 page faults

In this way

total no of page faults = 128 × 128

Page faults for 2nd program :

→ A[1][1], A[1][2] ... A[1][128]

A[1][1] : page fault

A[1][2] ... A[1][128] : page hit

∴ 1st row causes one page fault

Silly we get 1 page fault for 2nd row

∴ total no of page faults = 128

→ ∴ 2nd program is 128 times faster than that of 1st program

Note:

* If we use row major order to store 2d array then it is better to access in that order
  - and silly for column major order.

  This is known as principle of locality

* ∴ A programmer must use data structures that ensure principle of locality.

Case Study - II

* If we consider array & linked list,

  array allocation is contiguous.

  linked list allocation is non-contiguous.

  ∴ linked list may cause more page faults.

* The elements of array obeys principle of locality.

Case Study III

Linear Search vs Binary Search

→ In demand paging, a binary search causes more page faults because we don't access elements contiguously.
whereas in linear search we search in contiguous way causing less page faults and obeying principle of locality.

Case Study - IV

Stack:

Stack operation which are push & pop are performed at the same end.
∴ less page faults

Hashing:

Hashing distributes key across the memory and is more likely to cause more page faults.

Note:

→ In general a Data Structure or a programming technique in demand paging is said to be good iff it obeys the concept of locality of reference.

Working Set Strategy

* The objective of working set strategy is to control thrashing and utilize memory effectively.
* Working set strategy is based on principle of locality of reference.

Consider below program

```
main()//10kB        f()//5kB        g()//30kB       h()//3kB
{                   {               {               {
    :                   :               :               :
    f();                g();            h();            printf();//25kB
    :                   :               :               :
}                   }               }               }
```

Total program size = 73 kB

Assume page size = 1 kB

Assume the process has been allocated 35 frames

* This is static allocation.

→ As the process executes it changes from one locality to other locality (i.e., from one function to other function). When process is in main 10 frames would be enough. When it is executing in f(), 5 frames would be enough.

→ Thus allocating frames based on the locality in which the process is executing would reduce thrashing and utilizes memory more effectively.

→ To meet this requirement we use dynamic frame allocation technique.

* The working set strategy's working is based on the locality of reference.

It estimates the locality in which the process is running and no of frames are allocated accordingly.

* Estimating the size of locality:

Consider the reference string for process P_i

$\Delta = 10$

P_i : ⟨ 7, 0, 1, 2, 3, 0, 2, 16, 18, 22, 18, 16, 35, 38, 39, 34, 35, 37, 38, 36, 34, 36, 37 ⟩

Assume these references are generated in time t. After time t, we need to know the locality in which the process is executing and size of locality.

* observing the patterns we can know localities & size of localities.

* to estimate the size of locality we define a parameter called working set window (wsw) at time t for a process

Working set window$_i^t$ (wsw) = { set of unique pages refered in the past Δ references }

where Δ is an integer.

For example take Δ = 10

⇒ wsw_i^t = { 34, 35, 36, 37, 38, 39 }

⇒ $wsw\text{-}size_i^t = 6$

↓

size of locality is 6 pages

∴ 6 is the demand

Framework:

→ Let 'n' be no of processes

→ s_i : Demand for frames by process P_i (Estimated as shown in the previously example) (i.e., wsw size$_i$)

→ total demand, $D = \sum_{i=1}^{n} s_i = \sum_{i=1}^{n} WSWS_i$

→ Available frames = M

Case (i):

$D \cong M$

i.e., Balanced & no thrashing

Case (ii)

$D < M$

i.e., No thrashing & we can further increase deg of M.pr

Case (iii)

$D > M$

i.e., Thrashing

∴ Working sets strategy helps control strategy and utilize memory better.

Note:

* The success of Working Set Strategy depends on the value of Δ.

* If Δ is too small ⇒ more page faults.

* If Δ is too large

⇒ we also hold pages of previous locality and this is ineffective use of memory

(H4/24)

page ref : c c d b c e c e a d

$\Delta = 4$

$WST_0 = \{a, d, e\}$

0

i.e., $\{e, d, a\}$ (-2, -1, 0)

∴ ref string : e d a c c d b c e c e a d

-2 -1 0 1 2 3 4 5 6 7 8 9 10

• • • • • ⇒ 5 page faults

t_0 : <e d a> : 3

t_1 : <e d a c> : 4 → At time t_1 'c' is refered since wsw at t_0 has c it causes page fault

t_2 : <d a c> : 3

t_3 : <a c d> : 3

t_4 : <c d b> : 3 → Slly at time t_4 'b' is refered since 'b' is not present in wsw at t_3 a page fault is generated

t_5 : <d b c> : 3

t_6 : <d b c e> : 4

t_7 : <b e c> : 3

t_8 : <c e> : 2

t_9 : <c e a> : 3

t_{10} : <c e a d> : 4

Avg no of frames needed = $\frac{32}{10}$ = 3.2 frames