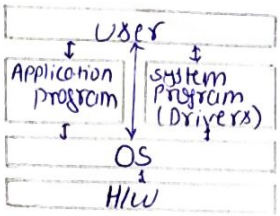
OS

1) What is OS?
- interface b/w user & H/w
- Set of utilities



2) Services :-
- Program execution
- CPU scheduling
- Memory, file, resource management
- Error detection
- Accounting • Security • I/O operations
- Inter process communication.

3) Goals:
- convenience • Efficient • Reliable
- Scalable • Portable • Robust (handle unseen error)

⇒ Types of OS:

1) Uniprogramming OS: only 1 program can reside in MM at a time.
- Disadv: inefficient utilization of resources

2) Multiprogramming OS: allows more than 1 process in MM at a time.
- If we increase degree of multiprog., CPU utilization increases but upto a certain limit.

* Degree of Multiprogramming: NO of processes in MM at a time.
- Types :- Preemptive, NON-Preemptive (or non-pre)
- NON-pre, process leaves CPU if it completes execution or goes for IO opn

NOTE: When system starts, booting process is done to bring OS from hard disk to MM. (bootstrap program runs)

3) Multitasking OS: (Time sharing OS)
Multiprogramming + preemptive round robin scheduling

4) Multiprocessing OS: required on system with multiple processors.
- Types: Shared memory system (Tightly coupled)
  Distributed system (Loosely coupled)

5) Multiuser OS: All users can execute their separate processes together on a system having single CPU.

NOTE: Microsoft OS are not multiuser whereas linux OS are ".

6) Real time OS:
OS gives deadline to processes.
- Types: Soft Realtime, Hard Realtime

7) Embedded system OS: used to communicate b/w mechanical machine & processor.

8) Handheld devices OS: of mobiles.

* System call: way for appl. programs or users to interact with OS. (invoked using s/w interrupt)

1) Parts of OS:
- Kernel: responsible for all functionalities
- Shell: used to interact with kernel.

2) In windows we have 3 shells :-
CMD, powershell, GUI

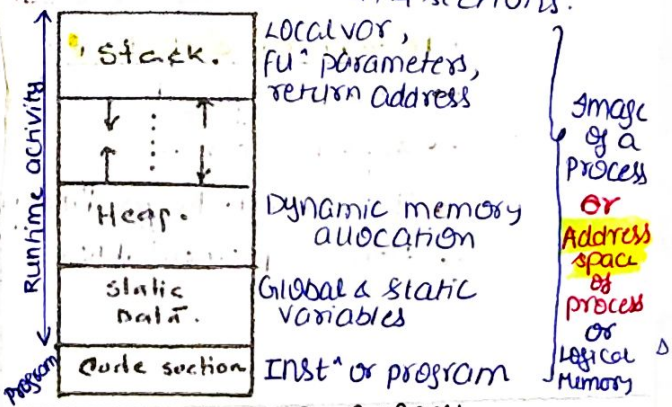
3) Dual Mode of Operation: (for protection)
- User mode (mode bit 1): appl. process is running & application process can perform only non-privileged operation
- Kernel/system/supervisor mode: (mode bit 0): to-do privileged opr- appl. process makes system call. (s/w interrupt)

1) Program is binary file stored in disk.

2) Running program is called Process
- unit of execution (CPU)
- Locus of control (OS)
- Schedulable/dispatchable unit

3) Process as a data structure:
- Definition : program
- Representation/Implementation
- Operations • Attributes

4) Process Representation: * Every process is stored in memory in 4 sections.



5) Operations of a process:-
- Create • Schedule • Wait
- Suspend, resume • Terminate

6) TO control process, OS maintains 1 PCB for each process. PCBs are stored under OS area in MM & processes cannot access it.

7) Each process has a unique process ID.

8) Process control block (PCB): collection of attributes of process (PID, PC, GPR's, List of files, devices, type, size, Memory limit, priority, state, etc)
- Also called process descriptor.

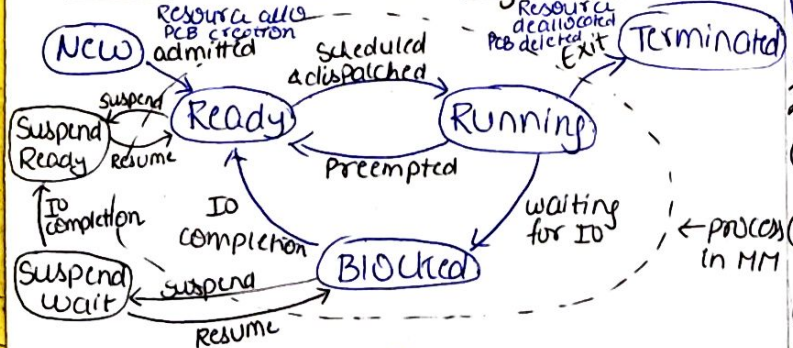
9) The content of the PCB at the moment is called context of that process.

10) Context switching: saving CPU register values into PCB of current running process & loading other process context into CPU registers.
- Program of OS which does context switching is called Dispatcher.

* For device queue of disk, scheduler is required, for all other IO devices scheduling is FCFS.
* If CPU utilization is less, long term sch. increases degree of multiprogramming.
* Swapped out process is still a process as PCB is present in MM.
* PCB is updated only by context switching
* If parent process is killed by some other process, its child process will also terminate
* The state of a process is defined by the current activity of the process.

NOTE: These formulaes are with assumption that process does not require IO.

10) State transition Diagram:-



11) Running → Terminated
Running → waiting
} voluntary transition done by process with
Rest all transition are done by OS.

12) If a process is killed forcefully, transition may take from Ready/waiting to terminated.

NOTE: while running also process is in MM only context is with CPU.

13) Process → CPU bound (CPU extensive)
→ IO bound (IO extensive)
- For better resource utilization, OS maintains a good mixture of both and IO bound process is scheduled before CPU bound process.

Process scheduling:

1) Queues:
- Job queue (New state)
- Ready queue (Ready state)
- Device Q (each IO device has queue of PCB)

2) Types of schedulers:-
a) LTS: New → Ready (Job scheduler)
- can increase degree of multiprograming
b) STS: Ready → Running (CPU scheduler)
- does not affect degree of multiprogr.
c) MTS: swaps from MM to swap space of disk & back to MM
- PCB is still present in MM.
- can increase or decrease degree of multiprogramming.

NOTE: swap space is under OS control

3) If swapping is done based on priority of process it's called Rolling

4) A process either in ready or waiting state can be swapped

5) Zombie process: if PCB of terminated process remains in MM. or content

CPU Scheduling: (short term sch)

1) Goal:-
- Minimize WT & TAT
- Maximize CPU utilization & throughput
- NO starvation

2) Scheduling Times:
(a) Arival time:- time at which process is admitted by long term scheduler.
(b) Burst/Service time:- time required for execution of a process.
(c) Completion time
(d) Turn around time:- Total time from arrival to completion.
TAT = CT - AT = BT + WT
(e) Waiting time
WT = TAT - BT
(f) Response time:- time from arrival to first response by CPU

3) Scheduling Length:
$L = \max(CT) - \min(AT)$

4) Throughput - NO of process executed per unit time. $T = \frac{n}{L}$

5) In non-preemptive, WT = RT (if no IO ops.)

Scheduling Algo :- (Gantt chart always start from 0)

1) FCFS : Criteria : AT Tie breaker : PID
Mode : Non-preem.
- Only FCFS suffers from convoy effect
- No starvation

*Convoy effect : large process is scheduled first (among process having small process) so it will make other processes to wait lot.

2) SJF : criteria : min BT Tie breaker : FCFS
Mode : Non-preem.
- Min avg WT among non-preem.

*Longest job first may suffer from convey effect.
*Starvation : If new processes keeps coming & executing and old process vation keeps waiting for indefinite time.
*Generally throughput
Other algo ≤ SJF ≤ SRTF

NOTE : If the process running is scheduled again, then context is for switch is not required. (required only in RR.)

NOTE : Optimal preem. algo → SRTF
Optimal non-preem. algo → SJF

3) SRTF : Criteria : min remaining BT
Tie breaker : FCFS , Mode : Preemptive
- Min avg WT among all scheduling algo
- Better throughput, no convoy effect in continuous run
- Starvation

NOTE : No practical implementation of SJF & SRTF as BT is not preknown.

NOTE : Min no. of context switches in preemp execution = (n-1)

* If all process arrive at same time, SRTF reduces to SJF.

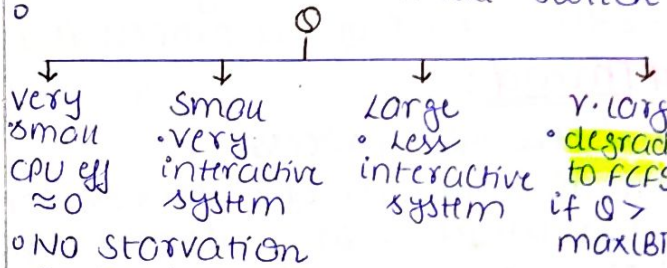
4) HRRN : favours shorter job also reduces WT of longer job.
Criteria : Response ratio $\left(\frac{WT+BT}{BT}\right)$ α WT α 1/BT
Tie breaker : SJF
Mode : Non-preemptive
- NO starvation

5) Priority based :
Criteria : Priority
Tie breaker : given in ques
Mode : Preemp / Non-preem.
- Better RT for real time situation.
- Starvation for low priority process.
- Solution of starvation : Aging (possible only with dynamic priority)
- Aging : After waiting for specific time, priority is increased.

6) Round Robin : (Time sharing)
- Criteria : AT + Q ; Q → time quantum
- Mode : preemptive
- No. of context switch = $\left(\sum\left\lceil\frac{BT}{Q}\right\rceil\right) - 1$

NOTE :

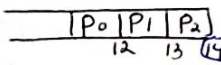
In RR, it will be counted as 2 context switch.



- No starvation
- Better interactiveness
- Best Response time. (min)
- Avg WT & TAT is more

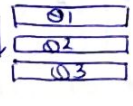
NOTE : Throughput & idle period is same with all algorithm.

7) LRTF : Tie breaker : PID Preemptive
- If all process have arrived before completion of any process then termination is in order of PID.



8) Multilevel Queue scheduling :
- using different algo for different queues based on requirement

→ Fixed priority preemptive sch :-
- All queues have different sch algo may be preemptive or non-preemptive.
- Process can't move from 1 Queue to other
- Starvation
- Preemption will happen with arrival of a process in higher priority queue.



→ Time Slicing : CPU time is divided among multiple queues.

9) Multilevel Feedback Queue Sch :
- Process may change queue, flexible
- Starvation for low priority queue

* If IO is also there, scheduling is like CPU, IO, CPU. WT = TAT - (CPU + IO + CPU)

* For n process, CPU utilization = $1 - p^n$
  p → fraction of time for IO
* If there is limited IO, process will execute IO opr" in FCFS.

→ All preemptive may behave as non preempt-
- only 1 process in MM
- all process arrived together.
- SRTF - later arriving process is longer
- Priority - " " " has low prior.
- RR - if Q is very large
- LRTF - if BT of all process is 1

#Thread:

1) Lightweight process
2) Multiple threads of a process can run parallely in multicore system.

* Shared among thread	Unique for each thread
Code section	Stack
Data section	Thread id
Heap • Page table (PTBR)	Register set
OS Resources	Program counter
Open files	
Address space	

3) Thread → user level thread (not known to Kernel)
   → Kernel level thread
4) In user level threading, process itself does context switching b/w threads.

* User Thread	Kernel Thread
• Multithreading in user process.	• In kernel process
• Created without Kernel intervention.	• Kernel itself does multithreading
• context switch faster.	• Slower
• If 1 thread is blocked, OS blocks entire process	• Individual threads can be blocked.
• Generic & can run on any OS (implemented by libraries) (HW independent)	• Specific to OS.
• Faster to create and manage.	• Slower to create & manage.
• created using fork call	• using system call

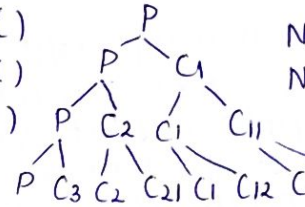
5) Thread context switching is faster than process context switching.

1) System call → requesting service from Kernel via Software Interrupt.
- Sw interrupt generated when further inst can't be executed.

NOTE: Sw interrupt known as 'Trap to OS' (Generally interrupt is given to CPU)

2) NO system call is made to use stack or heap space.
3) Child process will start from next stmt of fork().
4) Parent & child can run concurrently (not parallely).
5) Fork takes no parameter & returns an integer.
- -ve value: if creation of child process is unsuccessful.
- Zero: returned to child process.
- +ve value: PID of child process is returned to parent process

6) fork()
   fork()
   fork()



NO of Process = 2^n
NO of child process = $2^n - 1$

7) fork creates child not thread. Child gets copy of all things of parent.
8) Unless all child processes are terminates, parent can't terminate.
9) By default consider child is created successfully.

* Each Kernel level thread has its own PCB pointing to same address space.
* Multithreading Model:
  1:1 (user level), Many:1, Many:Many → Kernel level thread
* System call categories:
- Process control
- File management
- device "
- Security

* fork: system call to create new process in Linux & UNIX OS.
* if(fork()) { } else { } Parent process will execute if, child will execute else.
* concurrently: one after other (time sharing)
  parallely, simultaneously: एक साथ

NOTE: printf("Hi")
fork() fork()
Hi Hi Hi Hi

printf("Hi\n")
fork() fork()
Hi

Printf is buffered, buffer flushes only on using \n or flush(), without flushing buffer is passed to child processes.

Process Synchronization

1) Process types: Independent process, cooperating / coordinating / communicating
2) Synchronization needed b/w cooperating process, not independent process.
3) Problems without synchronization: Inconsistency, Data loss, deadlock.
4) Sync is required for only that part of process which is involved in communication ie critical section
5) communication b/w process can be by:
   - Message passing
   - sharing some common resource → H/w, S/w
6) Critical section: code section where shared variable can be accessed.
7) OS has to provide sync to CS.
8) Race condition: when final result of concurrent processes depends on the sequence in which the processes complete their execution.
9) When there is mutual exclusion, race condition is not there.
10) If 2 processes P1 & P2 r executing concurrently, they can be executed as
   - P1 then P2
   - P2 then P1
   - Concurrently & P2 finishes 1st
   - " & P1 finishes 1st

⇒ Critical Section Problem:

1) If there is CS, OS needs to provide sync. This is CS problem.
2) A perfect solution to CS problem must satisfy all 3 requirements: Mutual exclusion, Progress, bounded waiting.

NOTE: Concurrent: executing 1 after other
Parallel: executing simultaneously

3) • Mutual exclusion: At a time only 1 process can enter into CS.
   • Progress: If noone is in CS & some one wants to go, let it go in CS.
   • Bounded waiting: A process should not repeatedly enter CS keeping other processes waiting.
4) How to check these 3 in question:-

@ Mutual exclusion:
- 1 process in CS, another tries to enter → show that 2nd process will get blocked in entry code
- 2 or more processes are in entry code → atmost 1 will enter CS.

(b) Progress: (no deadlock)
- NO process in CS, P arrives → P enters.
- 2 or more processes in entry code → show that atleast 1 will enter CS (no deadlock)

© Bounded waiting:
- 1 process in CS, another process waiting to enter → show that first process after exiting CS cannot reenter CS; waiting process will get in.

5) Progress ⇏ Bounded waiting
Ex: Process P1 repeatedly enters CS. Process is there but not BW.
Bounded waiting ⇏ Progress
Ex: All processes are in deadlock in entry section. BW is there but not progress.

6) Progress + Bounded waiting → NO starvation
7) Progress & Bounded waiting are independent of each other.
8) Bounded waiting & deadlock are not related to each other.
   Deadlock → may or may not be BW
   BW → may or may not be deadlock
Ex: deadlock but no BW
Deny entry of P1, always allow P2, on first entry of P3 deny entry of everyone.

9) Process
```
{
   Entry section   ← code for synchronization
      CS
   Exit section    ←
   Remainder section
}
```

10) while(true) ⇒ while(TRUE); infinite loop

11) Preemption can happen anytime.

12) Progress → NO deadlock
   deadlock → NO progress

NOTE: If interrupts are disabled, process can't have context switch ie preemption not possible. It's a H/w sol^n of sync. only possible with single processor →

⇒ 2-process sol^n : (S/w solutions)

1) Sol1: Lock variable sol^n :

```
Lock = False
        Po                          |        P1
while (True)                        | while (True)
{                                   | {
  while (lock);   ← Busy waiting    |   while (lock);
  lock = True;                      |   lock = True;
    CS                              |     CS
  lock = False;                     |   lock = False;
}                                   | }
```

Process can execute any no. of times.

ME ✗ Progress ✓ BW ✗

- Starvation possible : 1 process keeps running again & again, other starves.
- NO deadlock
- Busy waiting

2) Sol2: Turn variable sol^n :

```
turn = 0;
        Po                          |        P1
while(true)                         | while(True)
{ while (turn!=0);                  | { while (turn!=1);
    CS                              |     CS
  turn = 1;                         |   turn = 0;
}                                   | }
```

ME ✓ Pr ✗ BW ✓

- Starvation
- NO deadlock
- Busy waiting
- Both process will execute in strict alternation.
- If initially turn=0, P1 may starve
  " " turn=1, Po " "

3) Sol3: Peterson's sol^n : Perfect sol^n for 2 process

```
want[2] = {False, False}
turn ;
        Po                          |        P1
while (True)                        | while (True)
{ want[0] = True                    | { want[1] = True
  turn = 1                          |   turn = 0
  while (want[1] && turn==1);       | while (want[0] && turn==0);
    CS                              |     CS
  want[0] = False                   |   want[1] = False
}                                   | }
```

- ME ✓ Prog ✓ BW ✓
- NO deadlock
- NO starvation
- Busy waiting

*Busy Waiting :- A process gets a chance to run on CPU but keeps executing certain stmts only for waiting for CS thus wasting CPU time.

Ex: while(.....);

NOTE: Strict alternation means after Po, only P1 can execute whether P1 waits for CS or not.

⇒ Hardware sol^n for synchronization

- using hardware atomic inst^n are implemented which is used to provide synchronization.

1) Test and Set() :-

- This inst^n takes a variable, return its current value & sets it to True. ie. combining stmt 1 & 2 of Sol1.

```
Lock = False
        Po                          |        P1
while (True)                        | same code
{ while (TestAndSet (Lock));        |
    CS                              |
  Lock = False                      |
}                                   |
```

- ME ✓ Prog ✓ BW ✗
- Starvation
- NO deadlock
- Busy waiting

2) Swap() :-

- Takes 2 var & swap their value.

```
Lock = False  ← Global & shared var
Key           ← Local var to each process
        Po                          |        P1
while (True)                        | same code
{ Key = True                        |
  while (Key == True)               |
    swap (Key, Lock);               |
    CS                              |
  Lock = False                      |
}                                   |
```

- ME ✓ Prog ✓ BW ✗
- Starvation
- NO deadlock
- Busy waiting

⇒ Sync. tools provided by OS :-

- Semaphore
- Monitor — NOT in syllabus

*Semaphore :

1) It is an integer variable which can be accessed by following fun^n only :
   - Wait() / P() / Degrade() / down() : decrement by 1
   - Signal() / V() / Upgrade() / UP() : increment by 1

   These are atomic functions.

2) Types of semaphore :-
   - Binary semaphore : can take value only 0 & 1
   - counting semaphore : can take any integer value (By default TVE)

NOTE: If explicitly given in ques $x++$ is atomic then consider else consider it non atomic. by default

3) Semaphore with busy waiting:

```
Wait(S)                    Signal(S)
{ while(S<=0);             { S++;
  S--;                     }
}
```

NOTE: This implementation is perfect for binary & non-ve semaphore.

- If Binary semaphore S=0,
  Wait(S) → Busy waiting
  If S=1, Signal(S) → S remains 1

4) Critical section sol^n :-

S=1 ← Binary semaphore

P0	P1
while(True) { wait(S) CS signal(S) }	Same code

- Priority inversion may be possible

ME ✓
Prog ✓
BW ✗
Starvation ✓
No deadlock ✓

- This code will work fine with n processes.
- This is a Perfect sol^n of mutual exclusion

5) - Binary semaphore used to provide sol^n to critical section problem.
- counting semaphore used to control access to a resource with multiple instances.

VI) CS P() NO mutual exclusion if implemented properly

6) Characteristics of semaphores:-

- provide mutual exclusion if implemented properly
- sol^n may have busy waiting, deadlock, starvation, priority inversion.
- Semaphores are slw sol^n ie h/w independent.

NOTE: Deadlock: processes get stuck, can't proceed anyway (infinite waiting)
starvation: indefinite waiting.

- Deadlock is a form of starvation

Deadlock → Starvation

Q: A process wants to run 52 P() & 23 V(). we need process to stuck at 25 P(). what should be initial value of S?
Stuck at 25 P() means it executes 27 P()
∴ S-27+23=0 ⇒ S=4

7) TO run 2 processes in strict alternation min no of binary semaphore required = 2
- for 3 processes to run in specific order ⇒ 3 binary semaphore required

8) If semaphore S=2 in sol^n at point (4), max 2 processes can enter in CS together.

9) Priority inversion: high priority process stopped or preempted becoz of low priority process.
Ex: P1 is in CS, P2 (higher priority) comes, P2 will wait until P1 leaves CS
P2 will get CPU, P1 can't complete CS

10) Semaphore without busy waiting:

```
Wait(S)                          Signal(S)
{ if(--S< 0)                     { if(++S <=0)
    block(); ← Ready to blocked state   unblock(); → blocked to ready state
}                                }
```

- works perfectly for binary & non -ve semaphore.
- sol^n to critical section prblm same as in point 4.
- when a process is blocked, its added to a queue. The order in which processes are blocked, they will execute CS in same order.
- ME ✓ Prog ✓ BW ✓
- NO starvation; If we use stack or priority queue then starvation possible.
- NO deadlock
- Priority inversion may be possible

⇒ Mutex lock (Spinlock):

1) Exactly same as binary semaphore, used to provide mutual exclusion to CS thus preventing race condition
2) A process must acquire lock before entering CS & release lock when it exits CS.
3) sol^n to CS problem:

```
while(True)                      acquire(lock)
{ acquire lock                   { if(lock); ← Busy waiting
    CS                             lock=True;
  release lock                   }
}                                release(lock)
                                 { lock=False; }
```

4) It's called spinlock becoz process spins while waiting for the lock to become available.
5) If CS takes very less time, in such case spinlock is preferred becoz moving process to blocked state will take more time.

⇒ classical problems:

- There are some classical prblms which requires synchronization.
  - Producer-consumer problem
  - Reader-writer problem
  - Dining philosopher problem

(A) Producer-consumer problem: (Bounded Buffer problem)

Multiple producers → [common Buffer] → Multiple consumers

1) Buffer size is fixed.
2) Producer must be blocked if buffer full consumer must be blocked if buffer empty
3) variables: (shared variables)
- Mutex: binary semaphore to take lock on buffer for mutual exclusion.
- Full: counting semaphore to denote no. of occupied slots in buffer.
- Empty: counting semaphore to denote no. of empty slots in buffer.

4) Initialization: Mutex = 1, Full = 0, Empty = N

5)

Producer()	Consumer()
1. wait(Empty)	1. wait(Full)
Produce an item	2. wait(Mutex)
2. wait(mutex)	Remove an item from buffer
Add to buffer	3. Signal(Mutex)
3. Signal(mutex)	consume item
4. Signal(Full)	4. Signal(Empty)

6) If ques asks, if these stmts are swapped what happens → check two conditions: buffer is full & buffer is empty.
7) If stmt 1 & 2 are swapped in producer, Deadlock possible when buffer is Full.
8) If stmt 1 & 2 are swapped in consumer, Deadlock possible when buffer is Empty.
9) If stmt 1 & 2 are swapped in both producer & consumer, deadlock possible in both case when buffer is full or empty.
10) If we use 1 binary & 1 counting semaphore then also deadlock is possible

(B) Reader-Writer Problem:-

1) Many Reader → [common file] ← Many writers
2) If writer is active then all other readers & writers will be blocked.
3) If reader is accessing file, then readers are allowed but writers will be blocked
4) variables: (shared variables)
- Mutex: Binary semaphore to take lock on readcount variable for mutual exclusion
- Wrt: Binary semophore to take lock over file
- readcount: integer variable denotes no of active readers.

5) Initialization: Mutex = 1, wrt = 1, readcount = 0

6) Implementation:

```
Reader()                                   Writer()
{                                          {
1. wait(Mutex)                             1. wait(wrt)
2. readcount++                                write
3. if(readcount==1)  <- 1st reader take lock 2. Signal(wrt)
4.       wait(wrt)      on file            }
5. Signal(Mutex)
      Read
6. wait(Mutex)
7. readcount--
8. if(readcount==0)  -> last reader unlock file
9.       Signal(wrt)
10. Signal(Mutex)
}
```

7) We can not compare semaphore value that's why taken readcount as int variable not semaphore.
8) In reader(), if we remove line (5&6) or (3&8), only 1 reader will be allowed to read at a time.
9) In reader(), if stmt 2 is placed before 1, deadlock is possible. (if last reader leaves without executing stmt 8&9)
10) In reader(), if stmt 3&4 are placed after stmt 5 then its possible writers can access file while readers are reading.
11) In reader(), if stmt 8&9 are placed after stmt 10, then deadlock is possible if last reader leaves without unlocking wrt.

(C) Dining Philosopher's Problem:-

1) By default consider 5 Philosophers & 5 forks.
2) philosopher: process
chopstick: shared resource
3) K philosophers seated around a circular table. There is 1 chopstick blw each pair of philosopher. A philosopher must pick up its two adjacent chopsticks in order to eat. 1 chopstick can be picked up by any 1 of its adjacent philosopher

4) Variable: chs[K] = {1, 1, 1, ... 1}
Binary semaphore array for chopsticks.

5) Solution:

Pi:
while (True)
{
1. wait (chs[i])
2. wait (chs[(i+1) mod K])
eating
3. Signal (chs[i])
4. Signal (chs[(i+1) mod K])
}

6) Deadlock possible: if all processes execute stmt 1 one by one.
Deadlock → starvation

7) Starvation possible: some philosopher may repeatedly take forks (chopstick) leading other to starve.

8) Ways to avoid deadlock:
- Have no of philosophers ≤ (K-1) (No of chopsticks)
- A philosopher should only be allowed to pick their chopstick if both are available at same time. (Prevents hold & wait)
- Break the symmetry:
1 philosopher pick first left then right fork, while all other picks first right then left. (For 1 process swap stmt 1 & 2 in solution.
OR
odd philosophers pick first left then right, while even philosophers pick first right then left.

9) Min no of forks required to avoid deadlock with 5 philosophers = 6

⇒ Deadlock :-

P(A) ↔ P(B)
P(B) ↔ P(A)

1) Deadlock: If 2 or more processes are waiting for such an event which is not going to occur ever.
2) Deadlock occur if we do not have good synchronization solution.
3) Necessary conditions for deadlock:-
- Mutual Exclusion: Resource can be used by only 1 process at a time.
- Hold & wait: holding some resource & waiting for some other resource.
- Circular wait: processes waiting for resources in circle.
- No preemption of resource: process can't forcibly take resource from some other process.

4) These are necessary conditions for deadlock, not sufficient.
- Deadlock → All 4 condition satisfies
- Not having any condition → deadlock will never occur
- Having all 4 conditions → deadlock may or may not occur.

5) If all processes needs at most 1 resource each, deadlock cannot occur as hold & wait can't occur.

6) Resource allocation graph:-
- Process (P)
- Resource [R]
- Resource with multiple instances [...] R
- From process to resource: Request edge
- From resource to process: (instance of) Allocation edge

NOTE: Process requests for resource, OS will grant instance if available.

#Strategies for dealing with deadlock:

1) Just ignore the problem (ostrich also)
- Pretend that deadlock never occurs in the system.
- Restart the system manually, if system seems to be deadlocked.
- Used by most OS including Unix.

2) Deadlock prevention:
- Prevent any 1 of 4 necessary conditions so deadlock will never occur.

@ Preventing Mutual Exclusion:
- All processes should be independent ie NO shared resource.
- All process should have there own resources

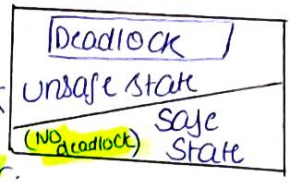
Both sol^n practically not possible.

ⓑ Preventing hold & wait:
- Process will either hold or wait but not together.
- If all resources are available then acquire or just wait for all.
- If process needs some more resource then it will release all acquired resources first then wait.
- Prblm: starvation; may not know required resources at start of run.

© Preventing NO preemption: Starvation
- Take resource from process forcibly.
- It is not a viable sol as process will have to rollback, all progress is lost.

(d) Preventing circular wait:
- Give all resources unique number.
- If a process holds resource Ri then it can request Rj only if $j > i$.
- If $j < i$ then process should release Ri & try to acquire Rj first.
- Its practical sol but may lead to starvation.

3) Deadlock Avoidance:
(requires knowledge of resource requirement apriori)
- We manage resource allocation so that deadlock can never occur.
- Safe state: system can allocate resources to process & avoid deadlock
- Unsafe state: deadlock may or may not occur.



- Before granting resource, OS checks if resulting state is safe then grant.
- Algo to avoid deadlock: Bankers Algo.
- Deadlock avoidance will never leave system in unsafe state.

* Bankers safety algo: TO check if the system is in safe state or not.

	Allocation			MAX			Available			Need		
Process	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

Allocation: present allocation of resources
MAX: MAX requirement of resources A, B & C
Available: available resources in system

- Need = MAX − Allocation
- Now check with available resources, anyone's need can be fulfilled or not.

	Available A	B	C	
	3	3	2	+ Allocation[P1]
After P1	5	3	2	+ Allocation[P3]
After P3	7	4	3	
P_0	7	5	3	
P_2	10	5	5	
P_4	10	5	7	

Safe sequence: <P1, P3, P0, P2, P4>

- If all process can run in any (atleast 1) sequence then its in safe state
- Total no of resources = ΣAllocation + Available
- If Need <= Available then processes can run in any order. (no of process)
- NO of possible safe sequence = n!

* Bankers resource request algo:
- TO check if the request can be granted.
- If Pi requests 2 instance of A & 3 instance of C ie $Ri = \langle 2, 0, 3 \rangle$

i) if Ri <= Need: goto step(ii) else raise error
ii) if Ri <= Available goto step(iii) else wait
iii) Available = Available − Request i
Allocation i = Allocation i + Request i
Need i = Need i − Request i
iv) Run safety algo, if safe then request granted else rejected.

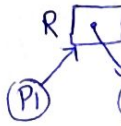
- If 2 requests Ri & Rj are made, possible options:-

i) both cant be granted
ii) both can be granted
iii) only i
iv) only j
v) Either i or j, not both

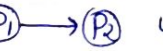
NOTE: If Ri is granted then Rj is granted then both can be granted in any sequence.

NOTE: Deadlock prevention is more stricter than avoidance & allows less concurrency. As, prevention may reject request even if resulting state is safe.

4) Deadlock Detection & Recovery
Detection:
- Deadlock detection is done using wait for graph.
- Wait for graph is made from resource allocation graph.



P1 is waiting for resource held by P2 ie P1 is waiting for P2



wait for graph

1) In wait for graph:
- if NO cycle ⇒ NO deadlock
- if cycle
  - if only 1 instance per resource type. ⇒ Deadlock
  - if resources have multiple instances ⇒ Possibility of deadlock

2) When resources have multiple instances deadlock detection is done using specific form of bankers algo.
In detection we just check given requests can be fulfilled or not.

3)

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

		Available A	B	C
		0	0	0
After	P_0	0	1	0
	P_2	3	1	3
	P_1	5	1	3
	P_3	7	2	4
	P_4	7	2	6

If all process can complete ⇒ No deadlock
Otherwise ⇒ Deadlock

NOTE: When allocation & Max given → Deadlock avoidance
when allocation & request given → Deadlock detection

4) When should detection be done:-
- Do after every resource allocation
- Or do when there is some clue like performance degradation.

Recovery from deadlock:
1) Inform to user & let him handle manually. (restart system).
2) Process termination:-
- Terminate all processes involved in deadlock.
- Terminate processes one by one untill deadlock is broken.

Factors deciding which process to terminate: process priorities, how close it is to finish, how many resources it holds & requires, etc.

3) Resource preemption:-
- Take care of process to rollback & process should not starve.

To find min no of resources system should have so that deadlock can never occur:-
- Allocate (max-1) no of resources to all processes
- Have 1 more available resource so that all process can run.
- Min no of resource = $(\Sigma(max-1)) + 1$ so that deadlock can never occur
- Max no of resources so that deadlock can occur = $\Sigma(max-1)$

Q: 3 process, 4 instance of a resource, each process request max K instance Max value of K that will always avoid deadlock?
$(K-1)3+1$ = Min no of instance to avoid deadlock
$\Rightarrow (K-1)3+1 \leq 4 \Rightarrow 3K \leq 6 \Rightarrow K_{max} = 2$

⇒ Types of Locks:-
1) Semaphore
2) Mutex (Spinlock)
3) Deadlock: processes will be blocked (in waiting state). will not waste CPU time.
4) Livelock: system is in livelock when the processes do repeated work without any progress its doing useless work. wastes CPU time.

Ex: Pi has to acquire R, if R is with some other process, kill it & Pi will take R.
Any process can kill another process & acquire resource, this can go in cyclic fashion ensuring livelock.

- Deadlock & Livelock are mutually exclusive, at any point of time only one can happen in a system
- Deadlock / Livelock → No progress
  Deadlock / Livelock → Starvation

5) Reentrant Lock: It is a mutual exclusion mechanism (type of mutex) that may be locked multiple times by the same process/thread w/o releasing previous lock & w/o causing deadlock while taking lock, count is incremented by 1 every time. A resource is locked until count returns to zero.

6) Non Reentrant lock : process/thread can't own some lock multiple times.

- If a process/thread tries to acquire already owned lock, it will get blocked its a deadlock.

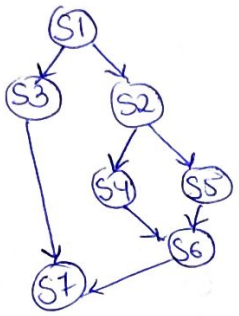
⇒ PARBEGIN / PAREND used to specify concurrent statements.

- Also called as Fork/Join, Cobegin/Coend

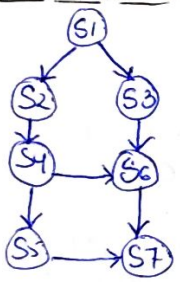
Ex:

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

(consider begin/end as 1 stmt)



⇒ fork/join :




```
S1
C1=2
fork L1
S2
S4
C2=2
fork L2
S5
goto L3
L1: S3
L2: join C1
    S6
L3: join C2
    S7
```

C2 initial value is 2 means 2 process which will execute it will merge at S7

fork L1 means a child process is created from label L1.

---

* Let n processes & k instances of a resource. Each Pi requires max Si instances of resource. To ensure deadlock does not occur :

$$\sum_{i=1}^{n} (S_i - 1) + 1 \leq k \Rightarrow \sum_{i=1}^{n} S_i < (k+n)$$

* I/O redirection : can use file for ip, op
  command <file : take ip from file
  command >file : put o/p into file
* Reference count is used by garbage collector Memory location whose reference count becomes 0 is cleared.

NOTE: Monitor is a syn tool, it ensures mutual exclusion but not deadlock.

Memory Management

1) Function:- Memory allocation, deallocation and protection.

2) Goal:- min wastage of space, ability to run larger program with limited space(virtual memory)

3) Memory management technique:-

- Contiguous
  - Fixed partition → internal fragmentation
  - Variable partition → External frag.
- Non contagious
  - *Paging → Internal frag.
  - Segmentation → External frag.

NOTE: virtual memory can't be implemented using contag. M. alloc

→ Fixed partition Cont. M. alloc:-

- MM divided into fixed no of (Default) partitions. Each partition can accomodate exactly one process.
- Partition allocation policy:- First fit, Best fit, Worst fit, Next
  - min internal frag.
  - max internal frag
- Degree of multiprogramming = No. of partitions in MM.

→ Variable Partition Cont. M. alloc:-

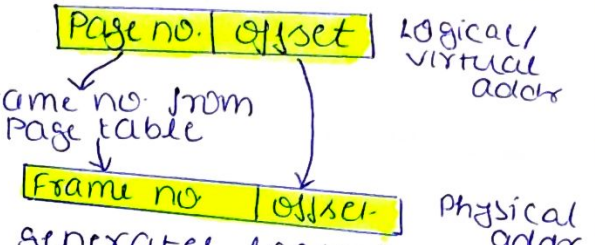
- Partition is created on arrival of process, of size equal to process
- Sol" of external frag:- Compaction. (time consuming)
- worst fit is better for it.

Non Cont. M. alloc:-

Process is divided into partitions & stored at different locations.

@ Paging:-

- Process is divided into equal size Partitions called Pages. MM is also divided into same size frames
- Each page can be stored at any frame in MM. This info is maintained in Page table for each process.
- Each Page table entry = Frame no + extra bits
- No. of entries in PT = No of Pages in process
- PT size = No of Pages * 1 entry size
- No. of pages in process = $\frac{\text{Process size}}{\text{Page size}}$
- No. of frames in MM = $\frac{\text{MM size}}{\text{Page size}}$
- No. of bits for offset = $\log_2$(Page size)
  By default Byte addressable → Byte in page
- No. of bits for page no. = $\log_2$(No of pages)
- No. of bits for frame no = $\log_2$(No of frame)
- Logical address: Byte no. of process. CPU knows only this not physical address.
- Physical address: MM addr at which corresponding byte is stored.



- CPU generates logical address. OS converts it into physical
- To find page no. 'p' entry in page table, skip p entries.
- OS maintains a page table for each process. Page table is also stored in MM in OS area.

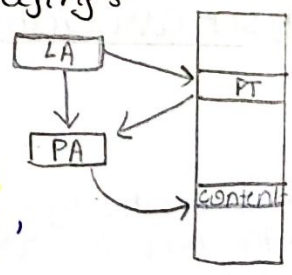
NOTE: If logical address is given in decimal & page size is not 2^k then don't convert in binary, calculate in decimal

- Page table base register (PTBR) stores base address of page table. Its present in PCB of process.
- Collection of logical addresses is called Logical address space or Process size. (or virtual)
- Collection of physical addresses is called Physical address space or MM size.

NOTE: if page table entry is 17 bits & page table size = $2^{20} * 17$ bits and it asks approx page table size then consider ans $2^{20} * 3B$ i.e 3MB not 2MB we may have extra bits, not less.

1) Performance of paging:-

- Time required to access content or Effective memory access time = $2 * T_{MM}$ (one for page table, one for content)



- If page table is very small & stored in a register
  EMAT = $T_{reg} + T_{MM} = T_{MM}$

* Translation lookaside buffer (TLB) is used to reduce EMAT, by storing frequently used page table entries.

2) CPU demands LA
↓
Search in TLB
↙ Hit → get PA → Access MM for content
↘ Miss → Access page table in MM to get PA

- TLB is a hardware.

3) EMAT with TLB = $H(T_{TLB} + T_{MM}) + (1-H)(T_{TLB} + 2*T_{MM})$
(NOT considering miss penalty) $= T_{TLB} + T_{MM} + (1-H)T_{MM}$
H: Hit ratio of TLB

4) Hit ratio = $\frac{\text{No. of hit}}{\text{Total no. of reference}}$

5) TLB always have hierarchical access

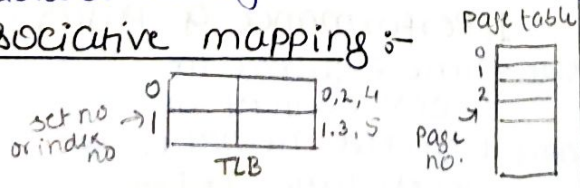
6) Few entries of current executing process page table is in TLB.

7) TLB Mapping:

ⓐ Direct Mapping:- uses ~~addressable TLB~~ (1 way set associative mapping)
- TLB entry no. = (page no.) % (No. of entries in TLB)
- Tag bits is used to distinguish b/w entries fighting at one slot of TLB.
- LA: | Page Number | Offset |
  | Tag | TLB entry no. |
- No. of entries in TLB = $\frac{\text{TLB size}}{\text{1 Page table entry size}}$

NOTE: Tag bits is not stored in TLB, its stored in a separate metadata storage.

ⓑ Set associative mapping:-



- Associativity: no. of PT entries in 1 set
- Tag bits is stored for each entry in TLB.
- TLB set no. = Page no. % No. of sets in TLB
- LA: | Page no | offset |
  | Tag | set no. |
- No. of sets in TLB = $\frac{\text{TLB no. of entries}}{\text{Associativity}}$
- 4 way set assoc. means 4 entries in 1 set

ⓒ Fully associative mapping:
- All entries in a single set
- | Page no | Offset |
  | Tag |
- Implemented using content addressable memory (Associative memory).
- Its very fast & costly.
- Page no. in LA is parallely compared to all cells & corresponding page table entry is taken.

accessing same address again →
NOTE: Based on temporal locality of reference, PT entry is brought to TLB.

NOTE: No. of bits in logical addr may be more than no. of bits in physical addr in case of demand paging.

1) Paging suffers from internal fragmentation (no external frag)
Int. frag. = Page size − (Process size % Page size)

2) Reducing page size may reduce internal fragmentation.

3) Reducing page size increases page table size. (bcoz no. of pages increases).

4) In paging, its possible that unrelated contents (like stack, heap) may be present in same page. so we went for segmentation.

5) Paging incurs memory overhead as it takes extra pages in memory for page table.

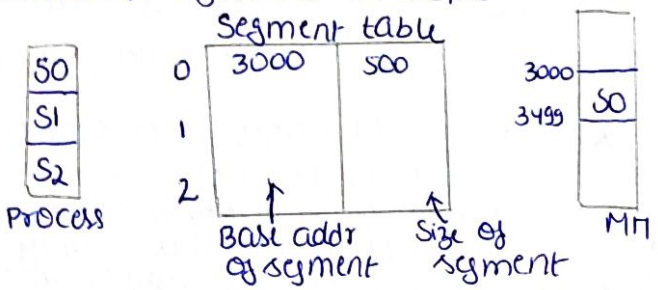
* By default, address space means logical address space.

NOTE: If there is no virtual memory concept in system, hardware support for memory management is no longer needed.

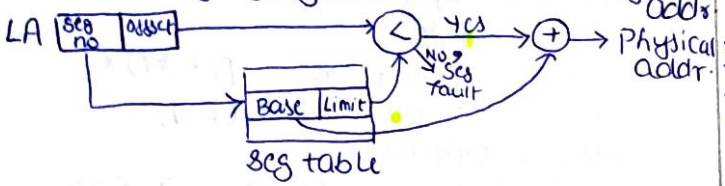
- In virtual memory system, MMU (Memory management unit) is required to translate logical address into physical address.

(b) Segmentation :-

1) Process is divided into logically related partitions called segments.
2) Segments may be of variable size.
3) MM is not divided, where space is available, segment is kept.
4)



5) Converting logical add. into Physical addr.



6) Seg. fault : Trying to access memory beyond limit. Its trap to OS (slw interrupt)
7) If limit is 500, it means segment has byte/word 0 to 499.
8) Offset bit size in logical add. is decided based on max segment size.
9) Segmentation suffers from External fragmentation. use compaction to avoid it.

⇒ Virtual Memory :-

* Size of virtual memory (VAS) is limited by size of disk, MAR
* In virtual memory system, practically
  Physical addr. space (MM size) ≤ virtual addr space (process size) ≤ Disk addr space (Disk size)
  - conceptually may have any relation

NOTE: 2 level virtual memory means 2 level paging.

Valid bit is 1 for pages that are in MM for rest pages it's 0.

5) CPU requests logical addr.
   ↓
   check valid bit of requested page in page table
   - 1 → Get frame no. & access content from MM
   - 0 → Page not present in MM (Page fault)

6) Page fault is a software interrupt. OS takes control, brings faulted page from secondary memory to MM, updates page table entry. Then the inst. is restarted. Its called page fault service.
7) If empty frame is available, OS keeps new page into it otherwise replaces an existing page. demand segmentation also possible.
8) Demand Paging: → default
   - Pure demand paging: initially when process is created, no page is loaded in MM.
   - Demand paging: initially few pages are loaded into MM.
9) Valid bit is used to ensure page hit / Miss.

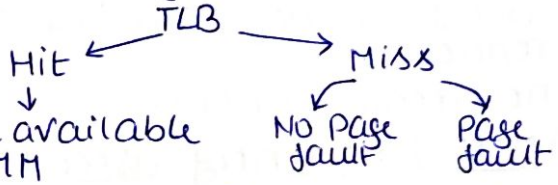
NOTE: Physical memory is MM (RAM).

10) Page swap time saving: have a extra bit "Dirty or Modified bit" in each page table entry.
   - Dirty bit 0 means only read is done on page, at time of replacement, overwrite it.
   - Dirty bit 1 means write is done on page, at time of replacement write back page to secondary memory
11) A victim page (to be replaced) is selected based on page replacement policy.

NOTE: Max process size = Max virtual memory size = Max secondary memory size.

12) Eff. memory access time
   = p * page fault service time + (1−p) * time w/o page fault
   p: page fault rate = No of page fault / No of memory access
   - Time w/o page fault = 2 TMM (default)
     = TMM (Page table in register)

13) Considering TLB :-



14) When a page is replaced from MM & if its entry is present in TLB, its made invalid & page table

NOTE: If a process requires more heap than allocated, then system call is made & process is reallocated memory.

⇒ Page replacement policies:-

used to find victim page.

(a) First in First Out:- (FIFO)

- Belady Anomaly: sometimes for some special page reference sequences, increasing no. of frames, increases no. of page fault. → Page which is replaced is next requested one
- It happens in FIFO (policies which are not based on stack)
- Poor performance ie more page faults.

(b) Optimal policy:

- Page jo future me jldi use nhi hona hai, replace it.
- use FIFO to break tie.
- not practically possible.
- Min no. of page fault.
- # Page fault of other policies ≥ # Page fault of optimal policy
- Requires future knowledge of program and is time consuming.

(c) Least Recently used (LRU): Practically its best

- Replace page which has not been used since longest period of time.
- use FIFO to break tie.
- Generally, LRU is better than FIFO.

NOTE: EMAT is for 1 page access.

- Total time for access of x pages
  = No. of hit * $(2T_{MM})$ + (x − No. of hit) * Page fault service time
- EMAT = Total time / No. of page access

* Counting also: LFU, MFU

(d) Least frequently used (LFU):

- Jiski frequency kam hai past me, replace it.
- Tie breaker: FIFO

(e) Most frequently used (MFU):

- Jiski freq zyada in past, replace it.
- Tie breaker: FIFO

(f) Last in first out (LIFO):

- Only last page is replaced always.

(g) Most Recently used (MRU):

- Replace page referred most recently.

NOTE: Locality of reference: page reference made by a process is likely to be one of the page used in last few reference.

- Temporal LOR: accessing same addr again
- Spatial LOR: accessing nearby addr again.

* Let p → no. of pages in a sequence
  f → no. of frames
  r → no. of repetition of sequence

If initially all frames are empty & f < p and all pages are distinct in seq. Then, no. of page fault in

- FIFO = pr
- Optimal (if f ≥ r) = pr − (r−1) * f
- LIFO = pr − (r−1)(f−1)
- MRU = Optimal
- LIFO − Optimal = (r−1) if f ≥ r

NOTE: virtual memory can't be implemented using contiguous memory allocation.

⇒ Frame Allocation: depends on Addressing mode, instruction architecture

1) Every process must have enough pages in MM to complete an inst → Min no of frames
2) Types:
   - Equal allocation: Every process is allotted equal no of frames.
   - Proportional allocation: Frames are allocated proportional to the process size.

⇒ Page Replacement: → Local / → Global

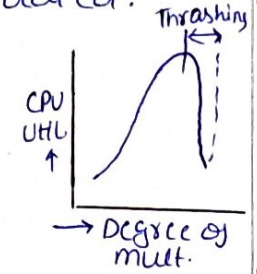
1) Local Replacement:
   - Page being replaced be in a frame belonging to same process.
   - No. of frames belonging to a process will not change.
   - Processes control there own page fault rate, do not affect page fault of other processes.

2) Global Replacement:
   - can replace a page of itself or other process from a set of all frames allocated to user processes.

- High priority processes can increase their allocation at the expense of lower priority process.
- More efficient use of frames & better throughput.

3) In local replacement, only 1 page table is updated & in global replacement 1 or 2 page table is updated.

⇒ Thrashing:



1) With increase in degree of multiprogramming, CPU utilization increases & after a point of time it decreases rapidly.

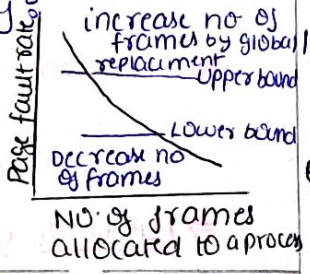
2) Thrashing: when CPU spends more time on page fault service rather than process execution.

3) On seeing CPU utilization low, long term scheduler schedules more & more no. of process. CPU is thrashed.

4) How to handle thrashing?

ⓐ Working set model:-
- Each process is provided with it's working set in MM.
- At present, set of pages process requires frequently is its working set.

ⓑ Page fault frequency
- Upper & lower bound of page fault rate is decided & OS tries to keep PFrate of each process within range.



* Hit ratio = $\frac{\text{No. of hit}}{\text{Total no. of page reference}}$

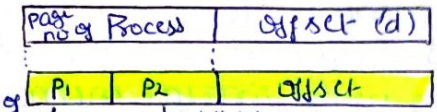
* Page no. of process = $\left\lfloor \frac{\text{Address}}{\text{Page size}} \right\rfloor$

* Random page replacement policy (may behave like FIFO): selecting a page randomly for replacement. It may suffer from belady anomaly.

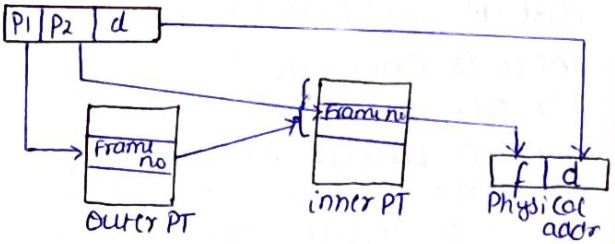
⇒ Multilevel Paging:

1) If page table can't fit in 1 page, it's also divided into pages & stored randomly on frames in MM.
↳ we have other page table to store its mapping.

* Helps to reduce page table size. In demand paging, pages that are used, PT entries of only that pages r present in MM

2) LA: 

3) This mapping is done until the outer most page table is stored on a single page.

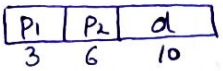
4) 

5) PTBR stores address of outer most page table.

6) More page table levels → translation overhead increases.

7) Every PT is stored in memory.

8) Total pages required for page table = Σ all levels.

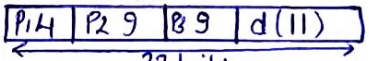
9) 

No of pages for outer page table = 1 (always)
No of pages in inner page table = $2^{P1} = 8$

10) Page table entry (frame no. + extra bits) will be same across all levels.

11) Total size process takes = Process size + (No of pages * Page size) PT takes

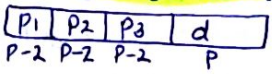
Q: Page size = 2KB, PT entry = 4B
Logical add size = 8GB

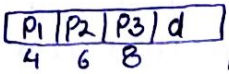


∴ 3 levels needed

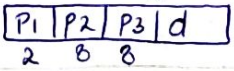
No. of PT entries in 1 page = $\frac{2^{11}}{2^2} = 2^9$
No. of pages in outermost PT = 1
" " " middle PT = $2^{P1} = 2^4$
" " " inner PT = $2^4 * 2^9$

Q: PT entry size = 4B, LA size = 16GB
3 level paging, outer PT occupies complete page. Page size = ?
Let Page size = 2^P; No of PT entries in 1 page = $\frac{2^P}{2^2} = 2^{P-2}$



12) 

This is not possible. All inner PT's will have max entries in each page. Outer may have ≤ max no of entries.



is correct.

13) 3 level page table means 3 times MM access required to get physical addr.

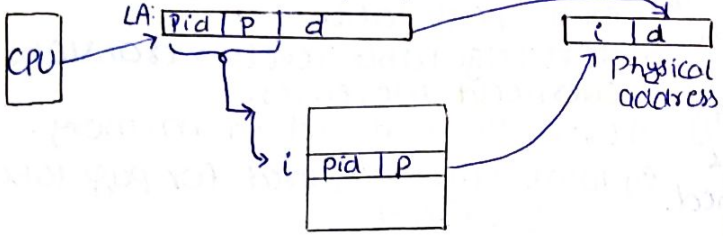
14) Prblm with virtual memory: If PT is too big, so many entries is invalid.

⇒ Inverted Page table:-

1) It has entry for each frame of MM which contains:
Page no, Process Id, Extra bits

2) There is only one inverted pagetable for all processes.

3) We do Linear searching, if found its index (frame no) is used else its page fault. Its time consuming but saves lot of space.



4) If we skip 'i' no of entries in inverted PT, we are at entry of frame no i.

5) Difficult to implement page sharing.

* In simple paging, page sharing is implemented by having some page table entries pointing to same frame in MM.

⇒ Hashed PT:

1) Entry of pages that are present in MM are hashed into PT as in chaining.
Ex: 16 pages in a process
4 hashed location in PT
Slot no = page no % 4

2) Each process has its own hashed PT.

3) Each node contains:
Page no, PT entry, pointer to next node

* TLB always have hierarchical access.

⇒ TLB and Multilevel Paging:- (ignoring Page fault)

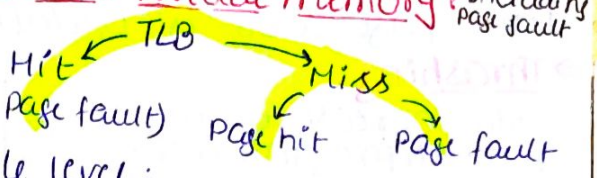
1) Single level paging:-
- Default EMAT = $2 * T_{MM}$
- $EMAT = H(T_{TLB} + T_{MM}) + (1-H)(T_{TLB} + 2 * T_{MM})$
$= T_{TLB} + T_{MM} + (1-H) T_{MM}$

H: Hit in TLB T_{MM}: MM access time
T_{TLB}: TLB access time

2) 2 Level paging:-
- Default EMAT = $3T_{MM}$
- $EMAT = H(T_{TLB} + T_{MM}) + (1-H)(T_{TLB} + 2T_{MM})$
$= T_{TLB} + T_{MM} + (1-H) 2T_{MM}$

3) K-Level paging:
- Default EMAT = $(K+1) T_{MM}$
- $EMAT = T_{TLB} + T_{MM} + (1-H) K * T_{MM}$

⇒ TLB with virtual memory: (including page fault)



1) Single level:
$EMAT = H(T_{TLB} + T_{MM}) + (1-H) * [T_{TLB} + T_{MM} + (1-p) T_{MM} + p * \text{PF service time}]$
($T_{TLB} + T_{MM}$: PT lookup; p: page fault rate)

2) 2-Level paging:
$EMAT = H(T_{TLB} + T_{MM}) + (1-H) * [T_{TLB} + 2T_{MM} + (1-p) T_{MM} + p * \text{service time}]$
($2T_{MM}$: For K level PUT KT_{MM})

⇒ Including dirty page:-

1) Single level:-
$EMAT = H(T_{TLB} + T_{MM}) + (1-H) * [T_{TLB} + T_{MM} + (1-p) T_{MM} + p\{d * \text{service time for dirty page} + (1-d) * \text{service time for non dirty page}\}]$
d: fraction of dirty page
p: page fault rate

2) In case of K-level paging, just change lookup time (real one) to $K * T_{MM}$.

NOTE: Look for TLB hit → Page fault → dirty page

NOTE: If page transfer time b/w MM & disk is x
For dirty page take it 2x.
For non dirty page take it x.

⇒ TLB with cache:-

1) TLB stores page table entries
cache stores process content

2) We can access cache from physical address only (not logical addr) (by default)

3) Page table is not kept in cache.

4) TLB
- Hit in TLB → we got phy addr
  - Cache hit (get content)
  - Cache miss (goto MM & access content)
- Miss (not considering page fault) → Access PT in MM to get Physical addr.
  - Cache Hit (get content)
  - Cache miss (get content from MM)

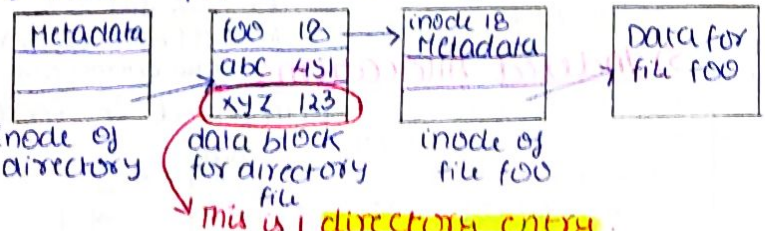
5) $EMAT = H_{TLB}[T_{TLB} + H_{cache} * T_{CM} + (1-H_{CM})(T_{CM} + T_{MM})] + (1-H_{TLB})[T_{TLB} + T_{MM} + H_{CM} * T_{CM} + (1-H_{CM})(T_{CM} + T_{MM})]$
$= T_{TLB} + T_{CM} + (1-H_{TLB}) T_{MM} + (1-H_{CM}) T_{MM}$

- Change lookup time (reason) to $K * T_{MM}$ in case of K-level paging.

File System implementation:

1) We r going to see how we store info/data ie files on magnetic disk.
2) Addressing unit of disk = unit of atomicity = **Block or Sector**
Here we will term sector as block.
3) Disk reads/writes in terms of block, not bytes.
4) File: Data in some format (.c, .exe, .pdf)
5) File System: module of OS to manage files
6) File operations: rename, read, write, etc
7) Objective of FS: efficient fetching, should not take much time to know where file is stored on disk.
8) Directory: A file with special bit set in metadata to indicate directory
NOTE: In UNIX everything is file.
NOTE: We are talking with reference to UNIX OS
9) File Metadata: Name, inode no., location, size, Protection, time, date, user.
10) Metadata is stored in data structure called **inode** (in UNIX) ie index node.
11) For **every file or directory → 1 inode**.
Every inode has unique inode number.
12) File (or directory) = **Metadata + data** (inode)
13) inode for a file:-

14) inode for a directory:-



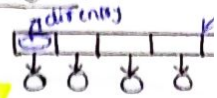
15) inode of directory can be implemented like this also:

Metadata	
foo	18
abc	451
xyz	123

16) File system job is to translate (filename, offset) into block no ie disk address.
17) inode is used to keep track of sectors allocated to a given file.

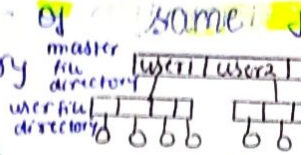
File directory structure:

1) Single level Directory
- 1 directory for entire system

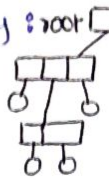


- **Two files can't have same name** (out of some)

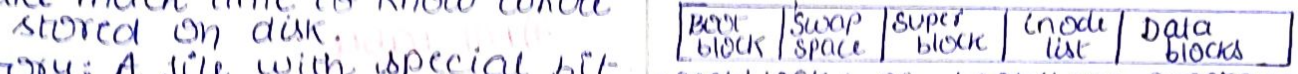
2) Two level directory
- 1 directory for each user
- All user names are visible but one user can't access other user data



3) Tree structured directory: root
- used in modern system
- If anyone has access to root folder can access everything inside.



Disk is divided into some parts:



- Boot block: OS, bootstrap program
- Super block: # files, free blocks, etc

1) Disk has **magnetic medium** of storage.
2) Disk partitioning/Formatting:-
- Physical (low level): creating tracks & sectors; done by manufacturer
- Logical (High level): creating logical partitions (drives); done by user

3) Types of logical partition:
- primary (C drive): contains OS + user files
- Extended: contains user files, can't install OS.

4) We may have different OS installed on different primary partition.
NOTE: n blocks are numbered 0 to n-1.

5) Types of File system: FAT32, **NTFS** (Present in windows), **HFS+**, Ext2/Ext3/Ext4 (Mac)
6) To open a file, CPU asks OS & OS asks file system to open.
NOTE: Any block can't contain content of 2 different files.
* A directory can't have 2 files of same name & type.

⇒ Disk blocks :-

1) Total disk size = No. of blocks * 1 block size
2) Disk block address = Disk block no.
3) Disk block add. size = $\lceil \log_2(\text{no of blocks}) \rceil$

⇒ Free space management:

- Which blocks are free, data is maintained in file system.

1) **Free List**: maintain linked list of free block no (address).
2) **Bitmap**: For each block 1 bit is stored (1: Empty / 0: occupied)
   For n blocks, n bit info
   Block → 0 1 0 0 1 . . . 0 1
- Free list does not require searching of free block, bitmap requires searching.
- **Free list is faster**
- Free list size is variable.

⇒ File allocation Methods :-

1) Contiguous Allocation:

- File is stored on consecutive blocks.
- Metadata (inode) contains starting block address & no of blocks.
- Fast sequential & random access.
- Low storage overhead.
- Fragmentation: **Internal & External**

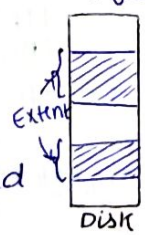
Here, compaction is known as defragmentation
Its time consuming process.

- Difficult to grow file

NOTE: All allocation may have internal fragmentation.

2) Extent-based allocation : variation to contiguous

- Multiple contiguous regions per file (like segmentation)
- Each region is an extent.
- Metadata: For each extent, start & size of extent is stored in inode.

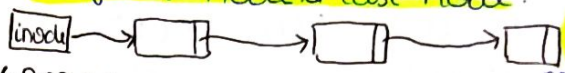


* Pros:
- Low storage overhead
- File can grow overtime.
- Sequential & random access.

* Cons:
- It helps with external fragmentation but still a problem.
- Internal fragmentation.

3) Linked Allocation:

- All blocks of a file are in linked list. Each block has a pointer to next block.
- Metadata: inode contains pointer to first node & last node.

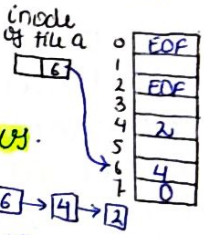


* Pros:
- NO external fragmentation.
- Files can easily grow.

* Cons:
- Internal frag.
- Slow sequential access, difficult random access not possible.
- Large storage overhead (one pointer per block).

4) FAT table : (variation to linked)

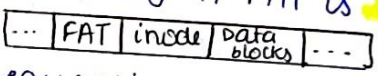
- Store linked list pointers outside block in File allocation table.
- A section of disk is used to contain this table.
- FAT has 1 entry for each block of disk.



- Used in MSDOS & Windows.
- Still we do pointer chasing but can bring entire FAT in MM so can be cheap compared to disk access.

file a: 6 → 4 → 2

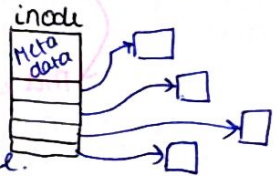
- Generally, entry in FAT is block no.



- Slow sequential access, random access possible.
- Large storage overhead for FAT table
- Assuming all available storage space used to store one file, Max file size = Total disk size - overhead (FAT table)

5) Indexed Allocation: → No external frag

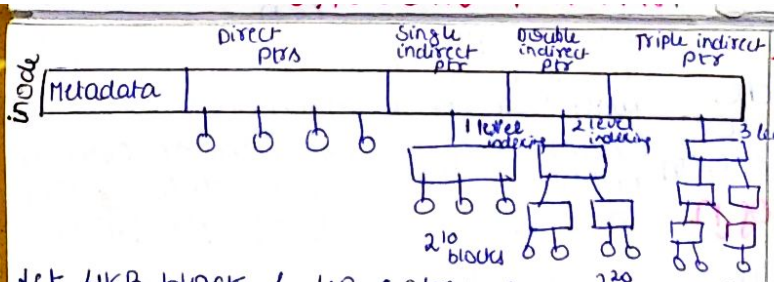
- Index table is maintained in inode
- If inode has space for 4 pointers, it can have atmost 4 sectors.



- Both sequential & random access possible.
- It requires large inode size so that file can grow as required.
- Large inode will have lot of unused entries initially.

6) Multilevel indexed :- (variation to indexed)

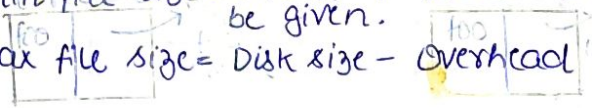
Unix inode (index node) structure



Let 4KB block & 4B entry size
∴ 1024 entries per block

NOTE: Min file size: atleast 1 block should be given.

Max file size = Disk size − Overhead



Q: Block size = 2^{12}B. Each file has 13 direct ptr, 4 singly indirect, a doubly indirect & a triply indirect ptr. No of inodes that can fit into a block. Ptr size 4B

pointers size = (13+4+1+1)4B = 76B

Assuming other metadata, consider it 128B

(closest ≥ power of 2)

No of inodes that can fit in a block = $\frac{2^{12}}{2^7} = 2^5$

* Performance of sequential access: contiguous > linked > indexed
* Performance of random access: contiguous > indexed > linked

→ Consider file consisting of 100 blocks. Assume index block is in MM. In contig. allocation, there is no free block to grow at beginning but can grow at end. Also block info to be added is present in MM. Calculate no of disk IO operations required.

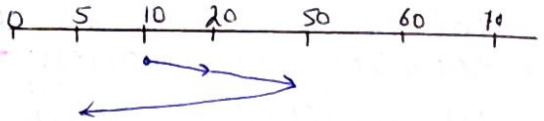
(disk access)	contiguous	linked	indexed
• block is added at beginning	201	1	1
• add in middle	101	52	1
• add at end	1	3 (assuming last node addr present in inode)	1
• Remove from beg.	0 (do it, shift free block)	1	0
• Remove from middle (51th block)	98	52	0
• Remove from last	0	100	0

⇒ Disk Scheduling:

1) Requests made by CPU to disk are kept in disk (IO) queue. OS need to schedule these requests in order to improve performance of system
2) To reduce seek time, we store file in cylinder. Considering request is made for a cylinder. (track no.)

1) First come First serve: (FCFS)

- Requests are served in same order in which it comes.



No. of head movements = (50−10)+(50−5) = 85

If for 1 cyl movement, disk head takes 2 ms, seek time total = 85*2 = 170 ms

No. of times head changes direction = 1

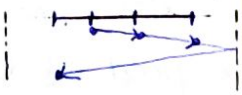
- No starvation

2) Shortest seek time First: (SSTF)

- Fulfill the request which is closest to the current position of the head.
- Throughput increases (Best throughput)
- Average response time decreases.
- can cause starvation.
- High variance of response time

3) SCAN: (Elevator)

- Go in one direction to end fulfilling all requests, then go in other direction fulfilling requests until all requests are served.

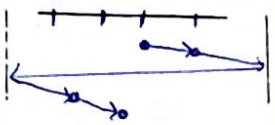


- Always 1 time head direction changes at corner (very last cylinder).
- High throughput
- Low variance of response time.
- Long waiting time for requests just visited by disk arm.

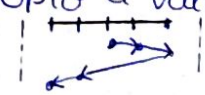
NOTE: By default, arm moves towards larger value.

4) C-SCAN: (Circular scan)

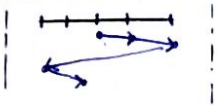
- without fulfilling any request go from 1 end to other ie requests are fulfilled in one direction only.
- Provides more uniform waiting time compared to SCAN.



5) LOOK: Similar to SCAN, don't go to corner, go only upto a valid request.



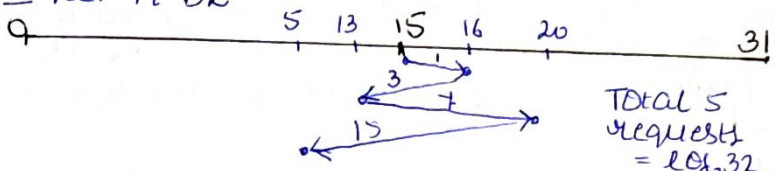
6) C-LOOK: Similar to CSCAN, we don't go upto corner.



NOTE: In case of tieup continue going in same direction (in SSTF)

* If the total no of tracks are n & head can start from any track. Max cardinality of request set so that the head changes its direction after servicing every request = $\log_2 n$ (follows SSTF)
  - For max cardinality, start from middle.

Ex: Let n=32



Total 5 requests = $\log_2 32$

- use this formula when n is power of 2 otherwise do manually.

---

* A process may be multithreaded & some threads may generate disk request parallely.
* If OS is capable of handling single sequential process at a time, then only 1 request will be live at a time & hence there is no need of disk scheduling algo. we may use any schalgo there will be no improvement in i/o performance of process.

---

* speed of CPU > disk > I/O
* Priority of interrupt by CPU > disk > mouse > keyboard
* A processor needs sw interrupt to obtain system services which need execution of privileged instructions.
* Generally, devices with high speed are given higher interrupt priority & slow devices get low priority.
* A multiuser / multiprocessing OS can not be implemented on hw that does not support: address translation, 2 modes of OS.

---

* overlay: is used to run a program that is bigger than the size of physical memory (MM) on systems where virtual memory is absent by only keeping required data in memory at given time.
* Overlaying is used to run a program which is longer than the address space of computer. False
* virtual memory is used to accomodate program which is longer than the address space of computer. False becoz by default address space means logical address space.

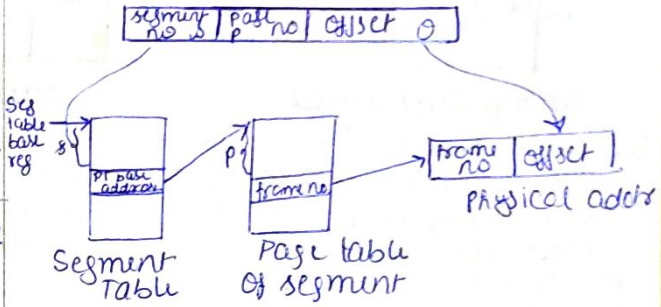
---

* Interrupt service routine can be written in high level language. also but its performance will be bad.
* Macro defination cannot appear within other macro defination in assembly language program.

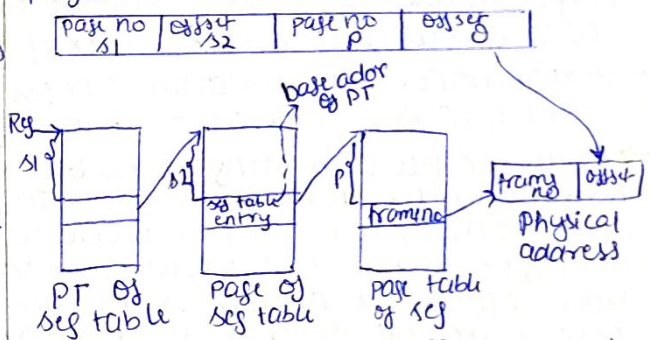
⇒ Combination of paging & Segmentation

- segments are divided into pages, there is page table for each segment

1) Segmented paging:



2) Paged segmentation: Sometimes seg table is too big to keep in 1 block. so, seg table is also divided into pages.



GO8: PA: frame no | offset (← 36 bits →)

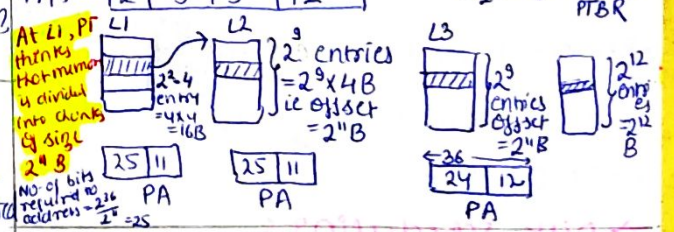
VA: 32 bits — L1 | L2 | L3 | offset (2 | 9 | 9 | 12)

Page: 4KB ; PTentry: 4B ; 3 level PT

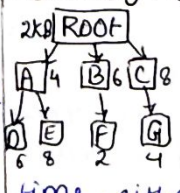
Find no. of bits required for addressing the next level page PT or page frame in the PTentry of 1st, 2nd, 3rd level PT

VA: 2 | 9 | 9 | 12

$2^{32}/2^4 = 2^{32} \Rightarrow$ 32 bit PTBR



* Overlay tree of a program:



What will be the size of partition in MM required to load & run this program?

In overlay, we only need to have the part in MM which is required at that instance of time, either we need root-A-D or root-A-E or root-B-F or root-C-G.

max (12, 14, 10, 14) = 14KB

If we have 14KB partition, we can run any of them.