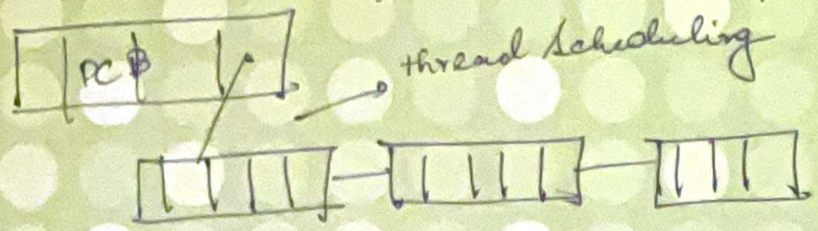
* DFG & CFG → data flow graph & control flow graph



* ~~Process~~ Threads share resource & variable
* no protection in threads.

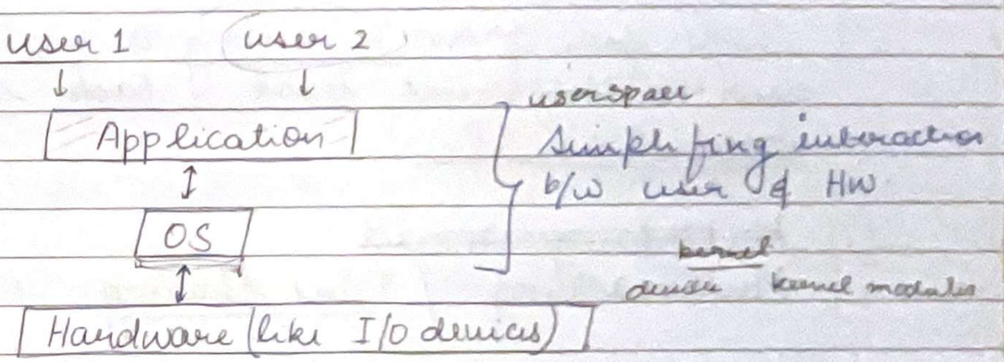
Caching is often used between storage systems
→ Copying information into faster storage system.

Memory layout

Stack → ~~program stack~~ Atomatic var / temp data
↓
↑
Heap → memory dynamically allocated during runtime.
data → global var
text → code/executable instructions
Sharable & Read only.

System Software AKA OS

- It is an interface between users and hardware



- Goal of OS is to bring higher convinence & throughput.

Functionality

1) Resource managment ✓
2) Process managment ✓ → CPU scheduling
3) Storage mandgment ✓
4) memory managment ✓
5) Security and privacy ✓

→ Types of operating systems

1) Batch

* tasks are divided based on similarity
* CPU can be idel sometimes.

eg: Fortran or IBSYS 709X & 1960's.

2) Multiprogrammed

* non - preemtive

Task goes from running to waiting and a different task ~~task~~ takes over the CPU.

3) ~~Multiprogrammed~~
Multitasking / Time Sharing RR.

* preemtive.

task is executed based on time

Certain amount of time is devoted for each task

* goal is to be more responsive.

4) Real - time OS.

* response needs to be immediate.

hard : missile systems ✓

Soft : local comp. ✓

5) Distributed.

The machines are spread over a large ~~go~~ geographical area

6) Clustered

Seperate machines connected together acting as a single machine

eg: Super computer

7) Embedded Ubuntu Core

used for a fixed functionality

- OS acts as a
  - Resource allocator.
  - Control program

First OS

ATLAS → Manchester University
evolved from control program

More detail on types of OS

* Multiprogramming

It increases CPU utilisation by keeping Jobs in memory so that CPU always has 1 to execute.

* Multitasking

It reduces response time.

Important definitons

Kernel

Part of OS, which interacts directly with hardware to perform most crucial tasks.

☆ Microkernel

Smaller part means for 'core' OS functionality.

Shell

Command interpreter

recives commands from user & gets them executed.

System Call mech

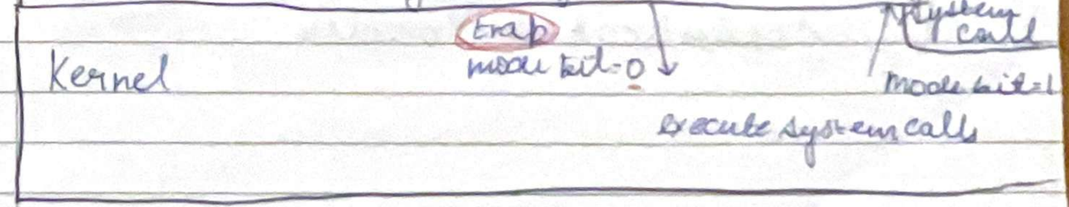
mechanism using which user prg can request a service from kernel for which it does not have access to.

eg: User program
|
wants access to
↓
I/O devices.

* System call provide interface between program and operating system.

eg: fork, exec, wait, exit

dual-mode operation
- user → 1 mode bit
- kernel → 0 mode bit



☆ Real time OS.

It has well defined & fixed time constraints which have to be met or the system will fail.

- used as a control device in a dedicated application.

eg: Embedded systems.

Booting → kernel

Process of starting the computer & loading the kernel.

Computer ON
↓
Power on self test are performed.
↓
Bootstrap loader resides in the ROM
↳ is executed.
↓
loader loads Kernel & Sophisticated loader.

kernel → device drivers

☆ Device drivers make up the major part of all operating systems kernels

They operate in highly permitted environment

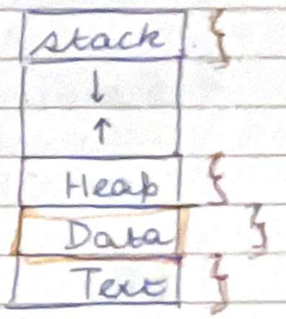
Process States & Life Cycle

(Passive)

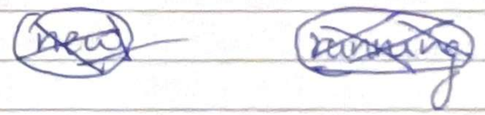
☆ Process/ Job: A program in execution

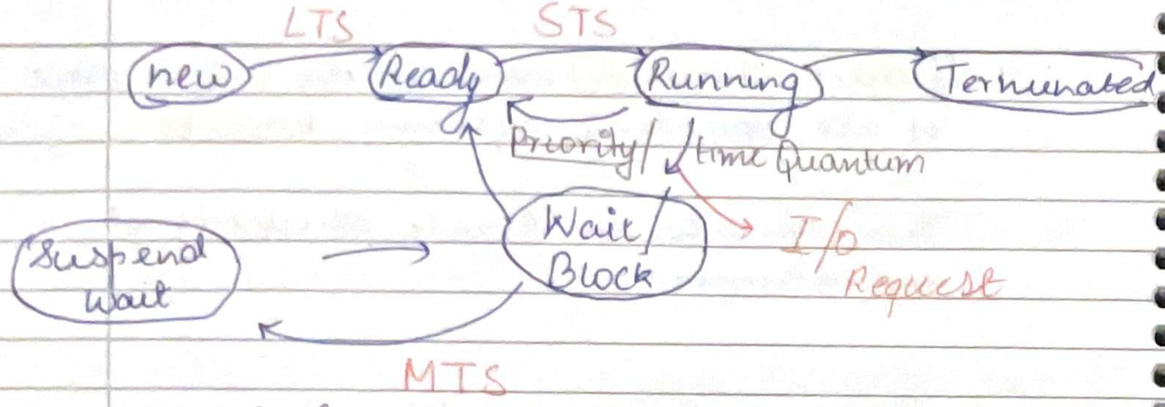
Program is a passive entity while process is an active entity

PCB



☆ Process states: Current activity of the process.





- new: Create a ~~program~~ process (Secondary memory)
- Ready: → Ready Queue (RAM)

- long term Scheduler (LTS)
  - Multiprogramming — non-preemtive

- Running state (also in RAM)
  Process dispatched from Ready queue to CPU

EP
Number of CPU will determine how many process are running at the same time.

- Terminate.

Deallocate the memory

- Short term scheduler
  It schedules task b/w Ready & Running state, depending on priority or time

  It will also decide if tasks are preemtive or non-preemtive

doubt
- Preemtive
  - priority/time tasks keep moving from CPU to Ready Queue & vice versa.

Non-preemtive
Single task based on priority

- wait state
  (Also in RAM) (also a Queue)
  for I/O Requests, after Request is over process is moved from wait to ready Queue.

- Suspend wait (secondary Memory)

Process removed from wait Queue to 'suspended wait state' (in Secondary memory), if wait Queue is full.

Moved back into wait state later.

- Medium Term Scheduler.

Move from & back

Wait ⟶ Suspended Wait

sometimes moves process from

Ready ⟶ ~~new~~ Suspend Ready

if Ready Queue is full & higher priority task comes in.

- Backing store (Main to Secondary Memory)

If wait & Ready Queue full & want to make process back then,

Suspend Wait ⟶ Suspend Ready

* Process Control Block

Process Representation

PCB

↳ Process
↳ Value of prog registers
↳ CPU sched
↳ I/O status
↳ accounting
↳
↳

located in kernel stack

- Process State ✓
- Process Number ✓
- Program Counter ✓
- Register ✓
- Memory limit ✓
- List of open files ✓

PCB also has
→ CPU scheduling info
→ Memory management.
→ I/O status infor
→ Accounting infor

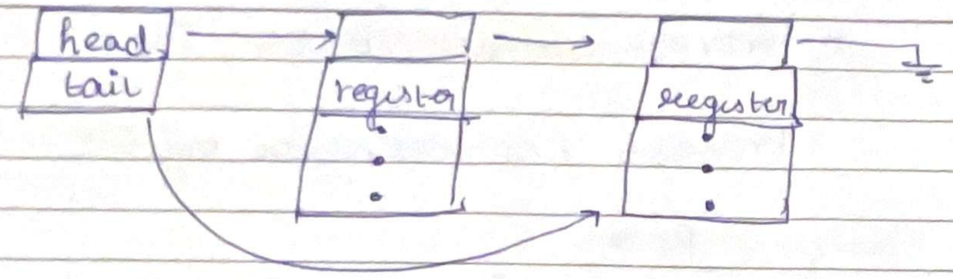
Scheduling

Selecting 1 process for execution out of all the ready processes.

Scheduling Queues

NEW STATE → Job Queue : All process stored when created.

Ready Queue : (last page)
↓
implemented as a linked list of PCB



Swapping memory

Long term Scheduler

non-prc

→ Job Queue to ready Queue (RAM)
→ not used frequently
→ should maintain balance of
{ compute process & I/O process }
→ Controles degree of multiprogramming

Short term Scheduler

pre

→ Ready Queue to Running state
(RAM) (CPU)

Intr ☆ Linux, Windows.

They are multitasking OS. They do not have LTS

Swapping
context Switching

Medium term Scheduler

→ Running ~~(RAM)~~ (CPU) ⇀ Ready Queue (RAM)

☆ Removes process from memory & from the competition for the CPU, hence reducing the degree of multiprogramming

☆ { This process can be later reintroduced into memory }

This is called Swapping

SWAP → Process main memory to disk (Secondary memory)

Context Switching

- Switching between process
- states are saved & reloaded into PCB.
- time is not well spent during context switching.
- Context switching is faster for RISC processor with overlapped register window.

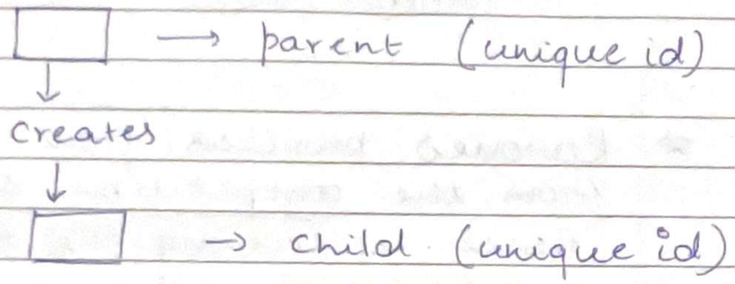
Reduced Instruction Set Computer.

fork() - System Calls creates new process

return code → 0 → for child
→ 1 → for parent

Process Creation.

* 1 process creating another process.



* Parent process
↓
Can execute simultaneously with child.

Can wait for child

Question →

* Child can be → duplicate
→ or new, new program loaded into it.

Process Termination.

* Deletion of the PCB

* if parent terminates before a child, child process is an orphan.

u { r 4, w 2, x 1 } g { r 4, w 2, x 1 } o { r 4, w 2, x 1 } → Octal rep for permission

* OS does not allow an orphan process
→ adopted & terminated

* Zombie process. PCB

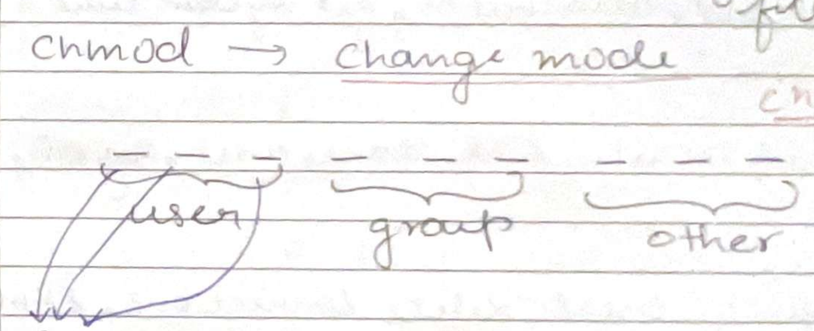
Child - Process terminates but its PCB still exists because parent has not accepted its return value.

Important linux commands

* command used to assign only read permission to all 3 categories for file note.

chmod → change mode

chmod



3 because we have 3 types of permission

ans: chmod ugo + = r note

Q Octal rep of ugo +rw note is

ugo 6 6 6 note.

r=4, w=2, x=1 } = 6 for u, g, o
6, 6, 6

System Calls

mech 40

* It is a software interrupt used by the user to get kernel's permission to perform certain tasks.

File
→ open(), Read(), write(), close(), Create file

Devices
→ Read, write, Reposition, ioctl, fcntl

Informati
→ get pid, attributes, get system time & data

Process control
→ Load, execute, fork, abort, wait, signal, allocati

Comm
→ pipe(), create/delete connections, shmget()

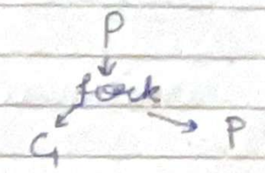
Fork System Call

* To create a child process

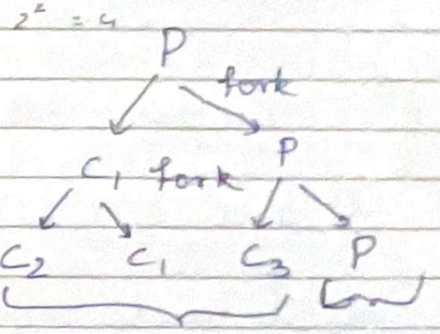
- Clone of parent, but id is new.

Fork → 0 child
+1 +ve parent
-1 → child was not created

```
main() {
  fork();
  printf("hello"), }
```




```
main() {
  fork(); fork()
  printf("hello");
}
```



* no of child process created → $2^n - 1$
n: no of times fork is called.

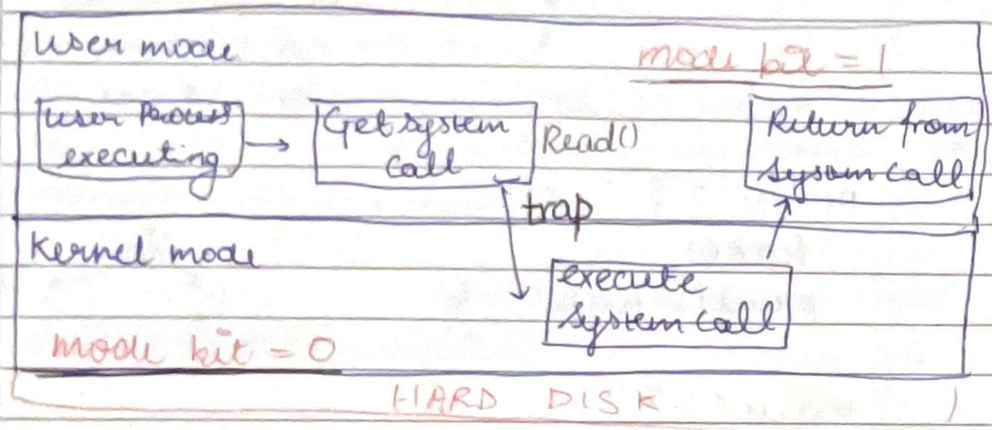
3 Child, parent

* total process = 2^n ✓

Q. if (fork() && fork())
fork()
printf("hello world"); → 4 times

System call is a software interrupt, but suppose Read() is used, it triggers trap which is an interrupt.

User mode Vs Kernel mode.



Process Vs Threads

	Process	Threads
1)	System Calls in process	1) No system call
2)	OS treats process differently	2) All user level threads treated as simple task for OS.
3)	Different process ~~copies~~ have different copies of data, files, codes	3) Threads share same copy of code & data & files
* 4)	Context switching is slower	4) Context switching is faster.
* 5)	Blocking process will not block Other	5) Blocking a thread will block entire process.
6)	Independent	6) Interdependent.

Cooperative threading models.

CHECK

	User level thread	Kernel level thread
1)	Managed by ~~OS~~ user level library.	1) Managed by OS
2)	faster	2) slower.
3)	Context switching is faster	3) Context switching is slower.
4)	If 1 thread is blocked entire process is blocked	4) 1 kernel level thread has no effect on others

thread share (Code & data)

Confirm both of these before the exam

If Kernel is single threaded, system call from any thread can block the entire task

Process does not share resources well
↳ high context switching overhead

Threads

* Smallest sequence of instruction that can be managed independently by a scheduler.

Seperate concurrency from protection.

Parallism : System can perform more than 1 task simultaneously

Concurrency : Supports more than 1 task making progress.
↳ Single core/processor

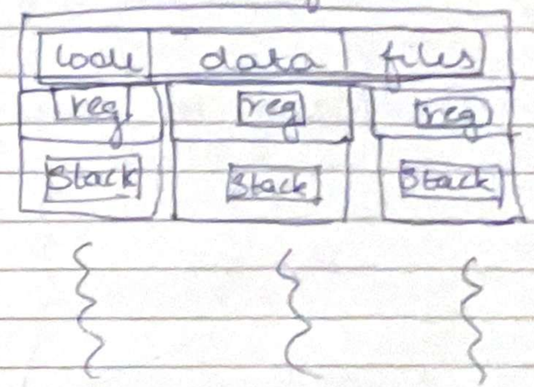
Type of parallelism

Data : Same data distributed on multiple core
Same operation on each.

Task para : threads are distributed each doing unique operation.

Multicore can have concurrency & parallism

Threads share



Imp

* Multithread task
1 thread blocked & waiting, a second thread can run.

memory, file system

Thread share → memory, filesystem, network connection

Private

TCB
CPU registers, execution stack
Program counter

Threads share CPU

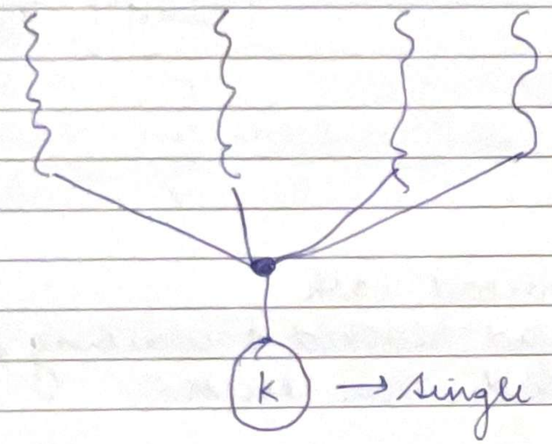
States

Ready | Blocked | Running | Terminated

like waiting in Process

Multithreading

1) Many to one



→ Single KERNEL thread.

* many user level thread
  ↓ to
  Single Kernel

* 1 thread blocked, entire task blocked

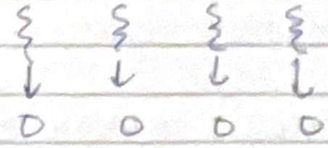
* Can not be parallel, since only 1 kernel thread.

eg: Solaris

In linux it's called task

fork() - clone

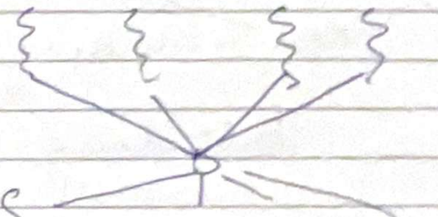
2) one to one



* More concurrency.
* creating user level create kernel

eg: Linux

3) Many to many



eg:

Thread libraries provide programmer with API to implement threads.

POSIX standard (IEEE 1003.1c) API for thread creation and synchronisation

Interprocess Communication

* 2 — Modes of IPC

IPC

shared memory
message passing

Direct & in-direct.

Mention process — Send to mailbox

Synchronisation

→ Message passing may be either blocking or non-blocking

Blocking → Blocking

Non-Blocking

Blocking → Synchronous

Non-Blocking → Asynchronous

Chapter - 3
Process Synchronisation

Process

Cooperative process — Independent

* Share - Variable, memory, Resources, code

* They can effect each other

* They dont effect each other.

* Race Condition

Situation where several process access and manipulate the same data concurrently and outcome depends on the order of execution.

→ can be solved by synchronisation

Producer and Consumer

* A race condition problem

Produce & consumer are permitted while executing their critical section.

Imp C++/C-- are 3 atomic instructions in assembly

Producer I_1, I_2, Consumer I_1, I_2, Producer I_3, Consumer I_3.

Printer - Spooler : Similar to producer consumer

Problem of race condition

→ Leads to loss of data

Critical Section

Part of the program where shared resources are accessed by various processes

Lock of res { Entry Section

Critical Sec } Common/Shared RESOURCES.

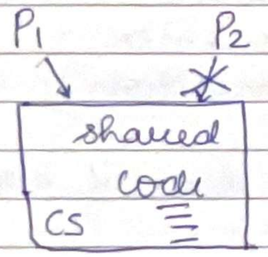
unLock of res { Exit Sec

REMAINDER SEC

4 RULES FOR SYNCHRONIZATION

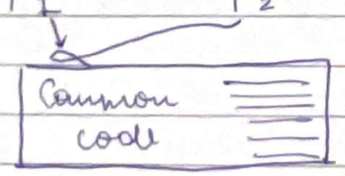
1) Mutual Exclusion
2) Progress
3) Bounded Wait
4) No assumption about HW or Speed.

1) Mutual Exclusion.



Both process can not be in critical section at the same time

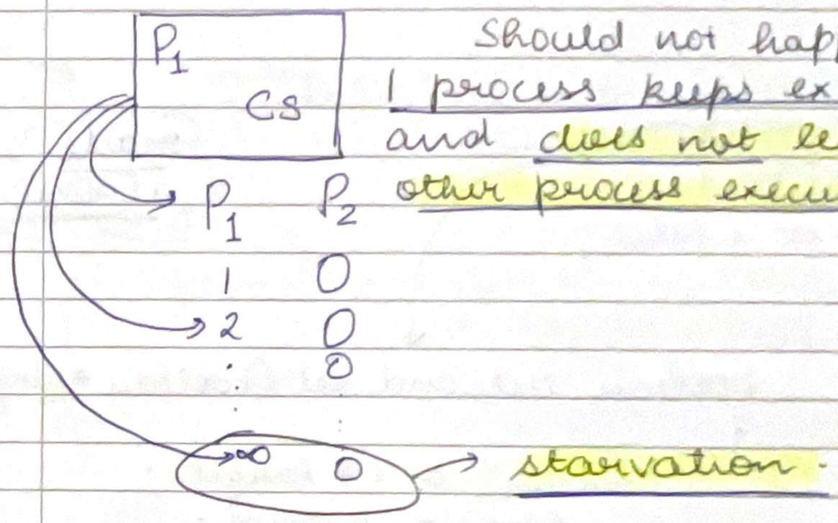
2) Progress



1 Process does not allow other one to execute.

* 1 entry code stoping from other process to start the entry code

3) Bounded Wait



Should not happen that 1 process keeps executing and does not let the other process execute

4) No assumption related to HW or Speed.

→ LOCK VARIABLE

In user mode

```
do{ acquire lock
    CS
    release lock
}
```

Lock == 0 / False
Vacant
Lock = 1 / True
Full.

No gaurantee of mutual exclusion.

⇒ Hardware level instruction

Test and Set Instruction

Entry Code	1. While (Lock == 1);
	2. Lock = 1
	3. Critical Section
Exit Code	4. Lock = 0

```
While (test_and_set (&lock));
   CS
lock = false;
```

made atomic

```
boolean test_and_set (boolean *target)
{
    boolean r = *target;
    *target = True;
    return r;
}
```

lock	Target	r
~~False~~	1000	False
1000		
True		True
True		

When it returns True it is waiting for previous process to finish

Turn Variable (Strict alternation)

* 2 Process sol
* user mode

int turn = 0/1

P_0	P_1
while (turn != 0);	while (turn != 1);
CS	CS
turn = 1;	turn = 0;

- ME ✓
- Progress ✗ → depends on starting value of turn variable.
- BW ~~✗~~ ✓

Semaphore → Binary / counting

* integer variable, used for mutual exclusion

P() , Down , wait } for entry section
V() , up , Signal } for exit section

Question: how do you find the value of the semaphore

S = 3, 2 } no of ~~resources~~ process that can go into CS
1, 0
-1, -2 } sleep, block, waiting

__/__/__

Mutex — Binary semaphore.

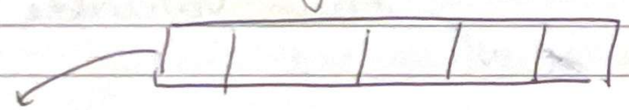
S = 0 or 1

Producer Consumer

* Solving by use of semaphore.

Starvation

* Prevent: Using FIFO



Wake up()
taken out by previous process

When previous process is executing signal() code

using Block() which is a basic System Call

→ Earliest deadline first

shortest remaining time first

→ Shortest Job First → Priority Scheduling
→ Round Robin → Multilevel queue scheduling
→ First come first Served → Rate Monotonic

→ Process Scheduling Algorithm → Real-time

Scheduling → moving process from ready queue to CPU

Pre-emptive → Reasons

Non-Preemptive

* full burst time is going to be executed

* Time Quantum: RR

* Priority.

Definitions:

1) Arrival time: Time at which process enters ready queue

2) Burst time / Duration: Time for process to get executed on CPU.

3) Completion time: When process ends.

4) Turn around time = CT - AT

5) Waiting time: TAT - BT

6) Response time: (time at which ~~CPU~~ a Process gets the CPU first time - Arrival time).

1) First come First Served

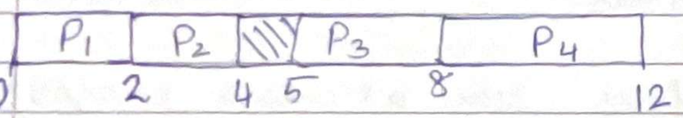
* Caiteria : Arrival time
* Mode : Non - preemptive

eg:

P.NO	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)	RT
P_1	0	2	2	2	0	0
P_2	1	2	4	3	1	1
P_3	5	3	8	3	0	0
P_4	6	4	12	6	2	2

Same since it is non-preemptive.

Grantt chart



* Convoy effect: Short process behind long process.

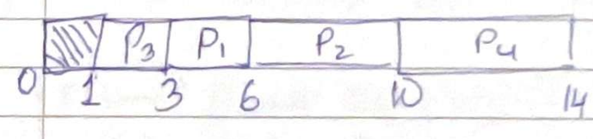
{ 1 CPU bound task
Many I/o bound task. }

Shortest Job First Non-pre

P.NO	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
P_1	1	3	6	5	2
P_2	2	4	10	8	4
P_3	1	2	3	2	0
P_4	4	4	14	10	6

Avg = $\frac{12}{4} = 3$

Grantt chart



here shortest job is completely executed and the next shortest job starts

* SJF is optimal → gives min avg WT
* difficulty: length of next CPU req

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

α, $0 \le \alpha \le 1$

usually 1/2.

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

Shortest Remaining time first

* Preemptive → Burst time.

P.N	AT	BT	CT	TAT	WT	RT
P1	0	~~5~~4	9	9	4	0
P2	1	~~3~~2~~1~~	4	3	0	0
P3	2	~~11~~	13	11	7	7
P4	4	1	5	1	0	0

R	P2	P2	P2	P4	P1	P3
0	1	2	3	4	5	9	13

{CPU first time ← AT}

P1 — 0 − 0
P2 — 1 − 1
P3 — 9 − 2
P4 — 4 − 4

Round Robin

* Criteria : Time Quantum
  Mode : preemptive

PNO	AT	BT	CT	TAT	NT	RT
P1	0	~~8~~2~~1~~0	12	12	7	0
P2	1	~~4~~2~~1~~0	11	10	6	1
P3	2	~~2~~0	6	4	2	2
P4	4	~~1~~0	9	5	4	4

Given (QT = 2)

Ready Q

Gnatt

Running

~~P1~~	~~P2~~	~~P3~~	~~P1~~	~~P4~~	P2	P1

P1	P2	P3	P1	P4	P2	P1	BASED ON QT

0 2 4 6 8 9 11 12

Context switching.

Priority Scheduling Preemptive

priority	PNO	AT	BT	CT	TAT	NT
low 10	P1	0	~~5~~4	12	12	7
20	P2	1	~~4~~3	8	7	3
30	P3	2	~~2~~1~~0~~	~~5~~4	2	~~10~~
high 40	P4	4	1	5	1	0

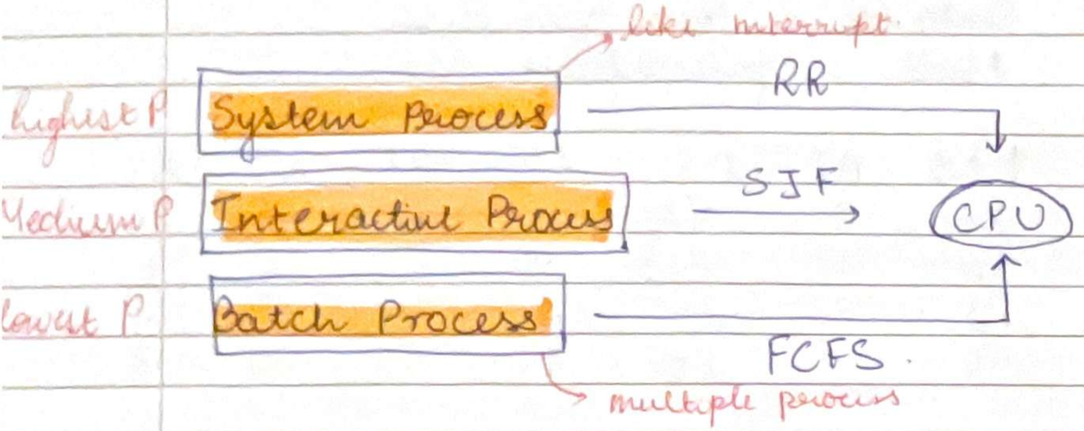
P1	P2	P3	P3	P4	P2	P1

0 1 2 3 4 5 8 12

given

* Starvation is a possibility

Multilevel Queue Scheduling

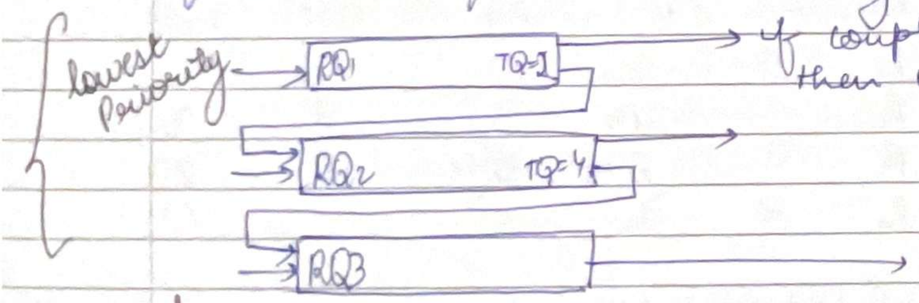


* Adv: Categorising process
* disadv: Starvation → solved by multilevel feedback

Multilevel feedback Queue

Aging implemented

* feedback is for lower priority queue.



FCFS manner

What is dispatcher?

* Ready queue → Partioned

RR → foreground (interactive)
FCFS → background (Batch)

High Priority → foreground.

After finishing all task move to next queue.

Time slice → Each queue gets certain amount of CPU time
→ 80% → RR → fore
20% → FCFS → background

Real - time scheduling

* Soft real - time → no gaurentee when process will be scheduled
* hard real - time → tasks must be serviced by deadline.

Imp

2 latencies → effect performance

1. interrupt latency → time from arrival of int to start ISR.
2. Dispatch latency
time to schedule to take current process off CPU and switch to another.

Conflict phases of dispatch latency

1) Preemption of process running in kernel mode.

2) Resources released by LPP for HPP.

Rate Monotonic

* A priority based on inverse of the period.

Short period → High Periority

long period → lower Periority

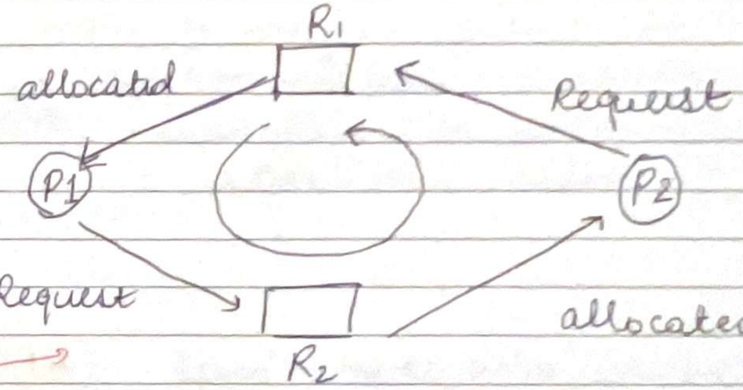
Earliest deadline first

Simple

can be solved if 1 resource prevents itself.

Deadlocks

* If 2 or more processes are waiting on happening of some event, which never happens, then the process involved are in deadlock.



P1 wants R2, R2 is occupied by R1
P2 wants R1, R1 is occupied by P1

4 conditions

1) Mutual Exclusion
2) No preemption
3) Hold and wait
4) Circular wait → loop kind of thing.

} necessary for deadlock

Resource of several type

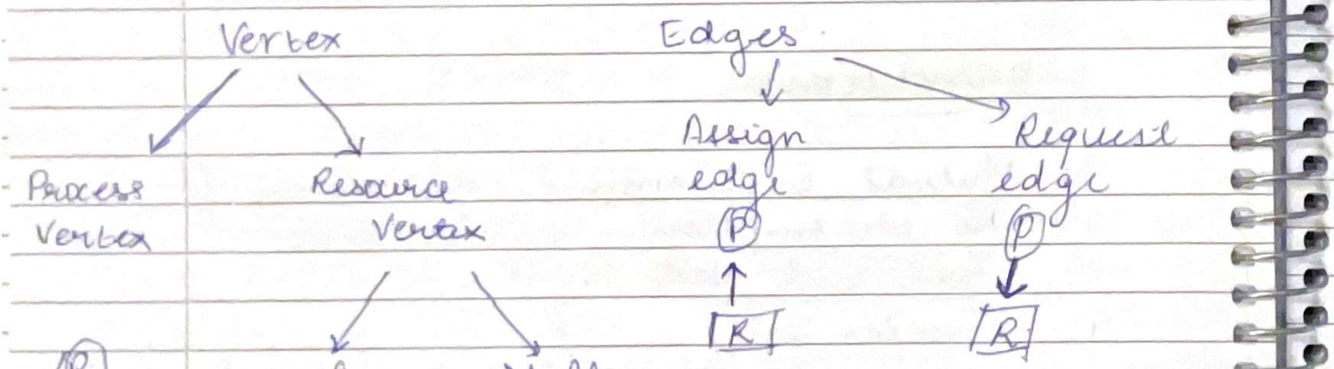
1) Serially Reusable Resource.
   eg CPU cycles, memory space, I/O devices
   acquire → use → release.

2) Consumable Resource.
   eg. Messages, buffers of information
   Create → acquire → use
   After this resource does not exist.

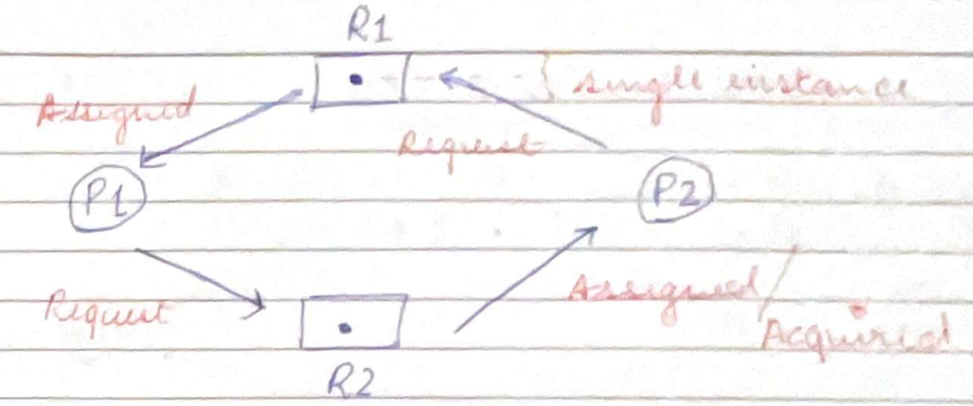
→ Resource allocation Graph. (RAG)

Vertex
- Process Vertex
- Resource Vertex
  - Single instance
  - Multi instance

Edges
- Assign edge
- Request edge



eg:

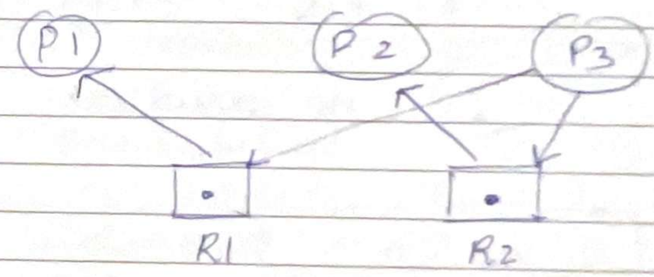


Since circular wait exists, there is a deadlock which means this RAG has a cycle.

	allocate		Req	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0

Availability (R_1, R_2) (0, 0)

eg:

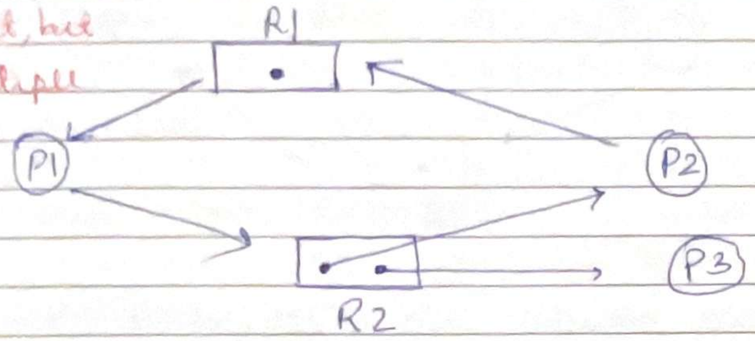


	allocate		Req		
	R1	R2	R1	R2	
P1	1	0	0	0	executed
P2	0	1	0	0	executed
P3	0	0	1	1	

Availability (0,0)
New (1,0) – After P1
New (1,1) – After P2

Multiinstance RAG

Cycle present, but this is multiple instances

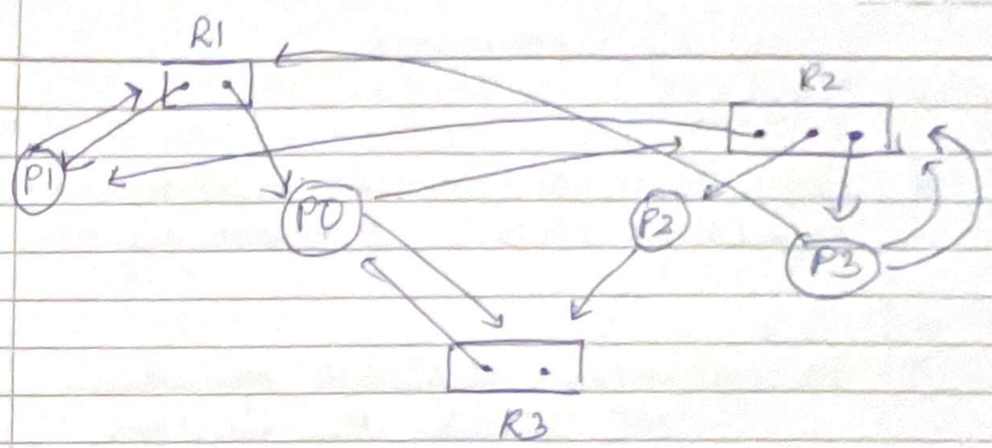


	allocate R1	allocate R2	Request R1	Request R2	
P1	1	0	0	1	
P2	0	1	1	0	
P3	0	1	0	0	X

Availability (0, 0)
~~P3 FIN~~ (0, 1)
P1 FIN (1, 1)

∴ no deadlock

SI && CW = DEADLOCK
MI && CW = POSSIBLE DEADLOCK



	allocate R1	allocate R2	allocate R3	K<=V R1	K<=V R2	K<=V R3	
P0	1	0	1	0	1	1	X
P1	1	1	0	1	0	0	
P2	0	1	0	0	0	1	X
P3	0	1	0	1	2	0	

~~availability (0,0,0) (0,0,0)~~

Availability (0, 0, 1) ✓
P2 end (0, 1, 1) ✓
P0 end (1, 1, 2) ✓
~~P3~~ (2, 2, 2)
~~P1 end~~ (2, 3, 2)
P1 end (2, 2, 2) ✓
P3 end (2, 3, 2)

No deadlock ✓

Deadlock handling

1) Deadlock Ignorance (ostrich)
2) Deadlock Prevention
3) Deadlock Avoidance
4) Deadlock detection and recovery.

1) deadlock ignorance

- ignore as it does not occur frequently
- If tackled, it will reduce speed and performance.

2) Deadlock Prevention

a) Mutual exclusion
To prevent mutual exclusion use shared variable / resources.
eg: Read - Only files

b) No preemption
Process holding resources can be preempted
Can use Time Quantum method.

c) No Hold & wait
- Take all resources in the beggining
- Release current resource before requesting for a new resource

d) No Circular wait

- Number the resources
- req in increasing order

If request Ri then release Rj for all ($j \geq i$)

3) Deadlock Avoidance

* Safe state: System can allocate resources to each process up to the maximum in some order to prevent deadlock.

System in safe state if there is a safe sequence.

If for each Pi the resources that Pi may still request can be satisfied by current resources.

→ Bankers Algorithm

* Deadlock avoidance technique
Deadlock detection can be possible
A = 10, B = 5, C = 7.

Process	Allocation			Max need			Available			Remaining		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
P4	2	1	1	4	2	2	7	4	5	2	1	1
P5	0	0	2	5	3	3	7	5	5	5	3	1
Total	7	2	5				10	5	7			

P2 get executed first
P4 gets executed
P5 gets executed
P1 gets executed
last P3 get executed.

Safe Sequence: P2 → P4 → P5 → P1 → P3.

Q

Process	Allocation			Max			Availability			Remain		
	E	F	G	E	F	G	E	F	G	E	F	G
P0	1	0	1	4	3	1	3	3	0	3	3	0
P1	1	1	2	2	1	4	5	3	1	1	0	2
P2	1	0	3	1	3	3	5	4	3	0	3	0
P3	2	0	0	5	4	1	6	4	6	3	4	1
							8	4	6			

P0 → P2 → P1 → P3 | P0 → P2 → P1 → P3

Total Resource + Total Process > Total demand

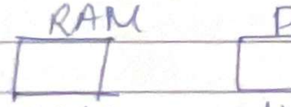
→ Recovery from deadlock

1) Terminate the process. { policy decision
2) Resource preemption
   ↳ selecting a victim
   ↳ Rollback

→ Memory Managment

* Main memory needs to be managed 'RAM'

* We need to ↑ degree of multiprogra to ↑ CPU utilisation



$\frac{4}{4}$ = 1 process at a time

If 'K' probability of I/O op

Then CPU utilisation is

$(1 - K)$

eg: K = 70

RAM	Process	K (assume)	1 - K
4	1 (4MB)	70%	$(1-K) = 30\%$
8	2 (4MB)	$K^2 = (.7)^2$	$(1-K^2) = .76$
16	4 (4MB)	$K^4 = (.7)^4$	$(1-K^4) = 93\%$

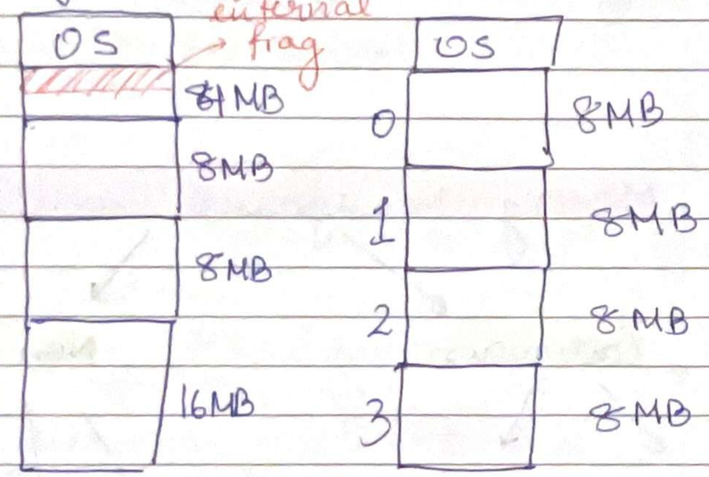
Memory Managment Techniques

- Contiguous
  - Static
    - Fixed Partition
  - Dynamic
    - Variable Partition
- Non-Contiguous
  - Paging
  - Multilevel Paging
  - Inverted Paging
  - Segmentation
    - Segmented paging

Fixed Partitioning (Static)

* No of Partitions are fixed
* Size may vary.
* Spanning is not allowed.

P1 = 2MB



* Internal fragmentation
  → space or size being wasted since 1 partition did not occupy the whole partition

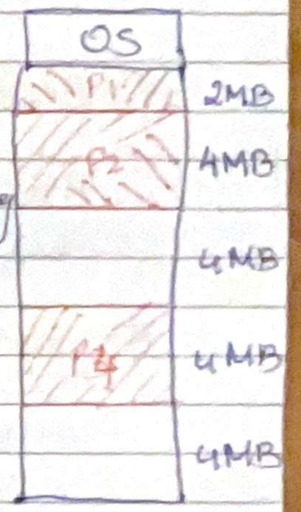
* Limit in Process size.
  eg: 32MB in partition of max size 16MB

* Limit to degree of Programming

* External frag
  → When there is space in the memory, but it is divided due to contiguous memory allocation and internal fragmentation

Variable Partitioning

P1 = 2MB
P2 = 4MB



* No internal fragmentation
* high degree of multiprogramming
* No limit to Process size.

* External frag still exists.
  P6 = 8MB
  Can not be split

We can use compaction, moving free space together but not suggested.

* Allocation & dealloocation is complex

Since we can not control no of process getting allocated or deallocated

Algorithm for variable partitioning.

* **First - Fit**
  Allocate the first hole that is big enough — Simple / fast

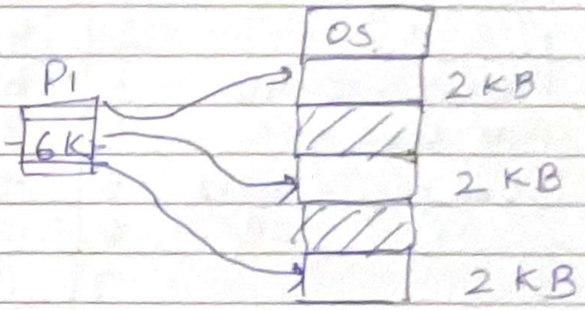
* **Next - fit**
  Same as first fit but start search from last allocated hole.

* **Best - fit**
  → slow / small hole left.
  Allocate that hole that gives minimal internal fragmentation

* **Worst - fit**
  — slow
  Allocate to that hole where the internal frag is maximum.

→ Non - Contiguous — Compaction

* Process can be divided



* External memory fragmentation does not occur here.

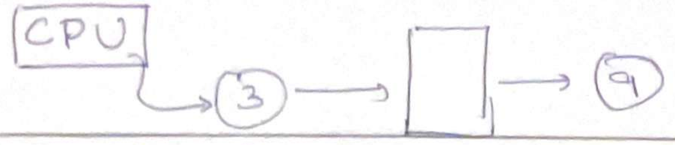
Paging

* The process is divided into pages (not compared to the frame size) in the secondary memory itself.
* Main memory is divided into frames

Size of Page = Size of frame

* Mapping done by Memory management unit

MMU



Page table is owned by every page

page no → Process

0	0 1
1	2 3

Process size = 4B
Page size = 2B
No of page = $\frac{4B}{2B}$ = 2

frame no →

0	0 1	RAM
1	2 3	
2	4 5	P_0
3	6 7	
4	8 9	P_1
5	10 11	
6	12 13	
7	14 15	

Main Mem = 16B
Frame Size = 2B
No of frame = $\frac{16B}{2B}$ = 8 frames.

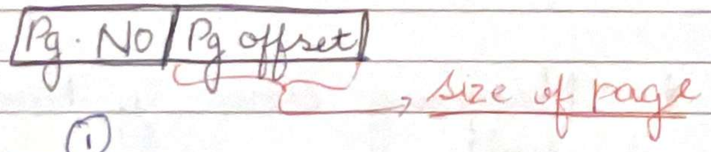
Page table of P1

0	f2	→ {frame 2}
1	f4	{frame 4}

P1

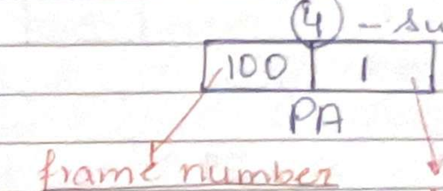
CPU works on Logical address

Logical address consist of 2 parts.



for P1
pg no & offset can be rep by 1 byte each

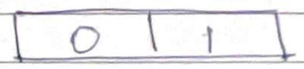
physical address → Main memory

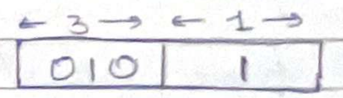


– since MM was 16B
can be rep by 4B

eg. CPU wants Byte no 1

0	f2
1	f4





To map logical address to physical address we need page table.

Quest:
2^{10} = 1M
2^{30} = 1G
2^{40} = 1T

LAS = 4GB (size of Process), LA = $2^2 \times 2^{30} = 2^{32}$
PAS = 64MB = $2^6 \times 2^{20} = 2^{26}$
Pg size = 4kB = $2^2 \times 2^{10} = 2^{12}$
No of pages = 2^{20}
No of frames = 2^{14}
No of entries in pg table = 2^{20}
size of pg table = $2^{20} \times 14$ bits

20	12

— 32 —

14	12

— 26 —

No of pages = No of entries in pg table

Q LA = 7bits
PA = 6bits
Pg size = 8words/byte. = 2^3

4	3	
3	3	PA

No of pages = 2^4
No of frames. = 2^3
No of entries = $2^4 \times 3$

$2^4 \times 2^3$

Page table

frame number	Valid(1)/Invalid(0) (Present/absent)	Protection R W X	Reference LRU	Caching (enable/disable)	Dirty modify

Mandatory field

Page fault → Read, Write, Execute

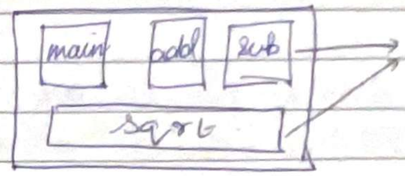
least recently used

~~Multilevel Paging~~

Segmentation

* Divide process into parts.
* Works on user point of view.

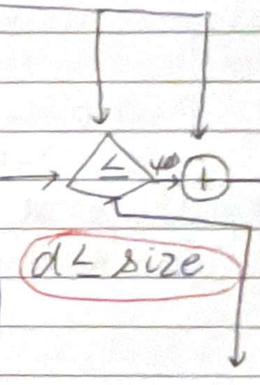
It will create segments according to the user



* Can be of different sizes.

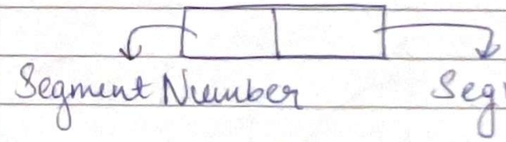
CPU → d

	BA	Size
0	3300	200
1	1800	400
2	2700	600
3	2300	400
4	2200	100
5	1500	300



	OS
1500	S_5
1800	S_1
2200	S_4
2300	S_3
2700	S_2
3300	S_0
3500	

CPU → LA



frame no | valid invalid | RWX Protection | Refernd LRU | Caching | dirty modify

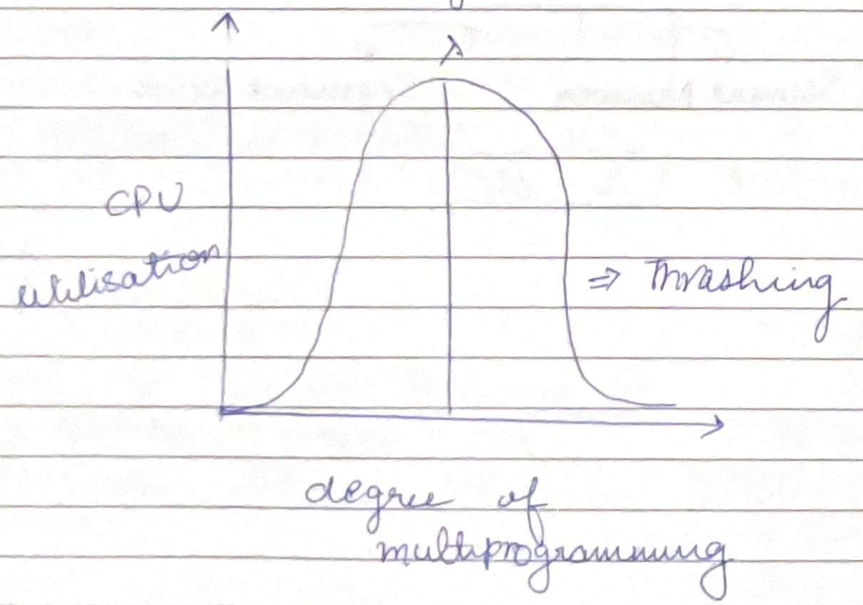
→ Thrasing

* linked to degree of multiprogramming

We bring only 1 or 2 pages of a process into RAM, which allows us to bring other pages of process which in turn helps degree of multiprogramming.

But if suppose RAM wants Proc1 pg 3 which is in Disk then it would take more time since it encountered page fault.

* Multiple page fault will cause thrashing.



How to prevent page fault

1) Increase size of RAM

2) Reduce speed of long term scheduler or limit degree of multiprogramming

→ Overlay - Memory Management.

* Methode by which a large size process can be put in main memory.

Main memory size < Process size

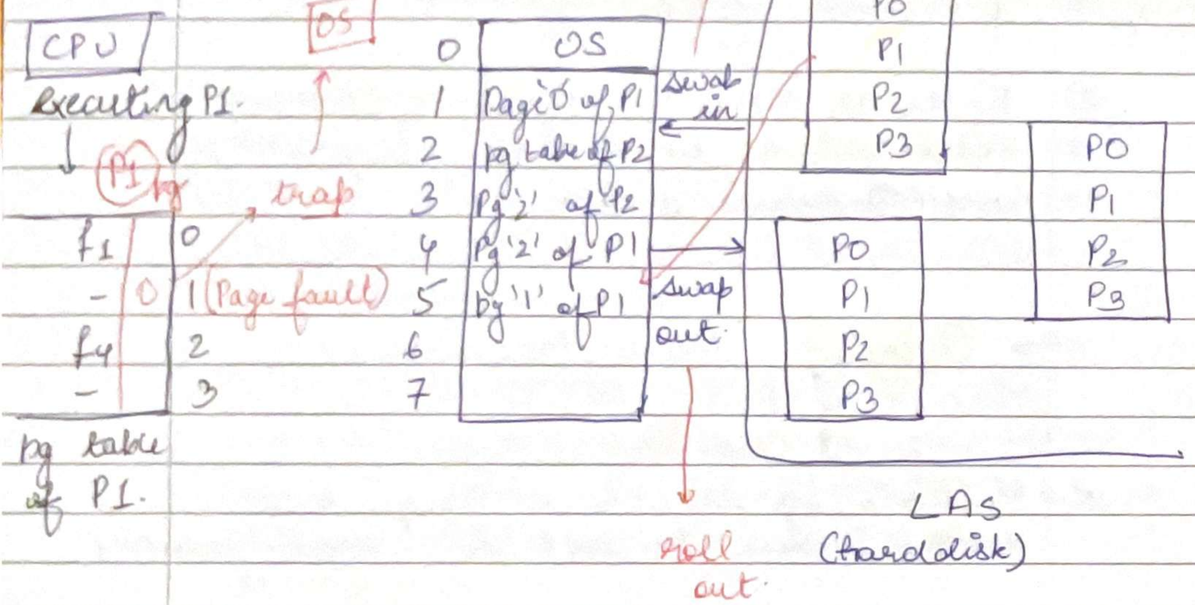
* Mostly used in embedded Systems

Thrashing

→ In order to ↑ degree of multi-programming, no of pages are increased in main memory, which may cause page faults.

excise page faults can cause decrease in multiprogramming and CPU utilisation.

Virtual Memory



* Gives user the illusion that main memory has space.
* No limit to number of processes and size of the process.

If page fault occurs, control is given from user to OS

1. OS will check authentication.

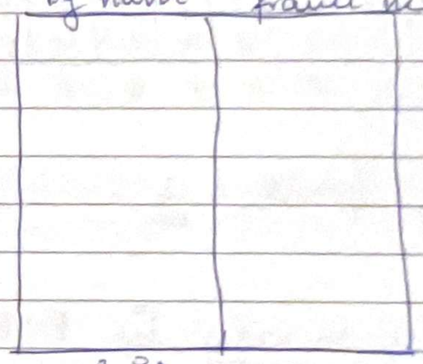
Effective memory access time

EMAT = P(page fault service time)$_{ms}$

= P + (1-P)(Main memory access time)$_{ns}$

$$EAT = P + (1-P)(MAT)$$

→ Translational lookaside buffer



$$EAT = (TLB + x) + Miss(TLB + 2x)$$

Main memory access time

Q TLB = 10ns MAT = 50ns
hit ratio = 90%

EMAT = .90 x (10+50) + (.10)(10+100ns)

= .9 x 60 + .1 (110ns)
= 54 + 11
= 65ns

→ Page replacement algorithm.

1) FIFO
2) optimal pg replacement
3) least recently used.

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0

f3			1	1	1	1	0	0	0	3	3	3	3	2	2			
f2		0	0	0	0	3	3	3	2	2	2	2	1	1	1			
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0			
	*	*	*	*	H	*	*	*	*	*	*	H	*	*				

Pg fault = 12

Beladys anomaly in FIFO

Ref: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Anomaly

f1	1	1	1	4	4	4	5	5	5	5	5	5
f2		2	2	2	1	1	1	1	1	3	3	3
f3			3	3	3	2	2	2	2	2	4	4
	*	*	*	*	*	*	*	H	H	*	*	H

Belady's Anomaly

f1	1	1	1	1	1	1	5	5	5	5	4	4
f2		2	2	2	2	2	2	1	1	1	1	5
f3			3	3	3	3	3	3	2	2	2	2
f4				4	4	4	4	4	4	3	3	3
	*	*	*	*	H	H	*	*	*	*	*	*

Optimal page replacement (Pg replacement)

Replace page which is not used in longest dimension of time in future

Ref string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	7	7	7
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1
f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	*	*	*	*	H	*	H	*	H	H	H	H	H	*	H	H	H	*	H	H

Which ever value is not req for a while replace that.

longest time

Least Recently used — LRU

Ref → 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	7	7	3	3	3	3	3			3	3	3	3	3	7	7	
	0	0	0	0	0	0	0	0	0			0	0	0	0	0	0	0	
		1	1	1	1	1	4	4	4			4	1	1	1	1	1	1	
			2	2	2	2	2	2	2			2	2	2	2	2	2	2	
*	*	*	*	H	*	H	*	H	H	H	H	H	*	H	H	H	*	H	H

7 was used the least ∴ replace 7

replace page used longest time in past

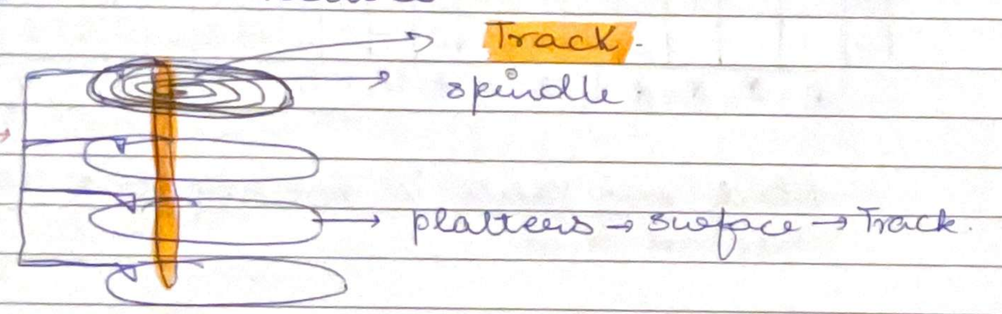
pg hit = 12

Most recently used.

* Replace the most recently used page in Past.

Hard Disk

→ Disk Architecture.



* Moves clockwise or anti, but can not move both

Platters → surface → Track → Sectors → Data

Disk size : P x S x T x S x D

eg: 8 x 2 x 256 x 512 x 512 KB

$= 2^3 \times 2^1 \times 2^8 \times 2^9 \times 2^9 \times 2^{10}$ B

$= 2^{40}$ B

= 1TB

No of bits req to represent disk size : 40

→ Disk Access time

* Seek time : Time taken by R/W head to reach desired track

* Rotation time : Time taken for 1 full rotation

* Rotation latency : Time taken to reach to desired sector
(Half of rotation time)

* Transfer time : Data to be transfer / Transfer rate

Transfer rate : (No of heads) x Capacity of 1 track x No of rotations in 1 sec

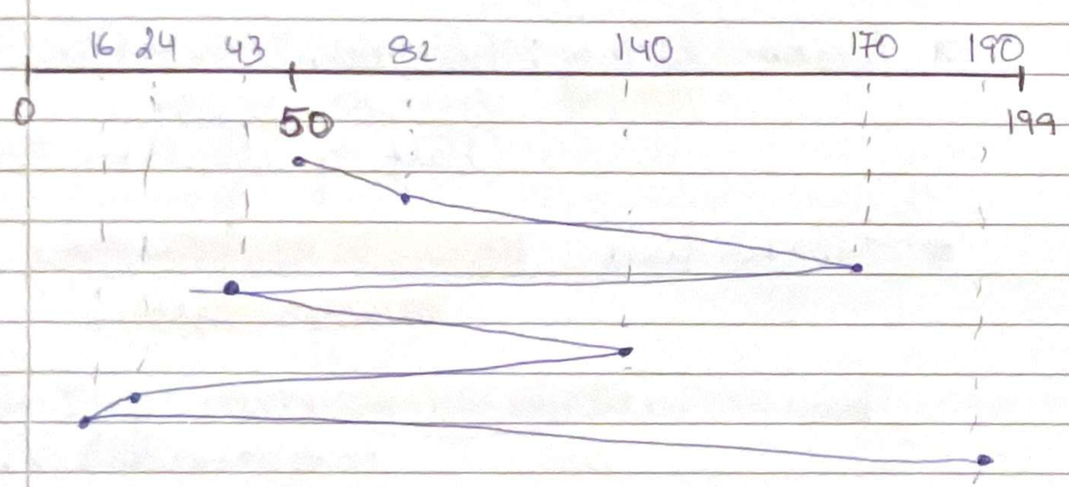
surface

DAT = ST + RT + TT + CT (optional)

→ Disk Scheduling Algorithms

Minimising Seek time

1) FCFS
- Req Que : 82, 170, 43, 140, 24, 16, 190 (200 tracks)
- curr pos : 50
- Cal total no of tracks movement by R/W head



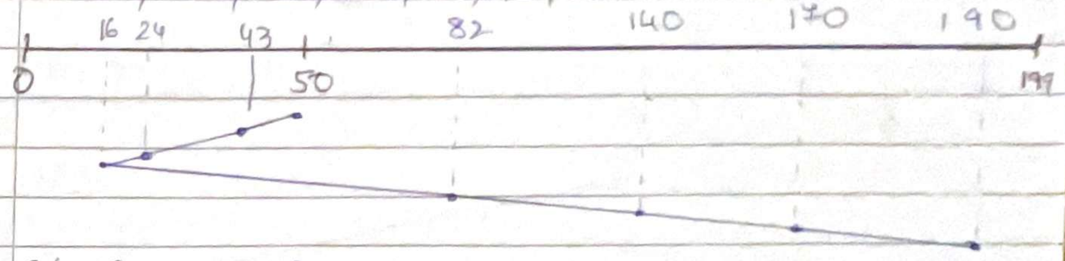
$= (170-50) + (170-43) + (140-43) + (143-16) + (190-16)$

$= 642$

* No starvation for this algo.

→ Shortest Seek time first (SSTF)

82, 170, 43, 140, 24, 16, 190



R/W head takes 1ns to move b/w tracks

$TT = (50-16) + (190-16)$

$= 34 + 174$

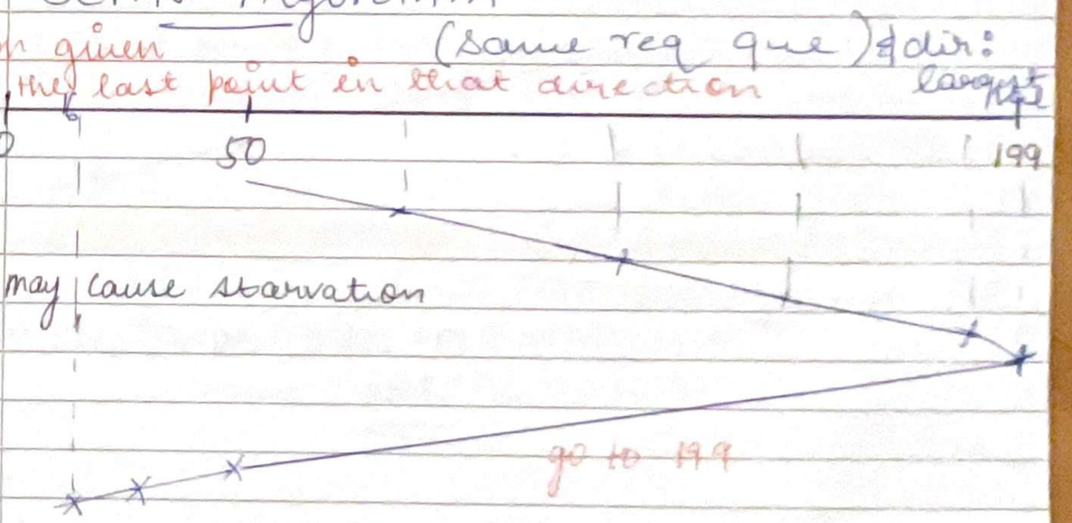
$= 208 \times 1ns = 208\ ns$

* High chance of starvation
* Generates overhead (complexity ↑)

→ SCAN Algorithm

direction given — (same req que) & dir: largest

move to the last point in that direction



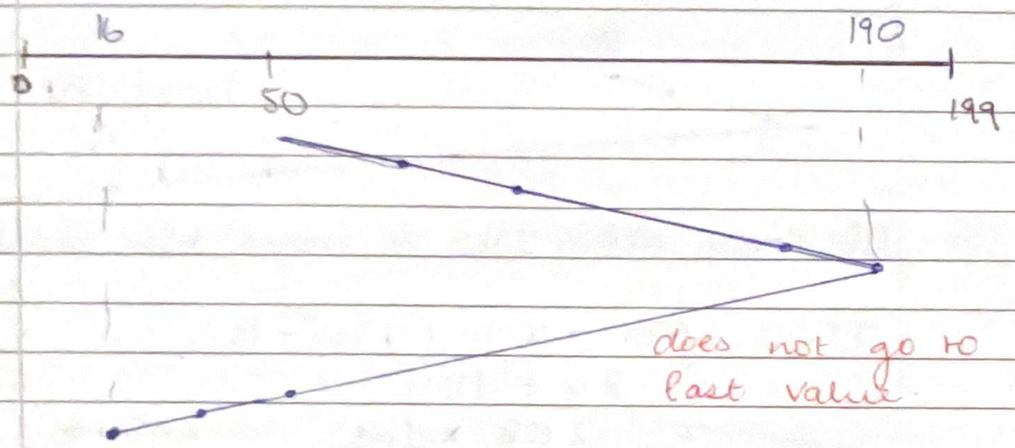
* may cause starvation

Since 16 is last value stop here

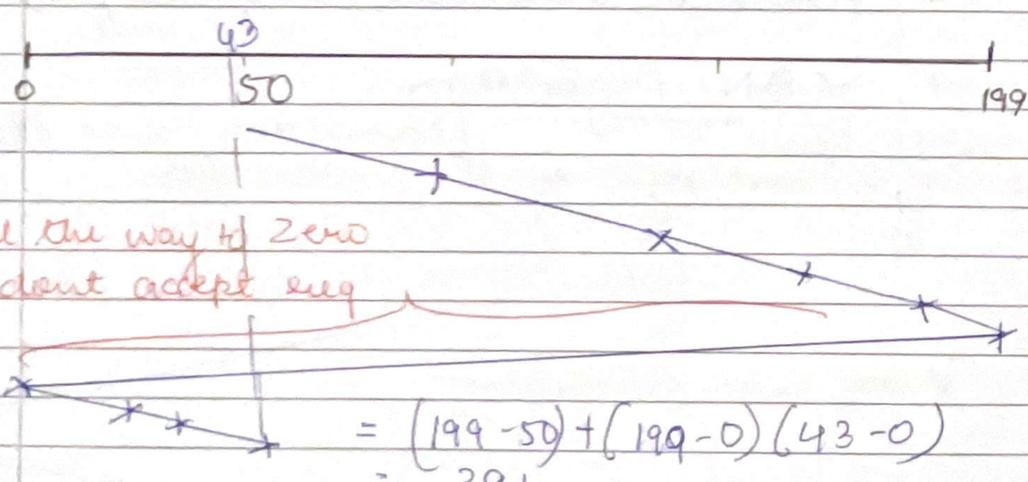
$= (199-50) + (199-16)$

$149 + 183 = 332.$

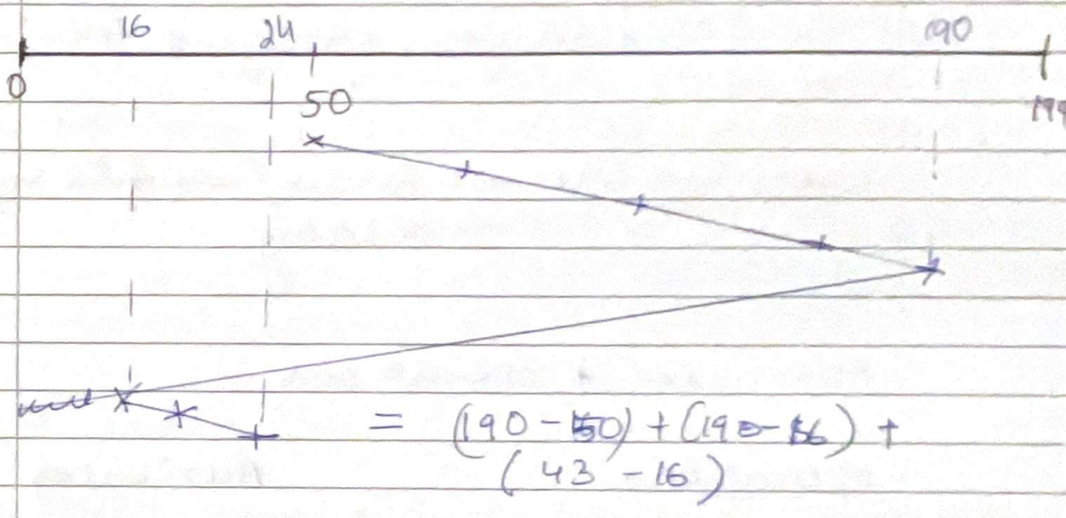
→ Look → Direction large



→ C-SCAN
→ Direction large



→ C-LOOK
→ Direction



* SCAN & C-SCAN are for heavier loads
* default can be SSTF & LOOK

Sector
Data header error-correction code

Virtual memory uses disk space as an extension of main memory. } space / swap

RAID
redundant array independent disks
→ reliability via redundancy
→ ↑ mean time to failure
mean time to repair } Mean time to data loss

→ File System

↓

Software

↳ manages storage & fetching

user → file → folder/directory → file system

Attributes & operations.

operations

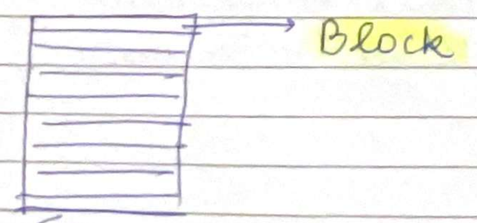
6 Important operations

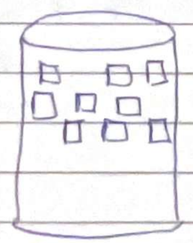
1) Creating
2) Reading
3) Write
4) deleting
5) truncate
6) Repositioning

Attributes.

1) Name
2) Extension
3) Identifier
4) Location
5) Size
6) Modified data
7) Protection
8) Encryption

Allocation Method





Allocation

Contiguous

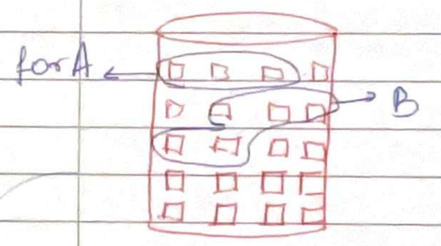
Non Contiguous Alloc
- linked list
- Indexed

* Allocation helps with efficient disk utilisation { Prevent fragmentation internal or External }

* Access should be faster

→ Contiguous

* Data stored in a continuous manner



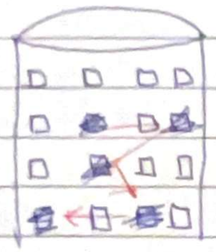
Directory

fil	start	leng
A	0	3
B	6	5
C	14	4

Advantage
- easy implement
- excellent read performance

Disadvantage
- Internal frag
- External frag
- difficult to grow file.

→ Linked List allocation.



file	start
F_1	2

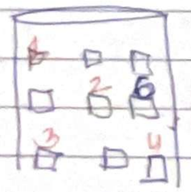
Pointer — Data and pointer both are stored

last block will have pointer value as -1.

Adv
- no external frag.
- file can grow

disadv
- large seek time
- Random access
- Overhead of pointers.

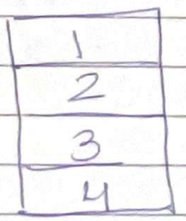
→ Indexed allocation.



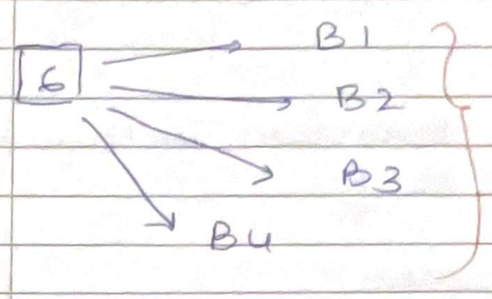
HD

Directory

file	Index Block
A	6



1 block is made as an index such as

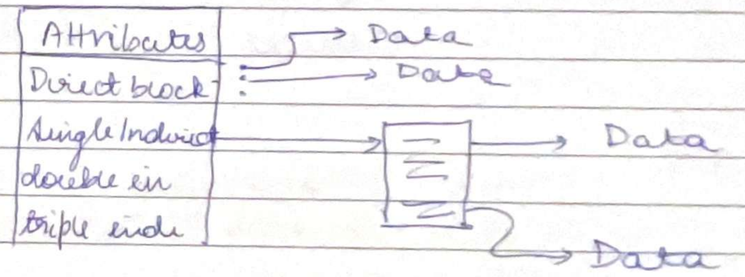


Advantages
- direct access
- no external frag

Disadvantages
- Pointer overhead
- Multilevel index.

→ unix Inode structure.

Index → block.



Q 8 direct block
1 double
1 single
1 triple

Size of each disk block = 128B
Size of each block address = 8B.

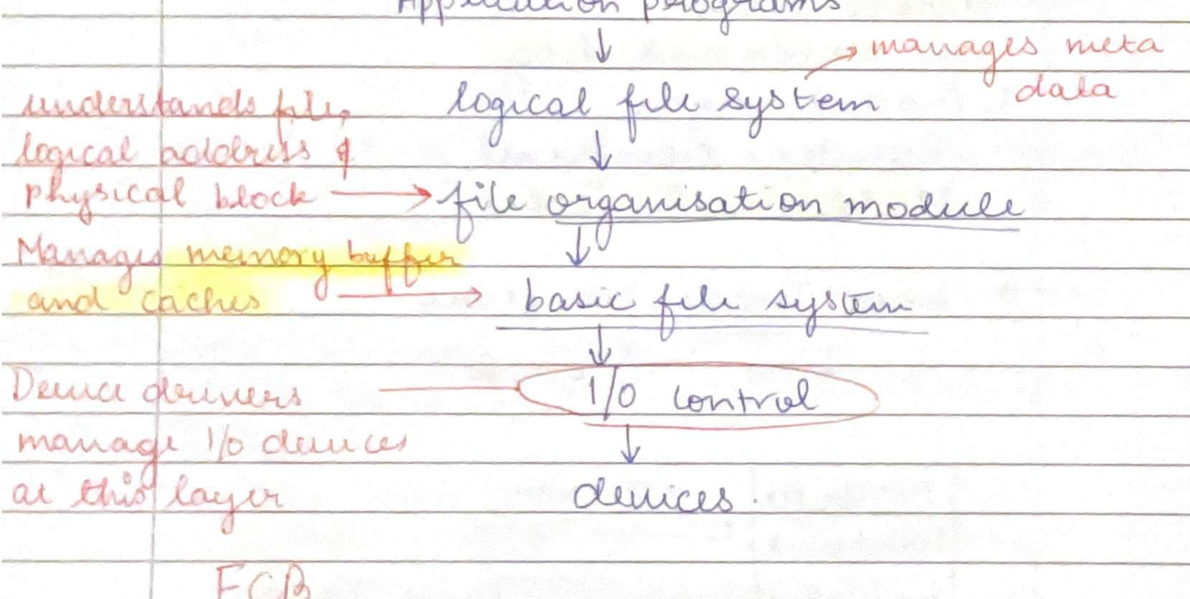
$\frac{128\,B}{8\,B} = 16$

Max file size $= (8 + 16 + (16 \times 16) + (16)^3) \times 128$

Information Associated with the open file
* file pointer, file-open count, access rights, disk location of file — permission

File control block : structure containing information about the file

Layered file System



FCB

file permissions
file dates (create, access, write)
file owner
file size
pointer to data blocks

Ready
Q (1) paging problems
(2) file system mounting
Gantt

Questions & doubts