Semester 6
CS341

05

Ch3: Synchronization

Process
- Cooperative
- Independent

→ Cooperative process share variables / memory / code / resources.

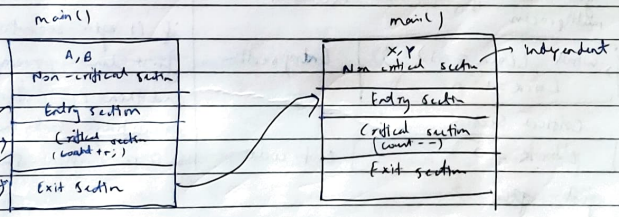
```
int shared = 5; // global variable shared by P1 & P2

P1                          P2
int x = shared (5)          int y = shared (5)
x++ (6)                     y--
sleep(1)                    sleep(1)
shared = x (6)              shared = y (4)
```

Race condition:
If P1 executes first, shared = 4
If P2 first, then = shared = 6
Requirement shared = 5

// the above two processes run on a uniprocessor, so they run concurrently (not parallelly)

* **Critical section**: part of program where shared resources are accessed by various processes.
(eg section containing global count variable)
Hence we need to synchronize them

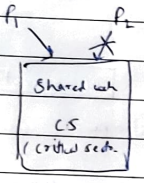


* 4 conditions: (to achieve synchronization)

i) Mutual Exclusion } Primary (must)
ii) Progress }
iii) Bounded wait
iv) No assumption related to hardware speed

* **Mutual Exclusion**

If P_1 enters critical section then the entry section of P_2 won't allow P_2 to enter the critical section.



(Both cannot enter the critical section at the same time)

* **Progress :**

P_1 wants to progress (enter the critical section) but P_2 doesn't allow P_1 to enter the critical section or vice versa, hence no progress can be made

* **Bounded Wait :**

P_1 uses the critical section and then again uses it before P_2 can use. This process continues (P_1 keeps executing critical section & P_2 keeps waiting (starvation)) ⇝ unbounded wait

Hence this shouldn't be the case (wait should be bounded)

i) **Lock Variables :**

→ (Busy wait mode (no extra required))

→ no guarantee of mutual exclusion

→ for multiprocess

```
while (LOCK == 1) ;      Entry section
    LOCK = 1 ;
Critical Section ;
    LOCK = 0 ;           Exit code
```

LOCK = 0 (initially room is CS)

if P_1 after execution of first line preempts, and then P_2 starts executing both will be in the critical section

ii) **Test and Set :**

→ Combine the entry section into one line

→ Mutual execution ✓ Progress ✓ Bounded wait ✗

→ Hardware based solution

All the methods are to prevent race condition which may cause loss of data, deadlock, etc.

Peterson's sol^n → turn
compare_and_swap → lock

```
while (test_and_set(&lock));   // entry
CS                             // critical section
lock = false;                  // exit

bool test_and_set(bool *target)
{
    bool r = *target;
    *target = TRUE;            // now lock becomes true
    return r;
}
```

Peterson's sol^n
→ software based
→ all conditions satisfies
→ turn, flag[2] → shared variables

iii) Turn Variable (Strict Alternation)
→ 2 process solution → Mutual exclusion ✓
→ run in user mode → Progress X
→ Bounded wait ✓

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

iv) Semaphore → Counting (integer) (−∞, ∞)
Binary {0, 1}

→ Semaphore is an integer variable used in mutually exclusive manner by various concurrent cooperative process in order to achieve synchronization.

P() , Down , Wait → entry section code
V() , Up , Signal (Post / Release) → exit section code

classmate
Date
Page

→ // Entry code

```
Down ( Semaphore S)
{  S.value = S.value - 1;
   if (S.value < 0)
   {  put process (PCB) in suspended list (sleep())
   }
   else
   return ; // go to critical section
}
```

// Exit code

```
Up ( Semaphore S)
{  S.value = S.value + 1;
   if (S.value ≤ 0)
   {  select a process from suspended list
      Wake Up() // move to ready queue (not critical section)
   }
}
```

→ If S is initialized with 1, at most 1 process can be executing the critical section.
(if S = 10, then atmost 10 processes can be in the critical section)

* Binary Semaphore : (Initialized with S = 1)

```
Down ( Semaphore S)
{  if (S.value == 1)
   {  S.value = 0; // can go to critical section
   }
   else
   {  Block the process & place in suspended list (sleep());
   }
}
```

```
Up ( Semaphore S)
{  If (Suspended list is empty)
   {  S.value = 1;
   }
   else
   {  Select a process from suspend list & wake up();
   }
}
```

* Problems solved:

i) Producer Consumer
ii) Readers writers (R, W) (W, R) (W, W) (R, R) → NP
iii) Dining Philosophers
(deadlock may occur if all philosophers pick 1st left & then right fork)
Sol^n → make any 1 philosopher pick right fork then left fork

Applicat' process

atomic (non interruptable)

Mutex locks — acquire() lock, then release()

Sol^n requires busy waiting

The lock is called spin lock

```
acquire() {
    while (! available)
        ; // busy waiting
    available = false;
}

release() {
    available = true;
}
```

→ do {

acquire lock

C.S

release lock

} while (1)

& Semaphore: ← Busy waiting implementation

Binary semaphore → same as mutex

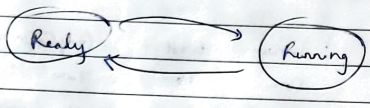
```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

signal (S) {
    S++;
}
```

Ch 2: Scheduling Algorithms

classmate
Date
Page

* Preemptive:



→ Process can go from running queue to ready queue
→ for eg: due to time quantum or priority
→ To increase ~~response~~ responsiveness or give priority to most imp. job.

* Preemptive

→ SRTF (shortest remaining time first)
→ LRTF (Longest remaining time first)
→ Round Robin
→ Priority based (more common)

Non-preemptive

→ FCFS (first come first serve)
→ SJF (Shortest Job first)
→ LJF (longest Job first)
→ HRRN (Highest respon ratio next)
→ Multilevel Queue
→ Priority based

* Time terminology:

i) Arrival time: Time at which process enter the ready queue
ii) Burst time: Time required by a process to get executed on CPU
iii) Completion time: Time at which process completes its execution

duration
Point of time

iv) Turn around time = { completion time − arrival time }
v) Waiting time = { Turn around time − Burst time }
vi) Response time = { (Time at which process gets CPU first time) − (arrival time) }

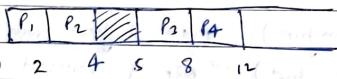
In case of Non-preemption,
WT = RT always

i) FCFS

	Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
(i)	P_1	0	2	2	2	0	0
(ii)	P_2	1	2	4	3	1	1
(iii)	P_3	5	3	8	3	0	0
(iv)	P_4	6	4	12	6	2	2

Gantt Chart (running queue);

Criteria : Arrival time
Mode : Non-preemptive
(TAT = CT - AT) (WT = TAT - BT)

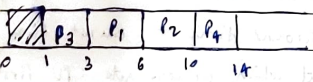


ii) SJF :

	Process No.	Arrival Time	Burst time	Completion Time	TAT	WT	RT
(ii)	P_1	1	3	6	5	2	2
(iii)	P_2	2	4	10	8	4	4
(i)	P_3	1	2	3	2	0	0
(iv)	P_4	4	4	14	10	6	6

Gantt Chart:

Criteria = "Burst time"
Mode = Non preemptive



P_1P_3 → P_3 chosen
P_1P_2 → P_1 chosen

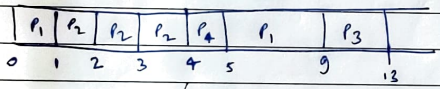
P_2P_4 → same burst time (tie)
⇒ based on arrival time → P_2 chosen
If same arrival time also, then based on process id

iii) Shortest Remaining ~~Time~~ Time first (SJF + preemption)

Process No.	Arrival Time	Burst Time	Completion time	TAT	WT	RT
P_1	0	~~5~~ 4 0	9	9	4	0
P_2	1	~~3~~ 2 ~~1~~ 0	4	3	0	0
P_3	2	~~4~~ 0	13	11	7	7 (9-2)
P_4	4	~~1~~ 0	5	1	0	0

(i)

Gantt Chart:



↳ Now all have arrived

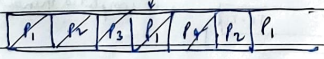
Criteria : Burst time
Mode : Preemption
RT = { CPU first time - AT }
Avg RT = 7/4

iv) Round Robin scheduling :

Process No.	Arrival Time	Burst time	Completion Time	TAT	WT	RT
P_1	0	~~5~~ 3 ~~1~~ 0	12	12	7	0
P_2	1	~~4~~ 2 ~~1~~ 0	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~ 0	9	5	4	4

Given time quantum = 2

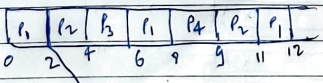
Ready Queue



inserted at end if first insert others who have arrived

Criteria : Time Quantum
Mode : Preemption

Running Queue



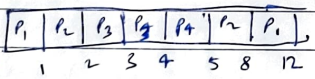
P1 P2 P3 ← context switch

Criteria : "Priority"

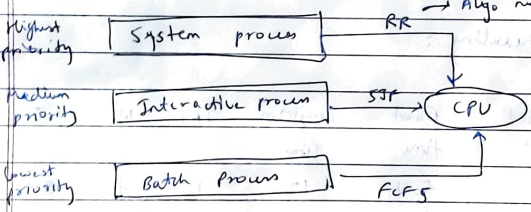
v) Preemptive Priority scheduling. Mode : "Preemptive"

Priority	Process No.	Arrival Time	Burst Time	Completion time	TAT	WT
10	P_1	0	5	12	12	7
20	P_2	1	4	8	7	3
30	P_3	2	2	4	2	0
40	P_4	4	1	5	1	0

In this case, higher the no, higher the priority (given in question)



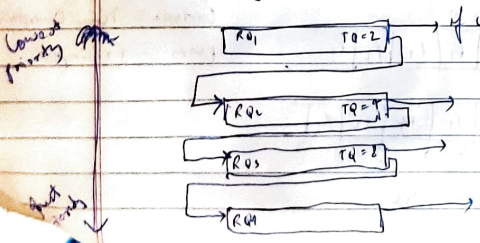
vi) Multilevel Queue scheduling :



→ Multiple queues (ready queue) exist for different type of process

→ Starvation (of lower priority jobs) is a problem

vii) Multilevel feedback Queue

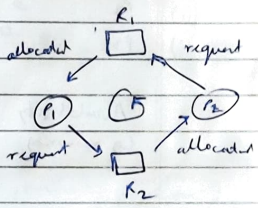


Ch 4 : Deadlock

* **Deadlock :**

→ if two or more processes are waiting on happening of some event which never happens, then we say these process are involved in deadlock & that state is called deadlock state.

→ eg: R_1 is allocated to P_1 which waits for resource R_2 in order to be executed but R_2 is allocated to P_2 which waits for R_1 to be executed



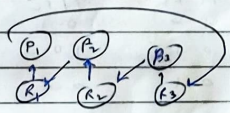
→ Necessary (conditions(and sufficient)

i) Mutual exclusion (all resources must be used (shared) one by one)

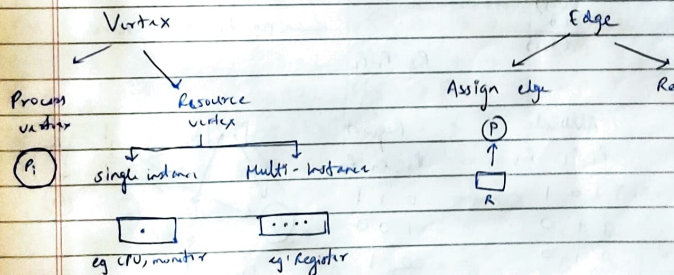
ii) No preemption

iii) Hold and wait (keep on holding on resource while waiting for other

iv) Circular wait



→ Resource Allocation Graph (RAG) :

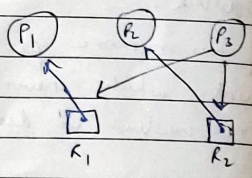


→ If single instance resource & ~~there~~ RAG has circular wait (cycle) ⟺ Always deadlock

→ Deadlock (infinite waiting ~~time~~) Starvation (finite waiting ~~time~~)

eg

		Allocate R_1	Allocate R_2	Request R_1	Request R_2
(i)	P_1	1	0	0	0
(ii)	P_2	0	1	0	0
iii)	P_3	0	0	1	1



→ No deadlock (acyclic)

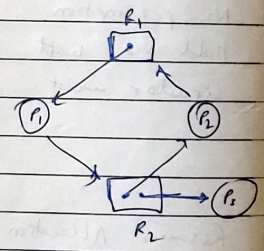
Availability (0,0) $\xrightarrow{P_1\text{ done}}$ (1,0) $\xrightarrow{P_2\text{ done}}$ (1,1) $\xrightarrow{P_3\text{ done}}$
(q resources)

* Multi instance resource allocation :

→ Multi instance ~~def~~ circular wait ⇒ always deadlock (false)

eg

		Allocate R_1	Allocate R_2	Request R_1	Request R_2
(ii)	P_1	1	0	0	1
(iii)	P_2	0	1	1	0
(i)	P_3	0	1	0	0



Curr avail (0,0)
+ 0 1
(0,1)
+ 1 0
(1,1)

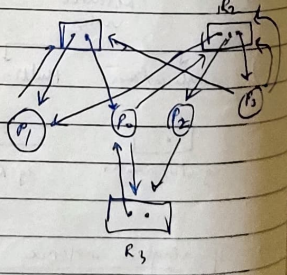
(Multi-instance RAG)

→ no deadlock here

P_3 P_1 P_2

eg

	Allocate R_1	Allocate R_2	Allocate R_3	Request R_1	Request R_2	Request R_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0



$P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

Curr Avail : (0,1,0,1)
0 1 0
0 1 1
1 0 1
1 1 2
1 1 0 → 2 2 2

1, 3, 4, 2, 3
1, 3, 4, 2, 0

classmate
Date
Page

* Methods to handle deadlock:

i) Deadlock ignorance (ostrich model)
(deadlock is rare, when occurs, restart) → most used (eg windows, linux)

ii) Deadlock prevention
(making one of the 4 conditions false)

iii) Deadlock avoidance (Banker's Algo)
(can also detect deadlock)

iv) Deadlock detection & recovery
(first detect & if found recover (eg kill the process or resource preemption))

* Banker's Algo:

→ if deadlock detected, unsafe, nothing to do
→ if no deadlock (safe), find the safe sequence (of process)
→ We need to know the max allocation for each process
(max resource instance required in advance (not practical) (static))

Process	Allocation (resources) A	B	C	Max Need A	B	C	Current Available A	B	C	Remaining Need A	B	C	Max Need − allocation
P_1	0	1	0	7	5	3	3	3	2	7	4	3	
P_2	2	0	0	3	2	2	5	3	2	1	2	2	
P_3	3	0	2	9	0	2	7	4	3	6	0	0	
P_4	2	1	1	4	2	2	7	4	5	2	1	1	
P_5	0	0	2	5	3	3	7	5	5	5	3	1	
	7	2	5				10						

If none of the process can fulfil its need, then deadlock

Total: A = 10 (CPU), B = 5 (memory), C = 7 (Printer)

Available = A: 10−7 = 3, B: 5−2 = 3, C: 7−5 = 2

Safe sequence: $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

Q) A system is having 3 processes each requiring 2 units of resource R. Find minimum no. of units R such that no deadlock occurs.

R = 3 P_1 P_2 P_3 ✗

R = 4 P_1 P_2 P_3 $P_1 \rightarrow P_2 \rightarrow P_3$ ✓

Ch1: INTRODUCTION

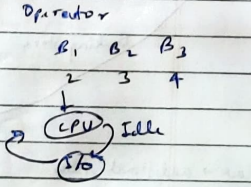
classmate
Date
Page

Operating System and its functions:

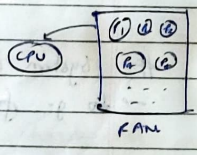
→ interface between applications and hardware
→ Resource management (eg giving resources to users on a server)
→ Process management (CPU scheduling)
→ Storage management (Hard disk → file systems)
→ Memory management (RAM)
→ Security

Types of Operating System:

i) Batch processing OS:
→ One process completes then moves to next process (If I/O the CPU idle)

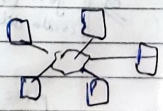


ii) Multiprogramming OS:
→ Non-preemptive (running to ready not possible)
→ But CPU can go to next task if I/O
→ CPU idleness reduces (multiple processes in RAM)



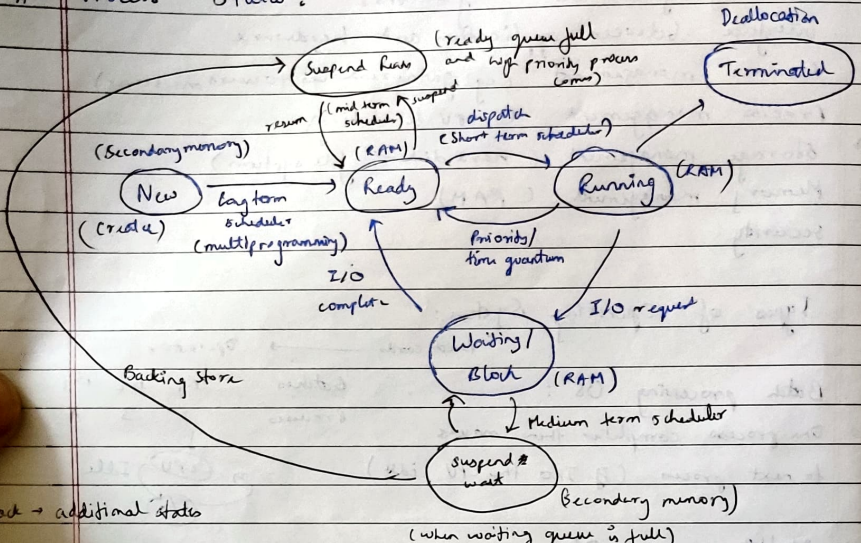
iii) Multitasking OS / Time sharing OS:
→ Preemptive
→ higher responsiveness (eg round robin)
(each process gets some time quantum or turn)

iv) Real time OS → Hard (strict time constraints) (eg missile)
→ Soft (Eg
v) Distributed OS (different geographical location)
vi) Clustered OS (load balancing, scalability, failure)
vii) Embedded OS (eg washing machine, AC)



classmate
Date
Page

Process States :



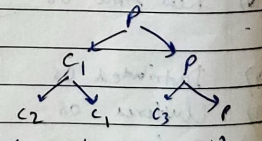
block → additional states

System calls : user program can request a service from the kernel which it does not have permission to perform
→ to go from user mode to kernel mode (eg : printf() ↔ write())

→ File related : open(), read(), write(), close(), create, delete
→ Device related : read, write, reposition, ioctl
→ Information : getpid, attributes, system time and date
→ Process Control : load, execute, abort, fork(), wait, signal
→ Communication : pipe(), create/delete connections, shmget()

eg fork() : (create child process)

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



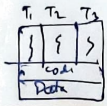
(1 parent, 3 child)

```
pid = fork();
```

pid = 0 (child process)
pid = +ve (parent process)
pid = -1 (fail to fork)

```
If (fork() && fork())
fork();
printf("hello")
```

4 times hello printed



classmate
Date
Page

#	Process	Threads
→	System calls involved (eg fork())	→ There is no system call involved (User level threads are created by apps).
→	OS treats diff. process differently	→ All user level threads treated as single task for OS
→	Diff. process have diff copies of code, data, stack, registers	→ Threads share same copy of code, data (diff stack & registers)
→	Context switching is slower	→ Context switching is faster
→	Blocking a process will not block other process	→ Blocking a thread will block the entire process
→	Independent	→ Interdependent

Kernel level threads : (similar to process)
→ managed by OS (system calls)
→ slower than user level threads
→ Context switching slower
→ If one kernel level thread blocked, no affect on others

Kernel : part of OS that interacts directly with hardware
Micro Kernel : smaller & supports only the core OS functionalities
Shell : CLI : receives commands from user & get them executed

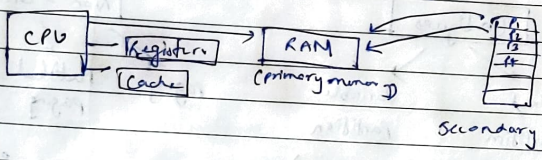
* Dual mode of operation :

User process calls system call (User mode (mode bit: 1)) → Kernel executes system call (trap : mode bit = 0) → return from system call to user process (mode bit = 1)

Ch 5: Memory Management

→ Memory management → method of managing primary memory

→ Goal: Efficient utilization of memory



→ Multiprogramming: multiple processes loaded in RAM (increases CPU utilization)

→ Degree of multiprogramming: (more and more processes in RAM) (in ready state)

Let there be n processes in RAM,

Let 'k' = probability of I/O operation

Probability that 'n' process will be performing I/O at same time = k^n

CPU utilization = $1 - k^n$

eg: 8 MB RAM, 4MB process (let I/O: 70% of time ($k = 0.7$))

at most 2 process in RAM

⇒ CPU utilization = $1 - k^2 = 1 - (0.7)^2 = \boxed{0.51}$ (51%)

4 MB RAM, each process also 4MB ⇒ 1 process in RAM

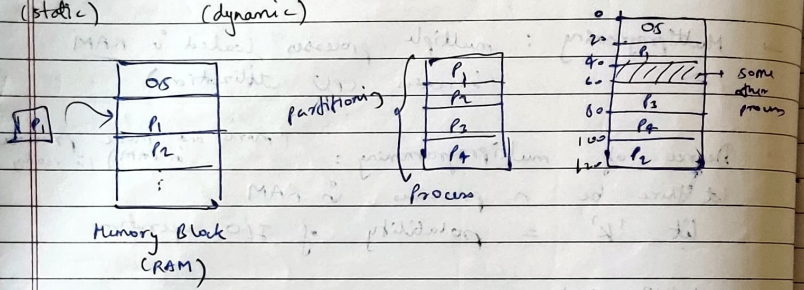
CPU utilization = $1 - k = 1 - 0.7 = \boxed{0.3}$

→ If n very large ⇒ CPU utilizat' ≈ 100%.

classmate Date Page

Memory Management Techniques

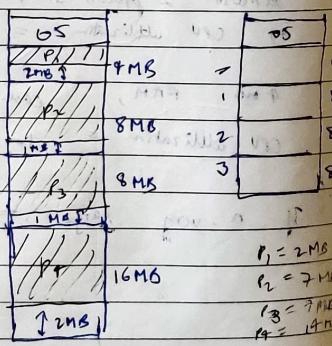
- Contiguous (used earlier)
  - Fixed partition (static)
  - Variable partition (dynamic)
- Non-contiguous
  - Paging
  - Multilevel paging
  - Inverted paging
  - Segmentation
  - Segmented Paging



Fixed Partitioning (Static Partition)

- → No. of partitions are fixed
- → Size of each partition may or may not be same.
- → Contiguous allocation so spanning is not allowed. (entire process must be allocated in one partition)

* Problems:
  - → Internal fragmentation (entire process in one partition → wastage of space)
  - → Limit in process size
  - → Limitation in degree of multiprogramming
  - → External fragmentation (memory available in different slots but cannot accomodate as allocation should be continuous)



(max process size = 16 MB)
(max 4 degree of multiprog.)
= no. of partitions

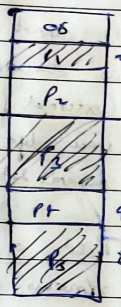
Variable Partitioning / Dynamic Partitioning :

→ Processes are allocated RAM at run time according to process size.

→ No internal fragmentation

→ No limitation on no. of process

→ No limitation on process size



* Limitations :

→ External fragmentation occurs :

→ Deallocation of process create holes

lets say P_2 and P_4 are deallocated,

memory deallocated = 4+4 = 8MB but now a process of 8MB cannot be allocated to these holes (allocation should be contiguous)

→ (can be removed using 'compaction', (remove empty holes by copy and pasting processes to available location)

→ Allocation & deallocation is complex

Memory Allocation :

i) First Fit : Allocate the first hole that is big enough

→ simple and fast

ii) Next Fit : Same as first fit but start search always from last allocated hole

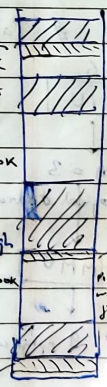
iii) Best Fit : Allocate the smallest hole that is big enough

→ internal fragmentation will be less

→ slow as entire RAM needs to be searched

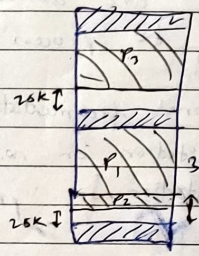
iv) Worst Fit : Allocate the largest hole

→ slow



eg: Requests from processes are 300K, 25K, 125K, 50K resp.

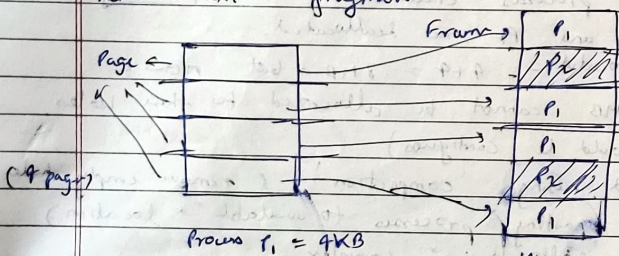
Best fit allocation :



P_4 cannot be accomodated as allocation must be contiguous (external fragmentation → space left = 50K but cannot accomodate)

Need of Paging :

→ removes external fragmentation.



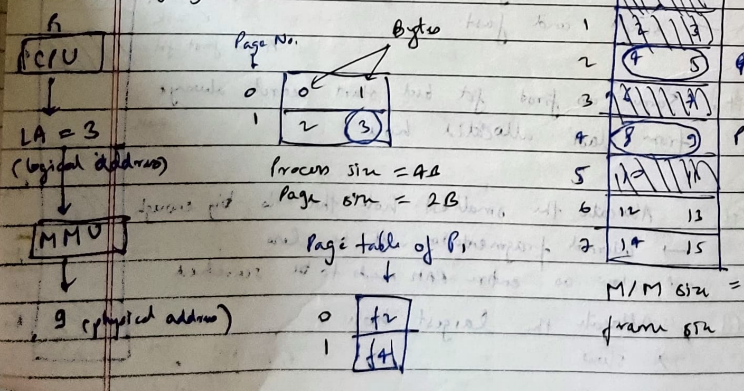
Process P_1 = 4KB

Main memory = 6 KB

(paging done in secondary storage block)

Page size = frame size

Paging :

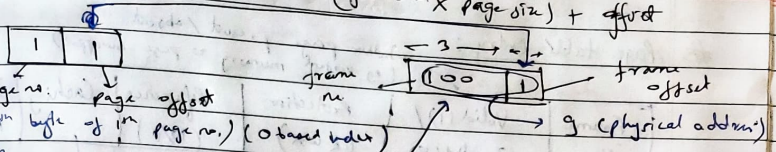


Given LA and page table, let from size = page size = 4 bytes (total 4 pages)
For LA = 3, PA = ?

Page no. → 2 bits, Page offset = 2 bits. 0 5 6 / 1 2 / 2 1 / 3 4

0 1 1 → 1 1 0 1 1 = (6 × 4) + 3 = 27

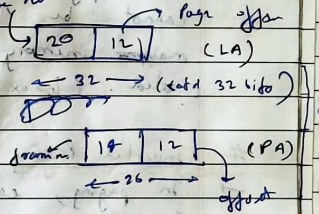
Logical address → (frame no. × page size) + offset ← Page table



Page no. = 1 ⇒ frame no. = 4 (from page table)
offset remains same

Hence page table is required for mapping logical address to physical address.
(done by Memory management unit (MMU))

Q) LAS (Logical address space) = 4 GB = $2^2 \times 2^{30} = 2^{32}$
PAS (physical) = 64 MB = $2^6 \times 2^{20} = 2^{26}$
Page size = 4 KB = $2^2 \times 2^{10} = 2^{12}$ (in bytes)
Memory is byte addressable.

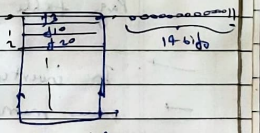


i) No. of pages = 2^{20}

ii) No. of frames = 2^{14}

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

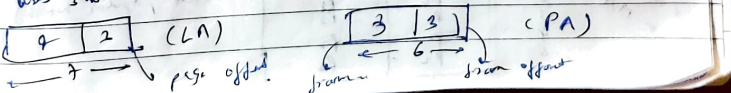
iv) size of page table = $2^{20} \times 14$ bits



no. of entries in page table
no. of bits required for representing frame no.
$2^0 - 1$

16 pages × 2^3 bytes = 128 Bytes LA

Q) If Logical address represented by 7 bits, PA = 6 bits represent
page size = 8 words
(Considering 1 word = 1 byte)
calculate no. of pages & no. of frame
$2^4 = 16$ $2^3 = 8$
8 bytes means 3 bits



Page table Entry :

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

(whether page present / absent (eg virtual memory so page swapped))

Mandatory fields

Optional fields

read, write, execute

Enable / Disable

(Page table uses MMU to map LA to PA)

→ If page is brought first time, reference = 0

(reference used in LRU (least recently used))

→ Dirty = 1 if page modified by user (eg write)

Multilevel Paging :

→ When page table size becomes very large and cannot be accomodated in a frame of main memory, another level of paging is created. (Index of index)

Inverted Paging :

→ In paging, each process has its own page table. Page table will be in main memory

→ In inverted paging, global page table

→ saves memory but searching would be linear

frame no	Page no	Process Id
0	p_0	P_1
1	p_1	P_2
2	p_2	P_1
3	p_1	P_3
4	p_0	P_2
5	p_2	P_3

To find frame no. corresponding to page no. & process ID, search would be done (not popular hence)

Q) Consider a virtual address space of 32 bits and page size of 4 KB. System is having a RAM of 128 KB. Ratio of page table and Inverted page table = ?
(Each entry in both is of 4B)

LA / virtual address

No. of page	page offset
20	12

← 32 →

No. of page = 2^{20}

PID	s	12

← 17 →

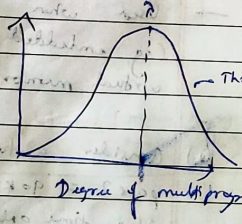
(RAM : PA)

No. of frames = 2^5 ($128\ KB = 2^7 \times 2^{10} = 2^{17}$)

$$\frac{\text{PT size}}{\text{IPT size}} = \frac{2^{20} \times 4B}{2^5 \times 4B} = 2^{15} : 1$$

Thrashing :

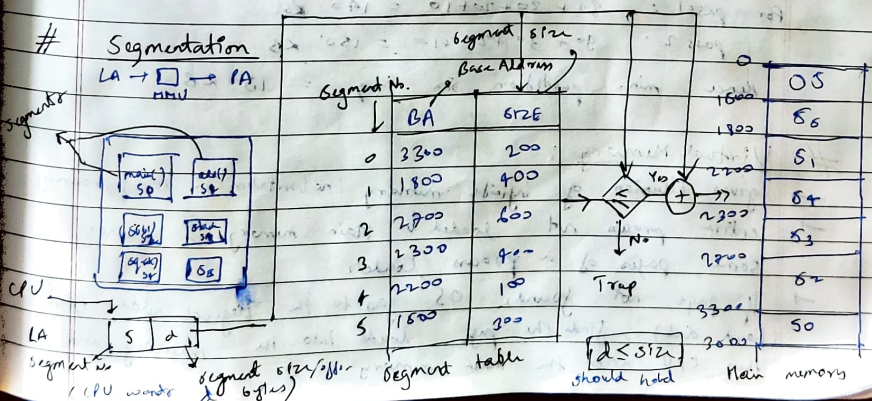
→ When all the pages of a process is not present in RAM, and CPU requires it then there would be page fault & time will be consumed to service the page fault (Bringing the page from secondary memory)



→ Sol^n : i) Increase RAM size
ii) Long term scheduler to limit degree of multiprogramming

Segmentation

LA → MMU → PA



→ [5 | d] : go to segment no. 5 in the segment table & find the base address. CPU wants d bytes. If segment size < d, then error (trap) else give the CPU d bytes starting from BA (base addr

→ Related data are present in a segment (user centric) (no random partitioning which is done with paging) Hence segment size may vary unlike page size which is fixed for each page.

Overlay:

→ Used when size of process > size of main memory (eg embedded systems) (in PCs, overlay not used rather virtual memory used)

Q) Consider a two pass assembler pass1 = 80KB, pass2 = 90KB, symbol table = 30KB, Common routine = 20KB. At a time only one pass is in use. What is the minimum partition size req. if overlay driver is 10KB.

80 + 90 + 30 + 20 + 10 = 230 KB

If size of RAM ≥ 230 KB, then no problem.

For pass1 : 80 + 30 + 20 + 10 = 140 KB
pass2 : 90 + 30 + 20 + 10 = 150 KB

Hence min. partition size req = 150 KB

Virtual Memory :

→ gives illusion of infinite memory (no limitation on no. of process)
→ entire process not loaded in main memory, rather some pages of a process loaded
→ If page not found, OS goes to the logical address space (hard disk), finds the page, loads into the memory and then CPU access the main memory

→ [s | d] : go to segment no. s in the segment table & find the base address. CPU wants d bytes. If segment size < d, then error (trap) else give the CPU d bytes starting from BA (base addr.)

→ Related data are present in a segment (user view) (no random partitioning which is done with paging) Hence segment size may vary unlike page size which is fixed for each page.

Overlay:

→ Used when size of process > size of main memory (eg embedded systems) (in PCs, Overlay not used rather virtual memory used)

Q) Consider a two pass assembler pass1 = 80KB, pass2 = 90KB, symbol table = 30KB, Common routine = 20KB. At a time only one pass is in use. What is the minimum partition size req. if overlay driver is 10KB.

80 + 90 + 30 + 20 + 10 = 230 KB

If size of RAM ≥ 230 KB, then no problem.

For pass1 : 80 + 30 + 20 + 10 = 140 KB

pass2 : 90 + 30 + 20 + 10 = 150 KB

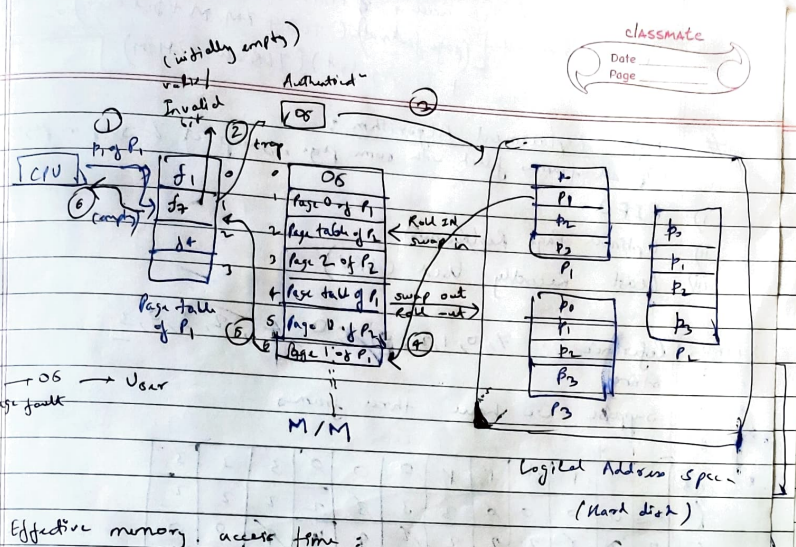
Hence min. partition size req = 150 KB

Virtual Memory :

→ gives illusion of infinite memory (no limitation on no. of process)

→ entire process not loaded in main memory, rather some pages of a process loaded

→ If page not found, OS goes to the logical address space (hard disk), finds the page, loads into the memory and then CPU access the main memory



Effective memory access time :

p → page fault probability

EMAT = p (page fault service time) + (1-p) (main memory access time)

nano sec

msec

Translation Lookaside Buffer (TLB) (cache) :

→ Page table entries are put in TLB (cache & hence faster)

→ If frame not found in TLB, then page table is used to find frame no. and then the frame is accessed from the main memory. (All frames can't be put in TLB as size is fixed & is costly)

→ If TLB hit, time taken = (TLB + x)
where TLB is time taken to access TLB
& x is time taken to access main memory

→ If TLB miss, time to access = TLB + x + x = TLB + 2x
(one x for accessing page table present in main memory)
(It is assumed that page fault does not occur)

EMAT = Hit (probability) (TLB + M/M) + Miss (TLB + 2 M/M)

EMAT # TLB hit rate (TLB access + M/M access)
+ TLB miss rate [(page fault rate)(TLB + 2M/M + Disk access)
+ (1 - page fault)(TLB acc + 2M/M)]

Page Replacement Algorithm :
(if all frames filled with some page, swap out & swap in page)

i) FIFO
ii) Optimal Page Replacement
iii) Least Recently Used (LRU)

i) FIFO : Reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

Suppose we have three frames

f3			1	1	1	2	0	0	0	3	3	3
f2		0	0	0	0	3	3	3	2	2	2	2
f1	7	7	7	2	2	2	2	4	4	4	0	0
	x	x	x	x hit	x	x	x	x	x	x	hit	

no replacement needed as empty space (but is also a page fault)

Page hit = 2

Page fault to = 10 (page miss)

Hit rate = $\left(\frac{2}{12} \times 100\right) = 16.67\%$.

Belady's Anomaly :

→ in FIFO page replacement sometimes even after increasing number of frames, page fault increases.

eg: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

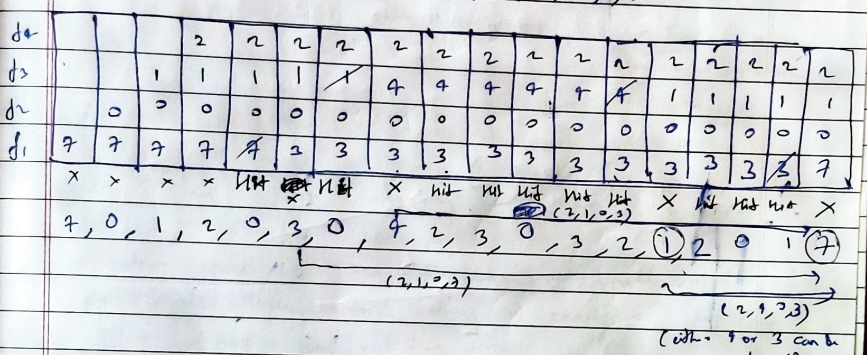
with no. of frames = 3, hits = 3, page faults = 9

" " " = 4, hits = 2, page faults = 10

ii) Optimal Page Replacement

→ Replace the 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



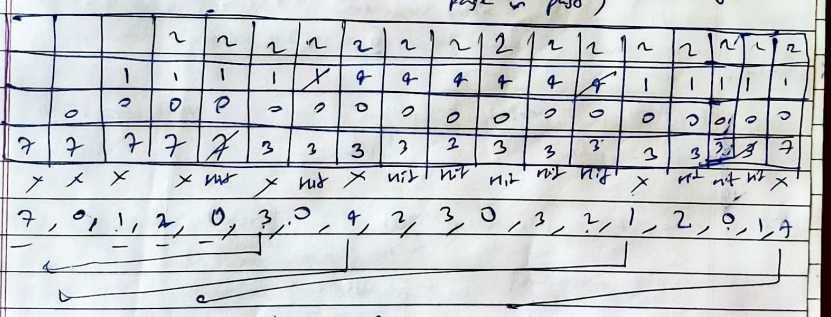
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7

(2, 1, 0, 3)

(2, 1, 0, 3)

(with 7 or 3 can be replaced)

iii) Least Recently Used (LRU): (replace the least recently used page in past)



7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7

iv) Most Recently used (MRU):

→ replace the most recently used in past

eg. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

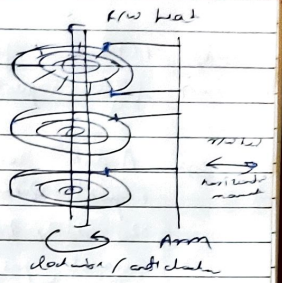
x x x x hit

(0 replaced)

(3 replaced)

Hard Disk Architecture :

Platter → Surface → Track → Sector → Data



Disk size = P × Su × T × Se × D

↳ generally 2 surfaces per platter

Disk Access Time :

i) Seek Time : Time taken by R/W head to reach desired track.

ii) Rotation time : Time taken for one full rotation (360°)

iii) Rotational latency : Time taken to reach the desired sector (half of the rotation time)

iv) Transfer time : $\frac{\text{Data to be transferred}}{\text{Transfer rate}}$

where transfer rate = (No. of heads (surfaces) × capacity of one track × No. of rotations/sec)

Disk access time = (Seek + rotational latency + transfer) → main + Controller time + Queue time

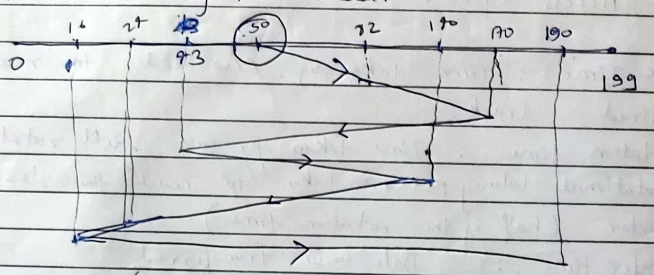
Disk Scheduling Algorithms : FCFS, SSTF, SCAN, LOOK, CSCAN, CLOOK

→ Goal : To minimize the seek time (time taken to reach desired sector)

i) FCFS (first come first serve)

→ fulfill request first for the request that came first

eg A disk contains 200 tracks (0-199)
Request Queue contains track no.
82, 170, 43, 140, 24, 16, 190 resp.
Current pos. of R/W head = 50
Calculate total no. of track movement / head movements by R/W head.

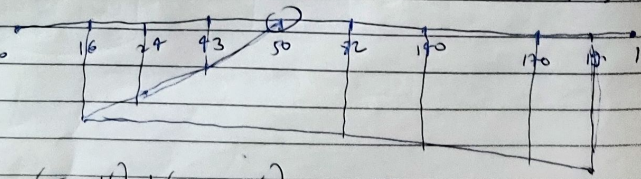


THM = (170-50) + (170-43) + (140-43)
+ (140-16) + (190-16) = 642

→ Advantage : No starvation
Disadvantage : inefficient

ii) SSTF (Shortest seek time first):
(go to the nearest track no. from current track.

eg same request as above:
sorted order : 16, 24, 43, 50, 82, 140, 170, 190



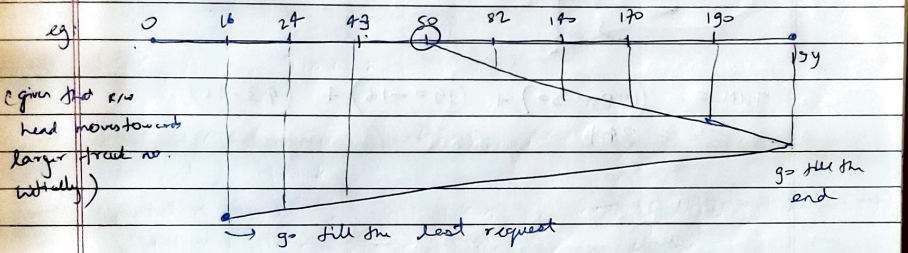
THM = (50-16) + (190-16) = 208

→ Advantages : Optimal (efficient), good avg. response time
Disadvantages : Starvation, higher complexity / overhead

iii) SCAN (elevator algorithm)

→ if from current position to the end to serve the request on the way.

eg:



THM = (199 - 50) + (199 - 16) = 332

→ If R/w head takes 5ns to move from one track to another, then total time = (332 × 5) ns

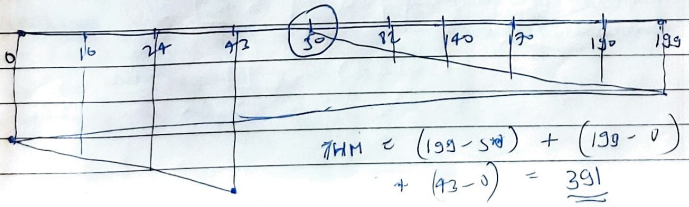
iii) LOOK :

→ similar to SCAN but don't go to the end.

THM = (190 - 50) + (190 - 16) = 314

iv) C SCAN :

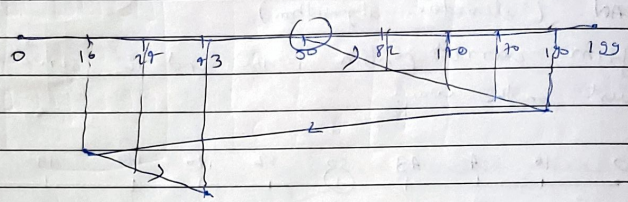
→ similar to SCAN but does not process request in reverse direction.



THM = (199 - 50) + (199 - 0) + (43 - 0) = 391

v) CLOOK : (circular LOOK)

→ similar to LOOK but do not process request in reverse direction



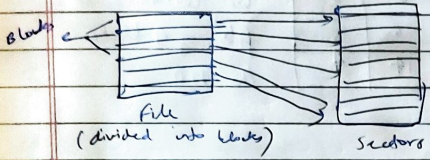
THM = (190 − 50) + (190 − 16) + (43 − 16)

= 341

File Systems in OS :

→ file system : software
functions : how file stored in disk and fetched.

User → file → folder/directory → file system



mapping from block to sectors

File Attributes & Operations :

→ Operations on File
i) Creating
ii) Reading
iii) Writing
iv) Deleting → file & attributes both deleted
v) Truncating → only data of file deleted
vi) Repositioning (attributes remain in the disk)
(change the position of r/w head within a file eg: seek, skip)

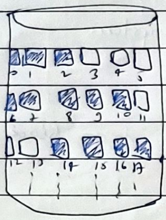
→ File Attributes (meta data)
Name,
Extension (Type)
Identifier (unique id)
Location (c://user/...)
SIZE (1 MB)
Modified / Created date
Protection / Permission (rwx)
Encryption, compression

Allocation Methods : Aim : i) efficient disk utilization
ii) faster access

Contiguous allocation
(for a given file, start from any location & store the blocks of file continuously in the disk.)

Non-contiguous allocation
→ Linked list allocation
→ Indexed allocation

Contiguous Allocation :



Directory

file	start	length
A	0	3
B	6	5
C	14	4

length → No. of blocks required

External fragmentⁿ :
Now if we consider first 3 rows in HD we cannot add a file of 4 blocks in the HD

* Advantages :

→ Easy to implement
→ Excellent read performance (both sequential & direct)

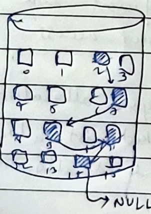
* Disadvantages :

→ Disk will become fragmented ⟨ Internal fragmentⁿ (2KB / 5KB) ; External fragmentⁿ ✓

(we can do nothing about internal fragmentⁿ as space will be left if file size (file block) < block size (physical block))

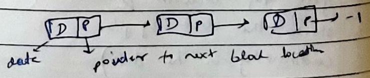
→ Difficult to grow file
(~~As more~~ contiguous space may not be available)

Linked List Allocation : (Non Contiguous)



Directory

File	Start
A	2
B	5



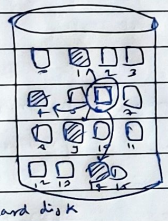
HAVELLS

classmate
Date
Page

* Advantages:
→ No external fragmentation
→ File size can increase

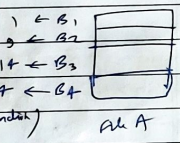
* Disadvantages:
→ Large seek time (blocks may be on different tracks)
→ Random access / direct access difficult (traverse from start always)
→ Overhead of pointers.

Indexed Allocation:



Hard disk

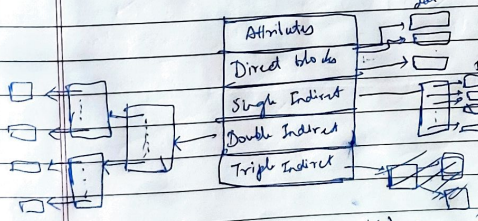
Directory	
File	Index Block
A	6



In block 6, the index file is present which contains pointers to the disk locations where the file blocks are present

Advantages:	Disadvantages
→ Support direct access	→ Pointer over head
→ No external fragmentation	→ Multilevel index (when index file becomes too large)

Unix INode structure:



Q) A file system uses Unix based data structure which contain 8 direct blocks, one double and one triple indirect block. Size of each disk block is 128 B & size of each block address is 8B. Find the max possible file size?

$(8 + 16 + (16 \times 16) + 16^3) \times 128\ B$