What is OS?

User → Application → OS → Hardware

MS-Word

1.1) operating System and its function

→ Facilitate interaction between Applications & Hardware.

→ Resource Management

→ Process management (Scheduling)

→ Storage Management (Hard Disk) → Through File system.

→ Memory Management (RAM) →

→ Security

1.2) Types of OS

makes Batch of similar kind of jobs / Non-preemption

Batch →

Multiprogrammed → **

Multi tasking → **

Real-time OS →

Distributed →

Clustered →

Embedded →

1.3) ~~Multiprogramming~~ → Minimize CPU idleness

Multiprogrammed → ~~Non-Preemptive~~

(i). If a process is given to CPU, it will execute it completely unless process goes for some I/O then it is preempted. → Maximize Responsiveness

multi tasking / Time sharing -

(i). A time is fixed then a process is preempted hence ~~a process don't take~~ increase in Responsiveness

L-1.4) Real-time OS

- Hard → Strictly Real-time Proces execution
- soft → Youtube Buffering.

~~L-1.~~ Distributed OS

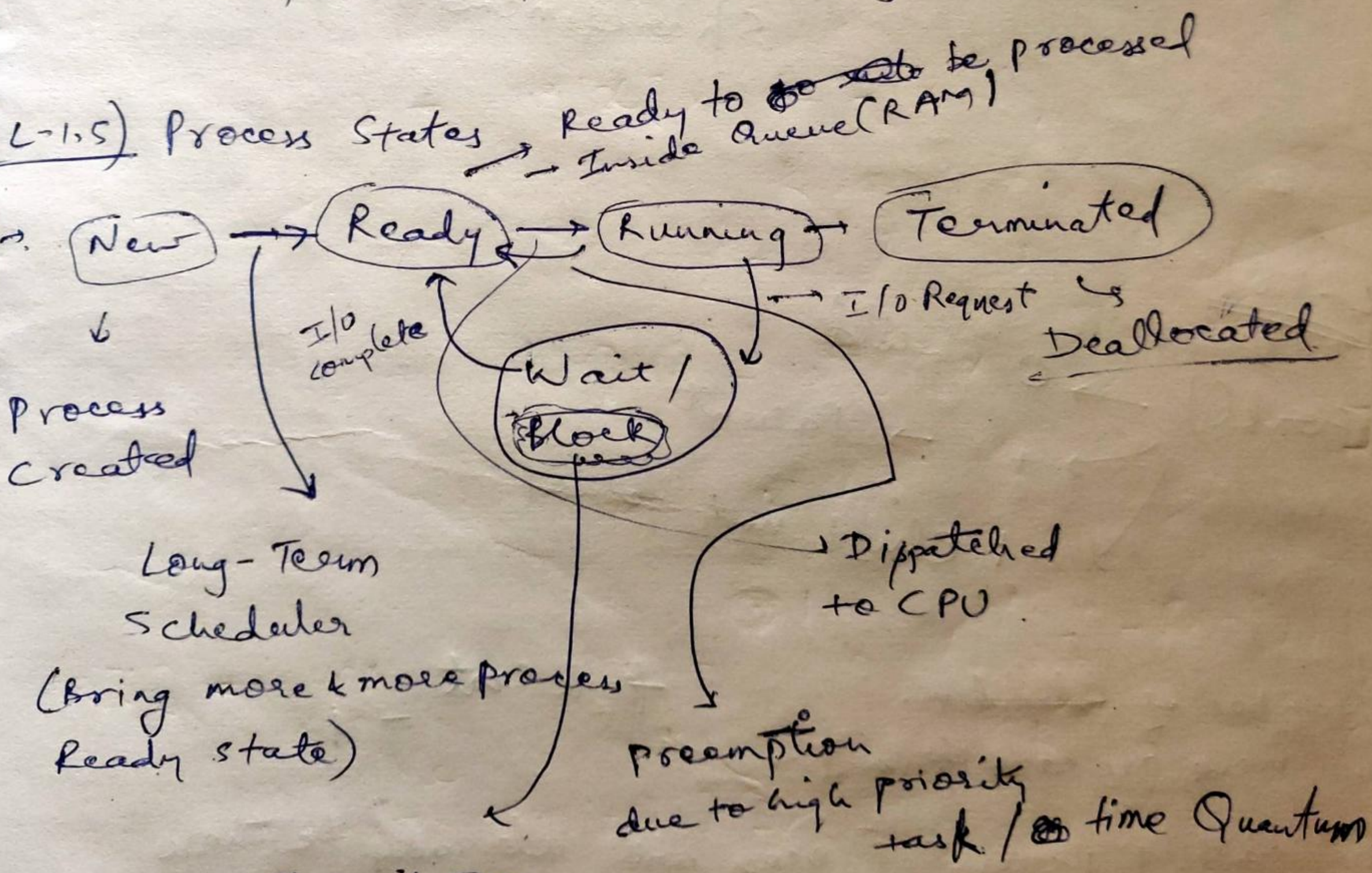
- Processing env is distributed and Connected via network.

Clustered OS

- using different machines as one cluster.
- Computation power increases

L-1.5) Process States

→ Ready to ~~be~~ be processed
→ Inside Queue (RAM)



If wait Queue is filled, swap out process to Secondary Memory

↓

Done via Medium-Term Schedules.

2-12) System Calls → Giving instruction to system

File related → open(), Read(), write()

Device Related → Read, write, Reposition

Information → getPid, attributes.

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

Communication → Pipe(), shmget()

↓

shared memory get.

2-18)

Fork() → 0 child

→ +1 → Parent

→ -1 child not formed.

Number of ~~childs~~. fork cnt

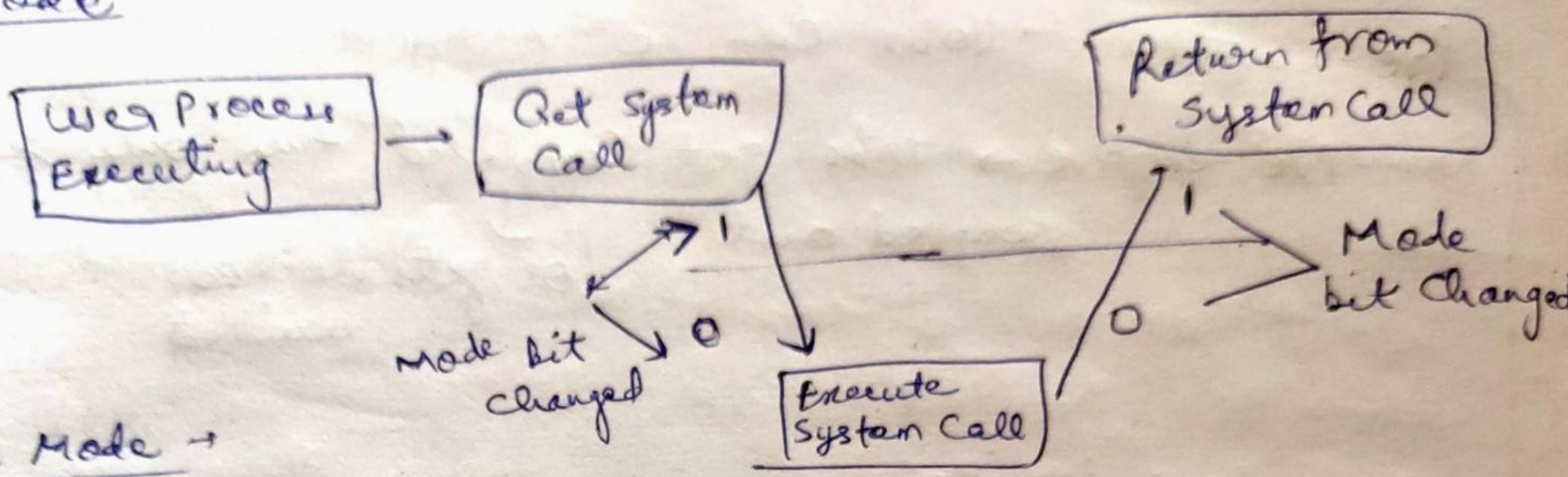
new process formed → $2^{\text{fork cnt}} - 1$

child + parent process will already be there

exec() →

L-1.10)

User Mode →



Kernel Mode →

Accesses Hardware

L-1.11) **

Difference between Process & Threads

Lightweight Process → Threads

Process

a). 5

L-1.12) ** User level Thread vs kernel level Threads

Q 2.1) → Increased Responsiveness

Pre-emptive → Jobs can be swapped

Non-Pre-emptive → Jobs get Completed then only gets swapped out
↓
or gets blocked/ waited

↳ Maybe done due to
→ Time Quantum
→ or Higher Priority Process arriving
→ or

Pre-emptive
→ Shortest Remaining time first
→ Longest —— 11 ——
→ Round Robin.

Non-preemptive
→ FCFS
→ SJF
→ LJF
→ Highest Response Ratio first
→ Multilevel Queue.

L-2.2)

→ Arrival time → The time at which process enter the ready state.
↑
Point of Time

→ Burst Time → Time required by process to get execute on CPU.
↓
Time duration

→ Completion Time → Time at which process get Completed
↑
Point of Time

→ Turnaround time → Completion Time - arrival Time
↓
duration

→ Waiting Time → Turnaround Time - Burst time
↓
duration

→ Response Time → Time at which Process get on CPU First Time
↓ - Arrival Time
duration

L-2.3)
First Come First Serve → Non-Preemptive

L-2.4)
Shortest Job first → Non-Preemptive
→ If any two job have same burst time select one with earliest arrival time
→ At a particular time, check for all the available jobs and select the one with smallest burst time.
→ once job get into running state, it wont be preempted even if a job with smaller burst time comes

L-2.5) Shortest Remaining time first — ~~Select on~~ In case of Burst time tie select job with earliest Arrival time.

- Based on Burst time
- Pre-emptive

(i) select ~~R~~ that process from all the arrived process which has smallest Burst time

(ii) At any time, if a job comes whose remaining time is less than remaining time of current running job then Pre-emption happens

~~L-2.6)~~

L-2.7) Round Robin

- Based on Time - Quantum
- Mode → Pre-emptive

→ Maintain a Ready Queue

→ At the ~~start~~ beginning of each time Quantum, update ready Queue with newly arrived process, and then pre-empt the current process, and push it back in ready Queue if its execution is remaining.

→ If a process get completed before the time Quantum expires, we take the first job in front on ready Queue and work on it. We dont wait for time Quantum duration to expire and then only bring other job.

→ Number of Content Switch in RR → Number of times we save current process and brings ~~back~~ new process

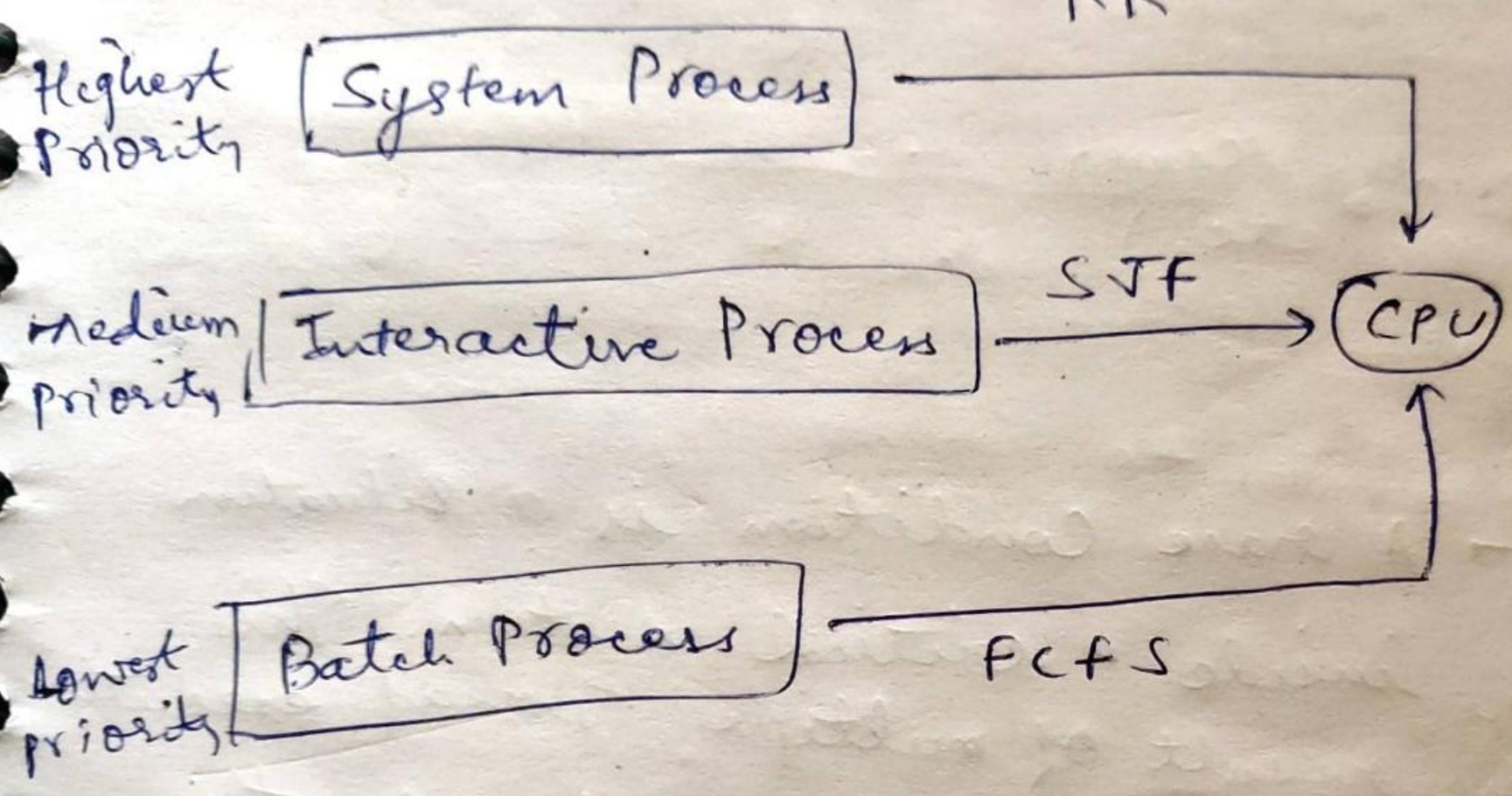
~~So, starting~~

L-2.8] Priority Scheduling

↳ Based on Priority

↳ mode → Pre-emptive

L-2.10] Multilevel Queue → may face the situation of Starving of low priority Process



L-2.11] Multilevel feedback Queue ← Preemptive

Similar to Multilevel Queue but here we increase the priority of process after certain time Quantum (i.e, we send it to higher priority Queue) to avoid starvation..

L-3.1)

Processes

- Co-operative Processes
  - Share Variable
  - —"— Memory
  - —"— Code
  - —"— Resource (printer, scanner etc)
- Independent processes

Race Condition → A race condition is a situation that may occur inside a critical section. This happens when the result of multiple Thread execution in critical section differs according to the order in which threads execute.

Critical Section → It is Code segment where the shared variable can be accessed by Cooperative processes. Only one process can execute in its Critical section at a time to avoid Race Condition.

L-3.2) Producer - Consumer Problem.

Code for Producer

```
void producer()
{ int itemp;
  while(true)
  { while(count == Buffer_Sz);

    Buff[in] = itemp;
    in = (in+1) mod Buffer_Sz;

    Count = Count + 1;

  }
}
```

How this line will be executed

(i) Store value of count on regist

(ii) increment registorValue

(iii) store value of register back to count

All the above three steps should be considered as atomic to avoid Race-Condition.

Code for Consumer

```
Void Consumer()
[ int itemc;
  while(true)
  { while (count == 0);

    itemc = Buff[out];
    out = (out+1) mod Buffer_Sz;

    Count = Count - 1;

  }
}
```

→ Critical section

L-3.3) Printer Spooler Problem.

avoid

L-3.4) Four Condition to satisfy for process synchronization

Primary Condition

(i) Mutual Exclusion → If one process is inside critical section, then another process can't enter critical section

(ii) Progress → If there is no process in critical section then a process should be free to enter critical section.

(iii) Bounded Waiting → ~~A process should~~ There should be a bound on waiting time of process to enter critical section

(iv) NO assumption related to Hardware & software

Secondary Condition

L-3.5) Lock Variable in OS.

Lock == 0 → Critical section empty

Lock == 1 → Critical section occupied.

```
while(lock == 1) ;      → Entry Code
lock = 1
Critical section
lock = 0                → Exit code
```

→ No guarantee of mutual exclusion if a process get preempt just before this line

(L-3.6) Test & Set instruction in OS.

→ Hardware based sol^n, no two function can enter it simultaneously.

```
while (test_and_set (& lock));  → Entry code

Critical section

lock = false;
```

boolean test_and_set (boolean * target)

```
{ boolean r = * target;
  * target = True;
  return r;
}
```

lock == True → occupied CS

lock == false → vacant CS

(L-3.7) Turn Variable (strict Alteration)

↓ No progress

Works on 2 process system only

Process P_0

```
while (turn != 0);
CS
turn = 1
```

process P_1

```
while (turn != 1);
CS
turn = 0
```

L-3.8) Semaphore

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative process in order to achieve Synchron--ization

Entry code

P, wait

```
Down (Semaphore S)
{
  S.value --;
  if (S.value < 0)
  {
    put the process in
    suspended list, sleep();
  }
}
else
  return;
}
```

Exit code

V, signal

```
up(semaphore)
{
  S.value++;
  if (S.value ≤ 0)
  {
    Select a process
    from suspended list
    and put it in ready
      queue, wake up();
  }
  return;
}
```

(-3.9) Binary Semaphore

```
Down(semaphore s)
{ if (S.value == 1)
    { s.value = 0;}
  else
  { put this process in suspended list or
    sleep();
  }
}
```

```
up(semaphore s)
{ if (Suspended list is empty)
    { S.value = 1 }
  else
  { Select a process from suspended list and
    wake up();
  }
}
```

Note → When a process is waken up from suspended list, it will only execute remaining part of code and not whole code.

(-3.10)

L-3.1) Solution of producer Consumer problem

```
Produce item( )
{
    down(empty);
    down(s);

  CS of Producer

    up(s);
    up(full);
}
```

```
Consume item( )
{   down(full);
    down(s);

  CS of Consumer

    up(s);
    up(empty);
}
```

L-3.2) Reader - Writer Problem

```
Semaphore mutex = 1, database = 1,
int rc = 0

Void reader( )
{ while(1)
  { down(mutex);
    rc++;
    if(rc==1) down(db);
    up(mutex)

  DB

    down(mutex)
    rc--;
    if(rc==0) up(db);
    up(mutex)
  }
}
```

```
void writer( )
{ while(1) {
  down(db)

  DB

  up(DB); }
}
```

mutex → It is to achieve mutual exclusion among readers

db → It is to achieve mutual exclusion between readers & writers

L-3.13) Dining Philosopher Problem

↳ For all N-1 philosophers

```
Void Philosopher()
{ while(1)
{ Think()
wait(fork_i)
wait(fork_i+1)
CS
Signal(fork_i);
Signal(fork_i+1);
}
}
```

For last philosopher

```
Void Philosopher()
{ while(1)
{ Think()
wait(fork_i+1);
wait(fork_i);
CS
Signal(fork_i+1);
Signal(fork_i);
}
}
```

L-4.1) Deadlock → If two or more processes are waiting on happening of some event, which never happens, then we say processes are in Deadlock.

Conditions for Deadlock

(i) Mutual Exclusion → Resource Causing deadlock should only be used by one process at a time only. Hence mutual exclusion

(ii) No preemption → There should be no preemption of process which is holding resource & causing deadlock

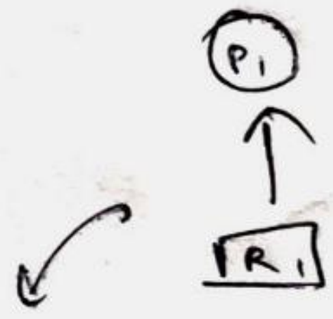
(iii) Hold & wait → Process involved in deadlock must hold some resource & wait for some other resource

(iv) Circular wait → There should be a cycle between waiting processes

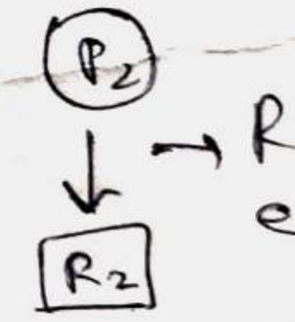
L-4.2) Resource Allocation Graph.

Process → ○

Resource → □



Assign edge



→ Request edge

Request resources in a particular order.

L-4.3) Methods to avoid deadlock →

(i) Ignore deadlock (ii) Deadlock Prevent → Remove any four of the necessary condition.

(iii) Deadlock avoidance (Banker's Algo)

(iv) Deadlock detection & recovery

~~L-5.1)~~

L-5.2) Memory Management Techniques

- Contiguous → Fixed Partitioning
  - → Variable Partitioning
- Non-Contiguous
  - → Paging
  - → Multilevel paging
  - → Inverted Paging
  - → Segmentation
  - → Segmented Paging

L-5.3) Internal Fragmentation in Fixed size partioning

→ No. of partitions are fixed

→ Size of each partition may or may not be same.

→ Contiguous allocation so partial insertion of process is not allowed in one partition and remaining in other partition

Disadvantages

(i) Internal fragmentation → When size of process is less than size of partition then the remaining space of that partition is wasted

(ii) Any process larger than max partition size wont get into RAM

(iii) External Fragmentation → ~~Due to wastage in space because~~ Even if we have total empty space equal to process size even then we cannot bring it inside RAM bcoz of contiguous allocation.

(iv) Since no. of partition is fixed, so a fixed number of process can be inside RAM

(-5.4) → Variable size partitioning

→ We will give spaces as the process arrives or make partition

→ No internal fragmentation

→ Increased degree of multi programming as compared to fixed partitioning

→ No bound on maximum process size

Problem →

(i) As a process leaves it will create a holes (non contig-ous) then there is chance of External fragmentation.

(ii) Allocation and Deallocation is complex.

(-5.5) Various Contigous memory allocation methods

First fit → Allocate the first hole that is big enough

Next-fit → Same as first fit, but starts search always from last allocated hole.

Best-fit → Allocate to smallest hole which big enough

Worst fit → Allocate largest hole.

L-5.8) Need of paging

In non-contiguous memory allocation, we divide process according to ~~xx~~ hole sizes, but this dynamic division is ~~sometimes~~ costly. Hence, we divide process ~~according~~ ~~to~~ into pages, and memory into frames. (in secondary memory already)

Page size = = frame size

L-5.9) Paging

Page no. → Page table → Frame No.

↓

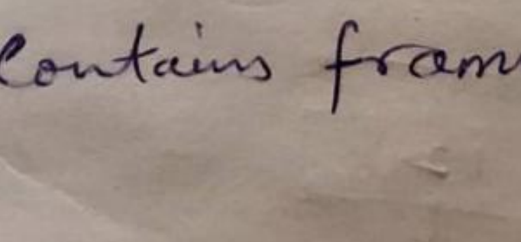
to map address generated by CPU to actual address in main memory

→ Number of entries in a process's page table = = Number of pages in that process

→ Entry of page table contains frame number of that page

→ Logical address generated by CPU

→ Physical Address



Page offset = = frame offset

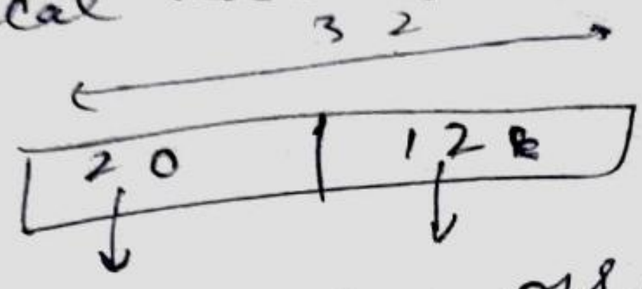
L-5.10) Question: Memory is Byte Addressable

Logical Address Space = 4GB, No. of bits in Logical address = $2^2 \times 2^{30}$

↑ Given

Page size = 4KB ∴ No of Bits to represent one page = $2^2 \times 2^{10}$

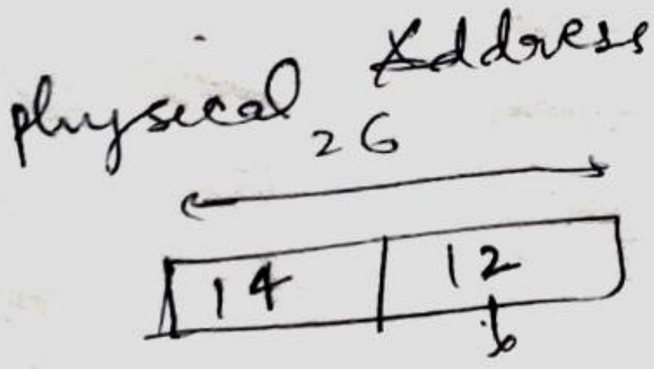
So logical address



Total page count = 2^{20}

Physical Address space = 64MB (Given)

Bits to represent physical address $2^6 \times 2^{20}$



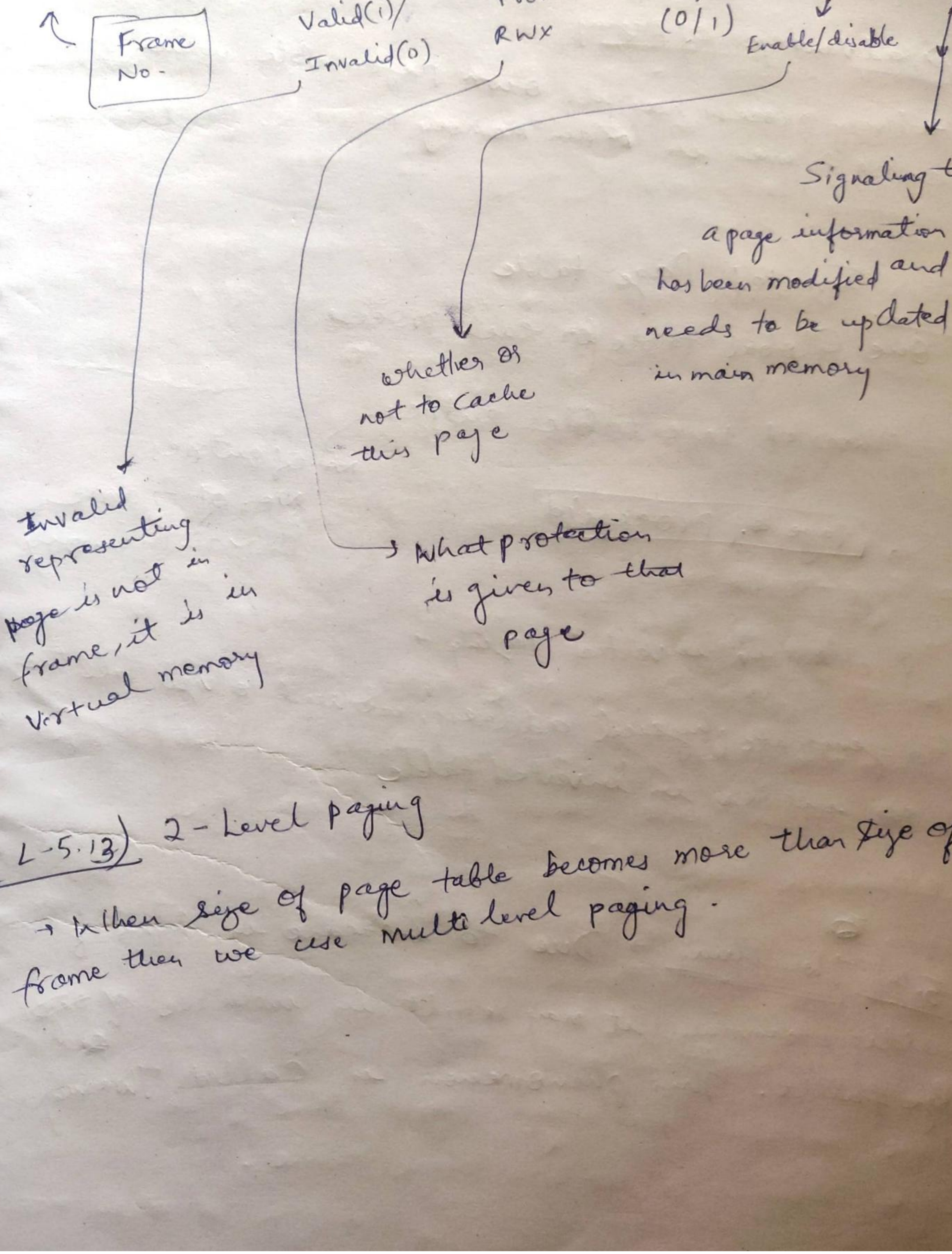
No. of entries in page table = No. of pages = 2^{20}

Size of page table = No. of pages × No. of bits to represent frame

$2^{20} \times 14$

L-5-12) Page-table entry

mandatory field. rest are optional



L-5-13) 2-Level paging

→ When size of page table becomes more than size of frame then we use multi level paging.

L-5.14) Inverted Paging → Saves Memory.

Disadvantages of paging

→ Every process has its own page table which occupies extra frames in main memory.

Inverted Page table

1). One Global table for each process.

Inverted Page table entries → No. of entries = No. of frame in Main Memory.

| Page no. | Process id |

Problem in inverted paging →

→ To generate physical address, we need to visit all the page entries in inverted table

L-5.15] Thrashing → Due to increase in degree of multiprogramming, it may result in increase of page fault. This phenomenon is called Thrashing

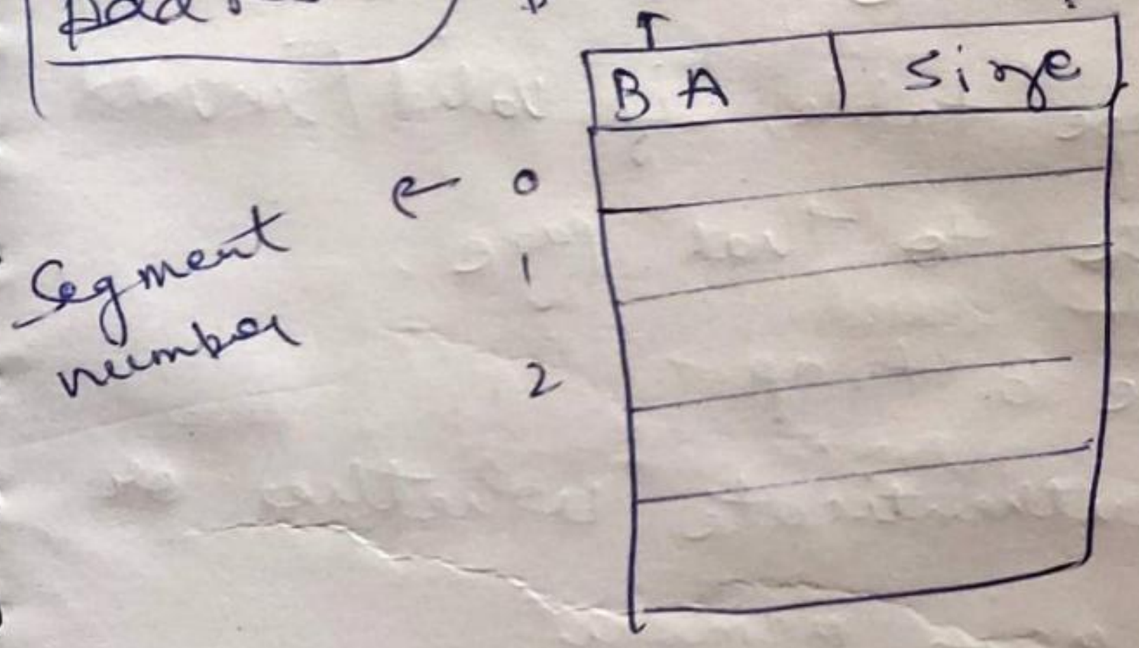
L-5.17) Segmentation Vs Paging

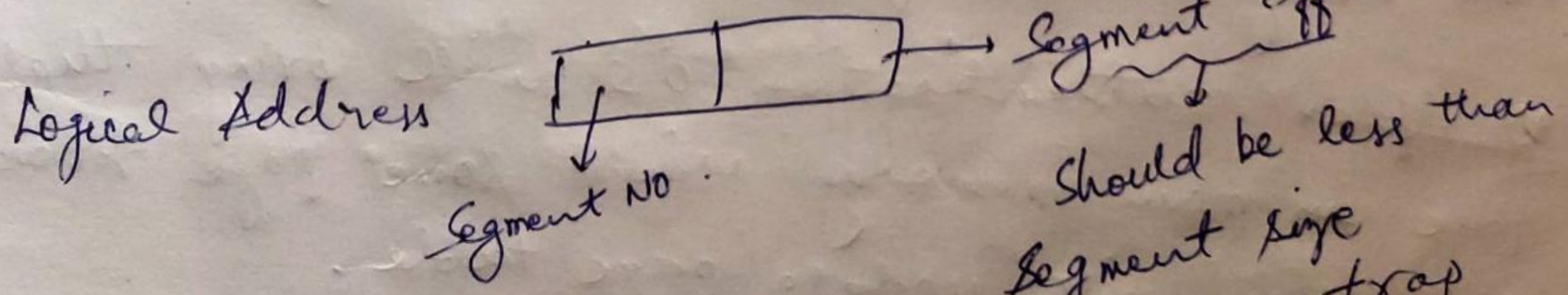
→ Paging blindly divides the whole process into pages, which may result in a particular functionality being divided into two or more pages

→ Segmentation will divide whole process into segments based on functionality resulting in Variable Segment size.

* * *
* CPU always works on logical Address

Logical Address → Segment table → Physical Address





→ useless topic

L-5.18) Overlay → Process by which a large size process can be inserted into main memory.

There is no driver in O.S which supports overlay & user has to provide appropriate partition for the process.

L-5.19) Virtual Memory → Using concept of virtual memory we can bring process of any size to primary memory, by swapping in and out required page.

How does it work

Page fault →

→ (i) When there is a request for a page, OS looks for its address in page table.

→ (ii) In the address in page table, if valid/invalid bit is set it will simply go to that page, otherwise will generate a trap.

→ (iii) OS will now first authenticate whether or not bring that page in main memory.

→ (iv) After authentication OS will bring that page in main memory using some page replacement algo. And will update page table.

Page fault → When OS looks for a page in main memory and couldn't find. Page fault happens

→ Effective Memory Access Time

$$= P \times (\text{Page fault service time}) + (1-P)(\text{main memory access time})$$

P → Probability of Page fault

2-5.20) Translation Lookaside Buffer (TLB)

→ TLB is faster than main memory

→ Let x be time to access main memory

→ In a normal page access.

Get frame no. from page table → Then look in main memory for that page

x + x

So, x + x time in general.

→ So, first, we try to search in TLB, if found there then simply go to main memory to obtain that page, this is called TLB-Hit

So, (Hit chance) × (TLB + x)

+ (miss change) × (TLB + x + x)

= Effective memory access time assuming no page faults

L-5.22) FIFO Page Replacement Algorithm

(1) Whenever page fault happens, remove first and frames are filled
Come page

L-5.23) Belady's Anomaly in FIFO

→ upon increasing number of frames, no. of page fault increases.

→ E.g 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

with 3 & 4 frames

L-5.24) Optimal Page Replacement

→ Replace the page, which is not going to be used for longest time in future.

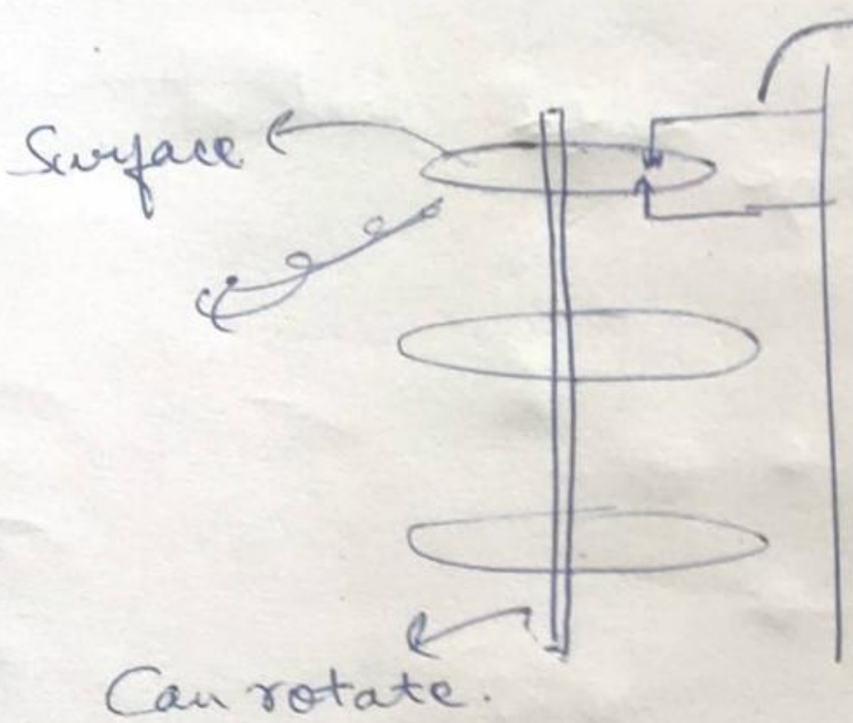
L-5.25) LRU (Least Recently used)

Remove the page which has been least recently used.

L-5.26) MRU (Most Recently used)

Remove the page which has been used most recently

L-6.1) Disk architecture



Platter $\xrightarrow{has}$ Surface $\xrightarrow{has}$ Track $\xrightarrow{has}$ Sectors → data.

Or

disks

L-6.2)

Seek time → Time taken by read-write head to reach desired track.

Rotation Time → Time take for one full rotation (360°)

Rotational Latency → Time taken to reach to desired sector.

Avg. → half of rotation time

Transfer Time → $\frac{\text{Data to be transfer}}{\text{Transfer rate.}}$

Transfer rate → No. of Surfaces (2 for up & down) × Capacity of one track × No. of rotation in one second

Disk access time → Seek time + Rotational latency + Transfer Time

L-6.4) ~~FCFS~~

Disk scheduling Algorithm

↳ Aim to minimize ~~total~~ seek time

FCFS → No starvation

Easy to implement

L-6.5) SSTF → Shortest Seek time first

Go to nearest track, resolve tie depending on direction.

↳ May face problem of Starvation.

L-6.6) SCAN → Direction will be given in question.

→ move in One direction till end then only switch direction

~~L-6.8~~ After switching direction go to only last query of that direction

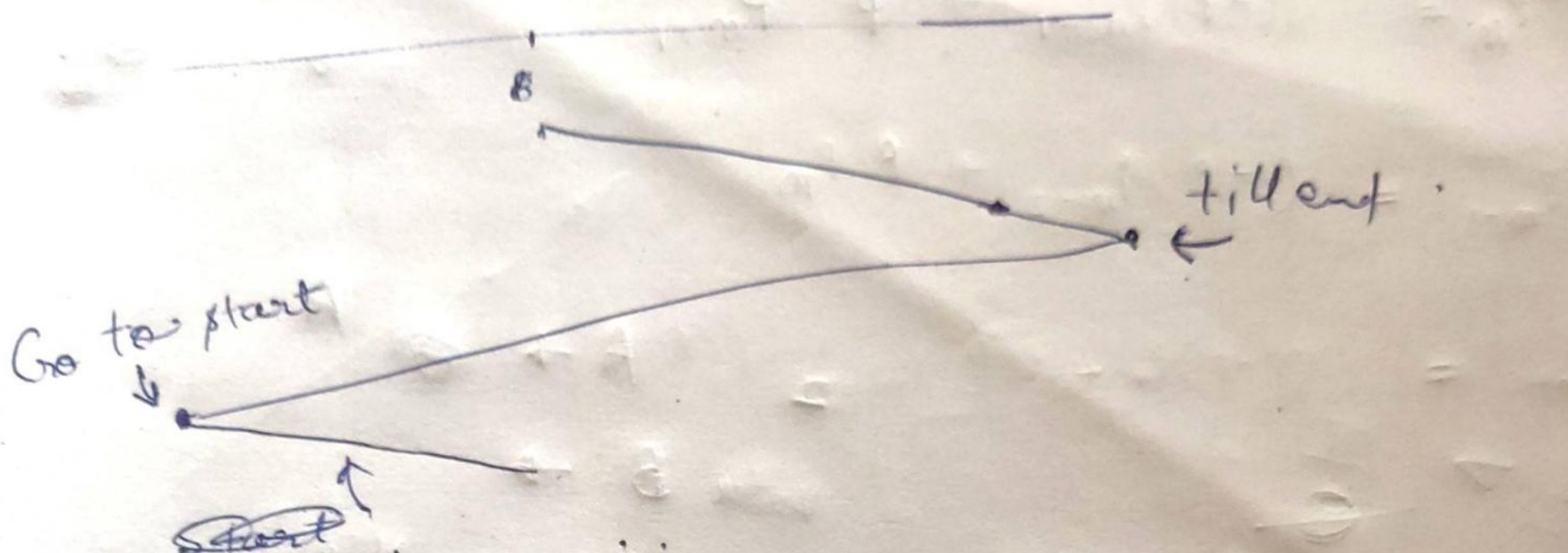
not only to last query but to Complete end

L-6.7) LOOK → Similar to SCAN but only move till last Queries in both end.

L-6.8) C-SCAN → ~~Simply~~ Circular Scan

L-6.9) C-LOOK → Circular Look!

L-6.8 C SCAN



L-6.9