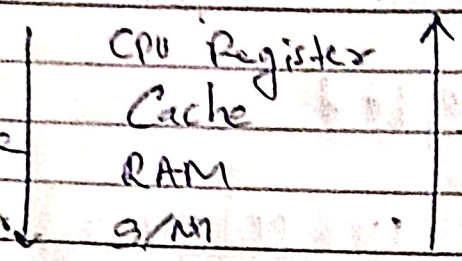
06/08/25

Sessional - 2

* Memory Management

Memory hierarchy



* Primary Memory

↳ Byte addressable sequential

Need of Memory mgmt.

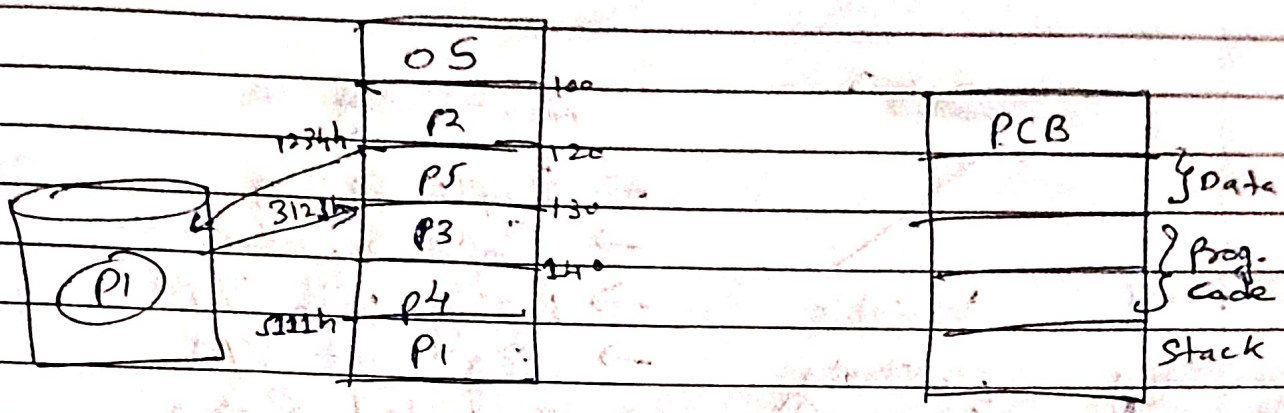
(i) Applications are demanding more memory
(ii) Swapping
(iii) I/O devices are slow compare to CPU.

* Memory Management Requirements

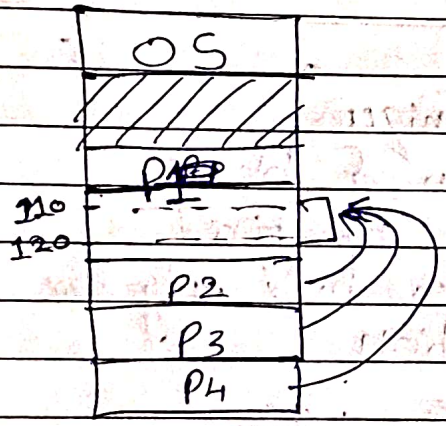
(i) Relocation → OS (time consuming)
(ii) Protection → H/W (TRAP)
(iii) Sharing → Multiple processes can access same function.
(iv) Logical organization
(v) Physical organization

Memory management

→ The memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.
→ Keeping as many processes inside RAM.

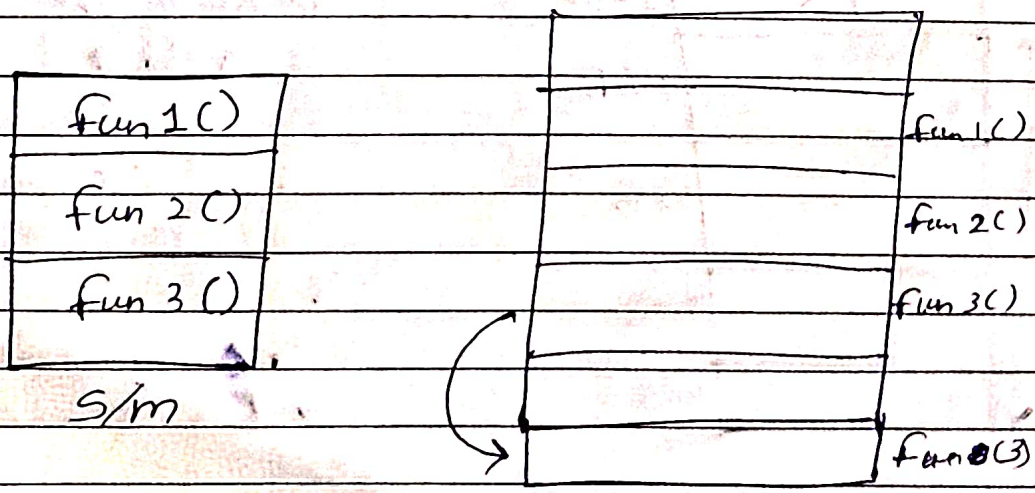


- Sharing



- These must protection mechanism

- Logical organization



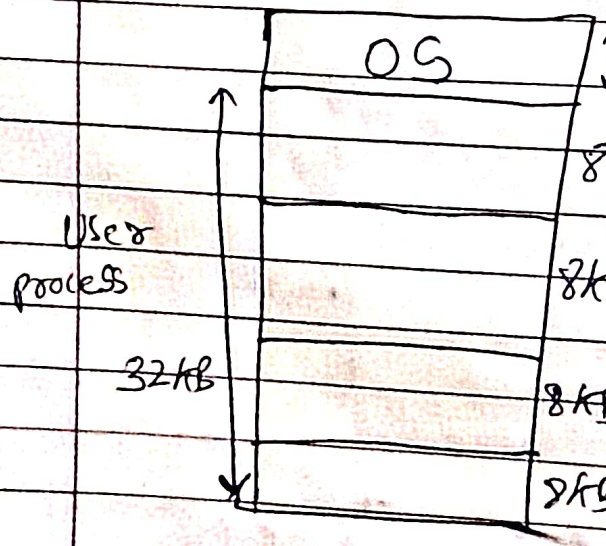
* Memory Management Techniques

- Contiguous
  - fixed partitioning (Static)
  - Variable partitioning (Dynamic)
- Non-contiguous
  - → Simple paging
  - → Simple Segmentation
  - → Paging with VM
  - → Seg. wit VM
  - → Inverted Paging
  - → Multi level Paging

07/08/25

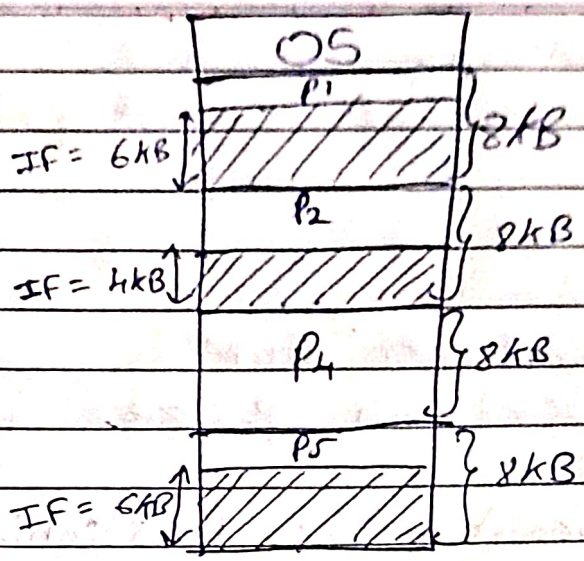
* Fixed Partioning Technique

↳ contiguous allocation of blocks
↳ No. of partitions are fixed
↳ Each partition can hold only one process
↳ Size of partition
  ① Equal size
  ② Unequal size



Consider a primary memory of 32KB is partitioned using fixed partitioning / Equal size partitions as given below. How does each of the following process request is processed by OS.

Eg. $P_1 = 2k$
$P_2 = 4k$
$P_3 = 9k$ X
$P_4 = 8k$
$P_5 = 2k$



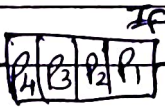
IF = 6+4+6 = 16KB

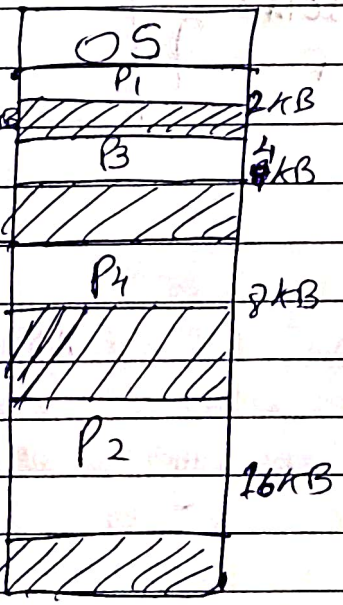
* Disadvantage
(i) Limit on size of process
(ii) No. of processes → No. of partitions
↓
Limit on DoM [Degree of multiprogramming]
(iii) Internal Fragmentation
↳ Difference b/w Allocated process size and Partition size.

* $P_1 = 1KB$
$P_2 = 14KB$
$P_3 = 2KB$
$P_4 = 1KB$
$P_5 = 32KB$ X
$P_6 = 2KB$ ✓

Unequal size

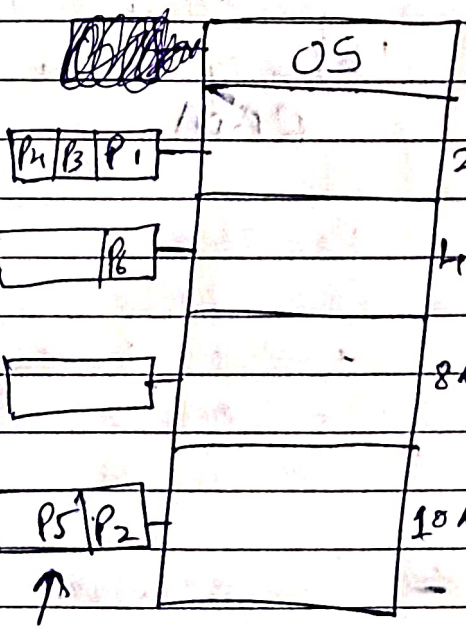
Single Queue





after say P_4 terminated then P_6 can be accomodated

Per Partition one Queue



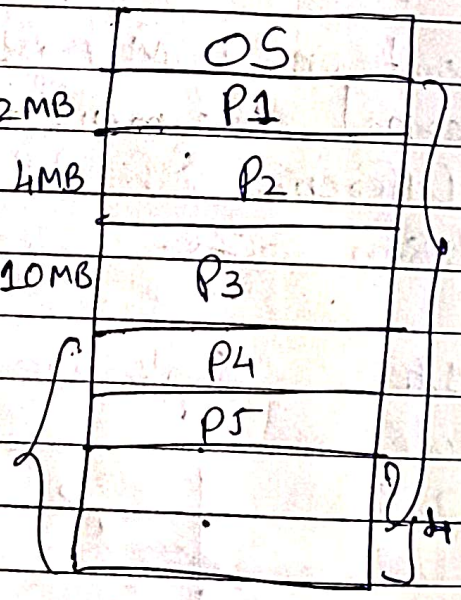
let say assume this scenario

- Advantage
  ↳ Different size of process { upto certain size
  [after that again there is limit

- Disadvantage
→ Limit on DOM
→ Limit on P-size.

* Variable Size Partitioning (Dynamic)
  ↳ Initially RAM is empty
  ↳ The partitions are created based on process size requirement
  ↳ Partition size no. of partitions. } variable

E.g. P1 = 2MB
P2 = 4MB
P3 = 10MB



- EF (External Fragmentation)
  ↳ Empty size outside partition
  ↳ Empty size in RAM.

P4 = 5MB
P5 = 1MB
t1 ⇒ P1 terminates, Now EF = 6 MB = 4+2 = 6
P6 = 5MB X ⇒ Can't allocate cuz no contiguous memory available.

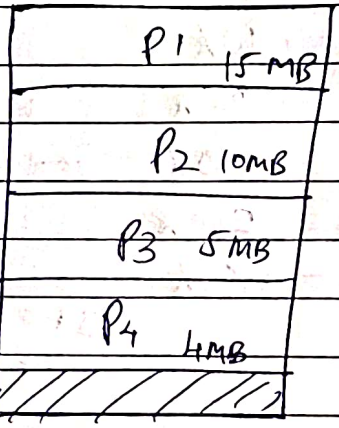
Compaction
→ Collecting the free holes and merging them as last EF.

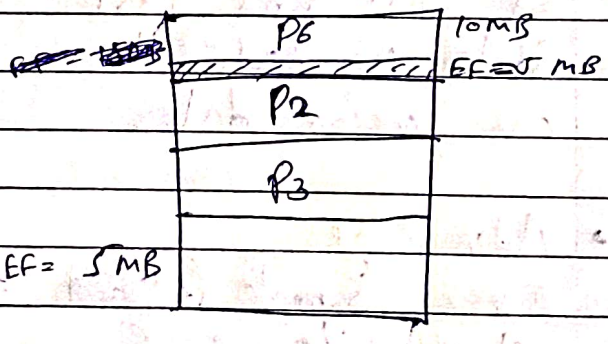
Compaction

⇒ Technique to collect the empty holes (EF) together to ~~form~~ a large memory chunk.

Q. Consider the primary memory of 35 MB is managed using variable partitions without compaction. How does each of the following request processed by OS?

Process requests
P_1 = 15 MB
P_2 = 10 MB
P_3 = 5 MB
P_4 = 4 MB
at time t1
P_1, P_4 ⇒ terminates
P_5 = 20 MB ✗
⇒ EF = 20 MB
P_6 = 10 MB
⇒ EF = 10 MB
P_7 = 9 MB ✗





- Compaction ⇒ time consuming, overhead on OS.

Q. 1000 KB of MM is manage using variable partition with compaction. It currently has two partitions of size 200 KB and 260 KB respectively. Which one of the following request in KB that could be denied for.

→ Right now these two are holes.
∴ Compaction = 200 + 260 = 460.

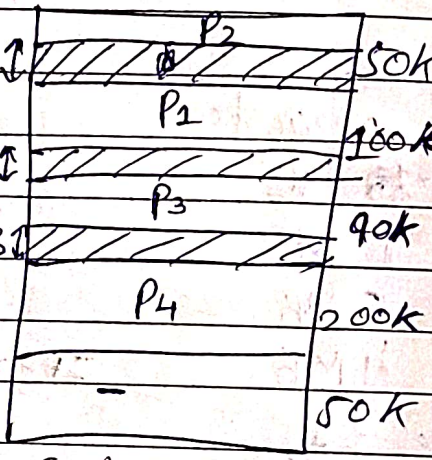
(a) 201 (c) 461
(b) 261 (d) 460

* Allocation methods in contiguous allocation
or
Placement Algorithm ⇒ Decision to select the hole

(1) First Fit [Always scan from first]
(2) ↳ OS scans the RAM from beginning and chooses the first available block which is large enough to hold the process.

Eg. $P_1 = 90k$
$P_2 = 20k$
$P_3 = 50k$
$P_4 = 200k$
$P_5 = 60k$

IF = 30KB
IF = 10KB
IF = 40KB
IF = 80KB
EF = 50KB



Fixed partition
∴ sizes are given

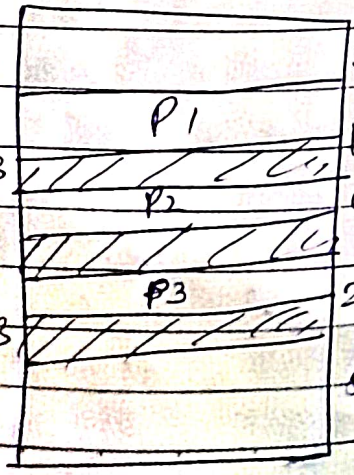
- It will find first hole in which it can accomodate.

- Advantage
↳ very simple & ⇒ Easy of Implement.
Slow in terms of movements.

(2) Next Fit
→ Wherever you are find the next favourable hole.

Eg. Same example as above
$P_1 = 90k$
$P_2 = 20k$
$P_3 = 50k$
$P_4 = 200k$ X
$P_5 = 60k$ X

EF = 10KB
IF = 10KB
EF = 100KB
IF = 230KB



- Next fit is faster then first fit

(3) Best Fit

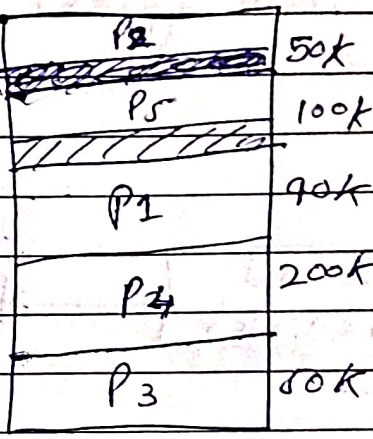
↳ Choose the smallest hole which is big enough to hold the process.

E.g. $P_1 = 90K$
$P_2 = 20K$
$P_3 = 50K$
$P_4 = 200K$
$P_5 = 60K$

IF = 30KB

IF = 40KB.

IF = 70KB.
EF = 0KB



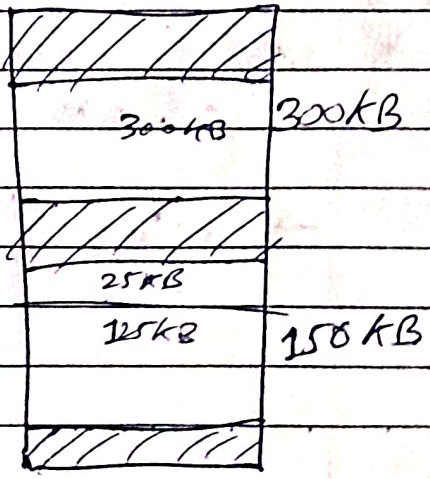
- Time consuming ⇒ Slower then all 3.
- IF is Less

(4) Worst Fit

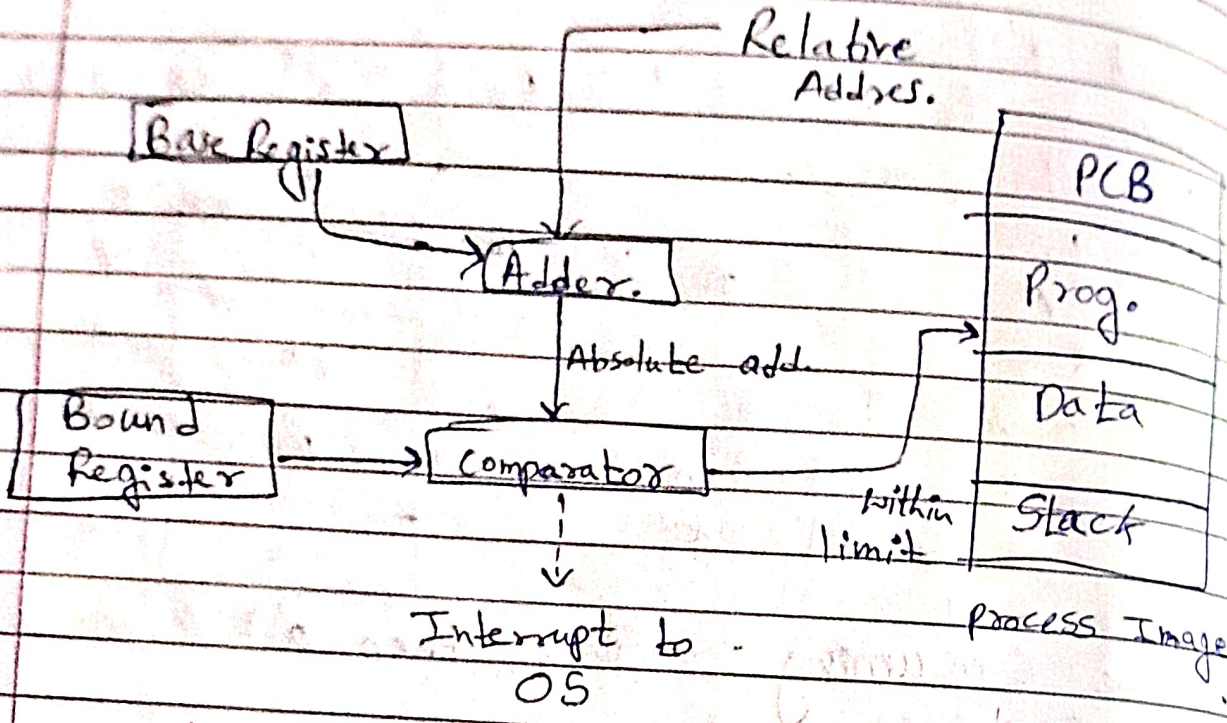
→ Choose the larges hole which is enough to hold the process.

Q. The following memory is using variable partitioning without compaction. It currently has 2 partitions of size 300KB and 150KB respectively. The request from the processes are 300KB, 25KB, 125KB and 50KB respectively. Which placement algo. can satisfy this request.

→ Next Fit



* H/W support for Relocation



Absolute Adder. = BA + RA

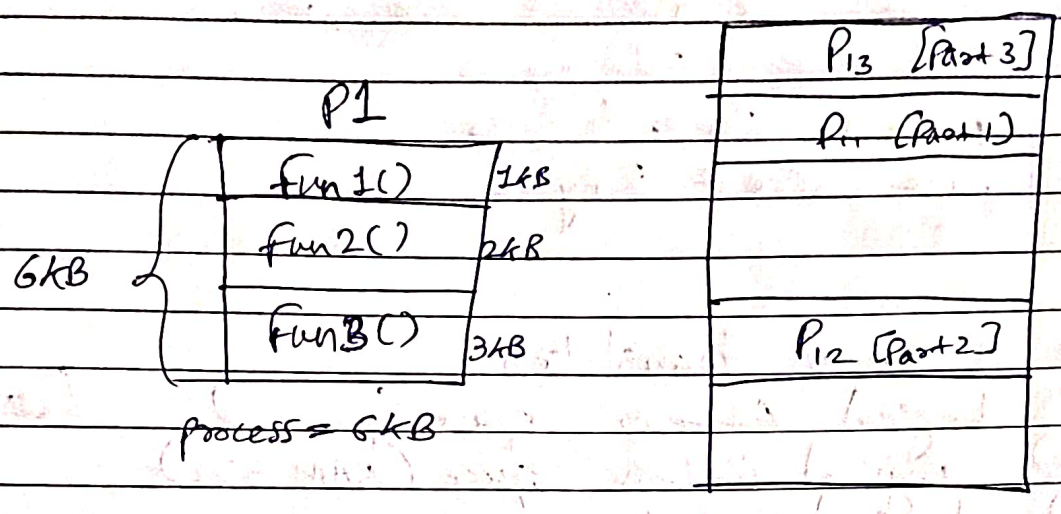
- Logical address
  ↳ Reference to the logical memory location of current assigment.

- Relative Address
  ↳ LA, a location Relative to some known point.

Physical address = Absolute address
location in M/M.

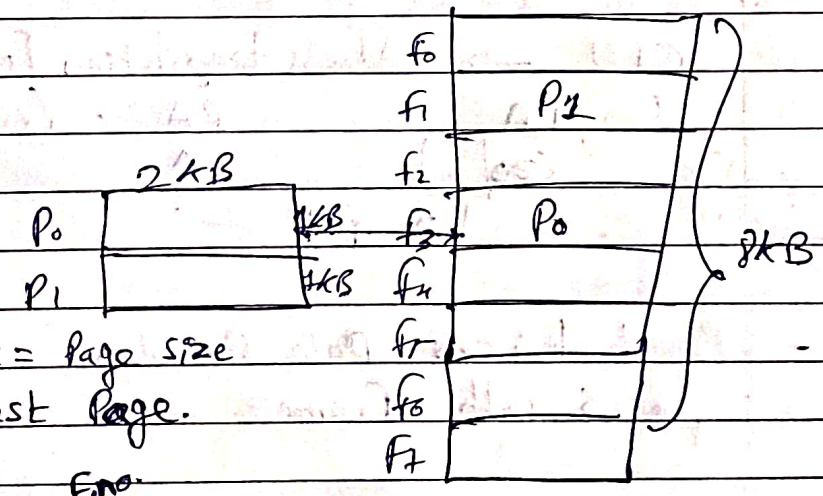
02/08/25

* Non-Contiguous Memory allocation
  ↳ allocation of processes in non-sequential manner.
  ↳ Paging (Simple)

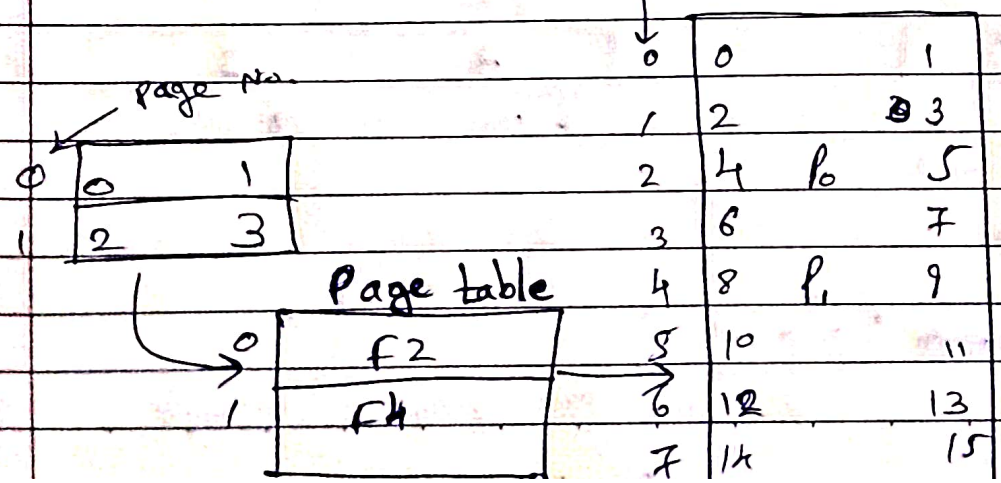


- page:- The process is divided into equal parts called as page.
- Frame:- The P.M. is divided into equal no. of parts called as frame.

E.g. RAM = 8kB
Frame size = 1kB
Process size = 2kB
page size = 1kB



Frame size = Page size
These can be IF in last Page.
IF = last page



[question of Previous diagram]

Q. Main Memory = 16 Bytes
Process size = 4 Bytes
page size = 2 Bytes

* Address translation
(1) Logical Address space (process) [e.g. ./a.out file]
(2) Physical Address space (RAM size)
(3) Logical Address
(4) Physical Address

→ No. of Frames = $\frac{16}{2} = 8$

P_size = $\frac{4}{2} = 2$

CPU → Address translation [MMU] (Memory mgmt. Unit)
↳ LA → 3 → PT → PA → (Physical Addr.)
(logical Addr.)

- Page table :- is Data structure that stores mapping of pages with frames.

LA
| 1 | 1 |
Page No. offset (Page size)
$2^1 = 2$ Bytes

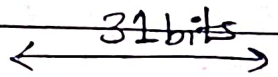
PA
| 3 | 1 |
F. No. offset
$2^3 = 8$

| 11 | → | 1001 |
LA PA of our previous example

[k]?

- Size of PT = no. of pages * bits to represent 3 byte frame.

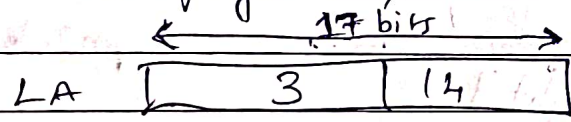
LA = 31 bits



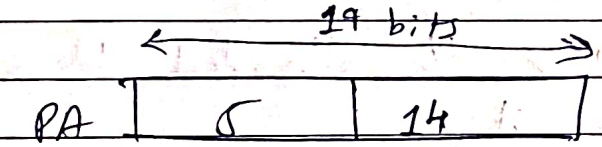
2^{31} bytes
= 2GB

Eg. LAS = 128KB = 2^{17}
PAS = 512KB = 2^{19}
Page size = 16KB = 2^{14} [Block size = frame size]
No. of pages ?, no. of frames?, PT size

→



No. of pages = $2^3 = 8$



No. of frames = $2^5 = 32$

~~No. of pages = $2^3 = 8$~~

PT size = 2^3 pages * 5 bits for each p.no
=

12/08/25

* Fixed partition → limit on DOM
  Dynamic partition → Compaction overhead

* Buddy system:-
→ It allows single block to be spited into two half called as buddies.
→ memory block is available in 2^k such that

$$L \leq k \leq U$$

L → small block
U → largest block

→ Allocated block:- find a free block of size 2^k mark it occupied and return a pointer to it. [Always left to Right]

→ Deallocated block:- Mark the allocated block as empty and may merge it.

Steps

Step1:- If a process request of size s such that

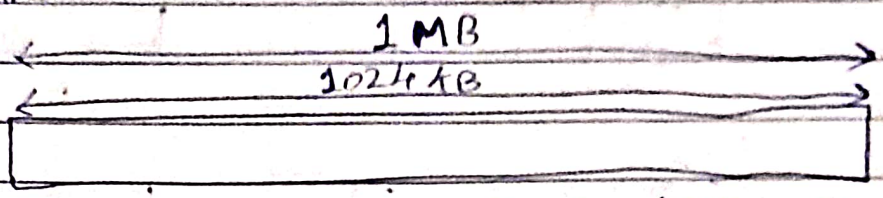
$$2^{U-1} \leq s \leq 2^U$$ then entire block is allocated

Step2:- If the block found is too large then divide it in two halfs.

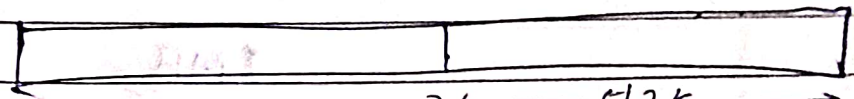
Step3:- If the block is too large then Repeat Step 2

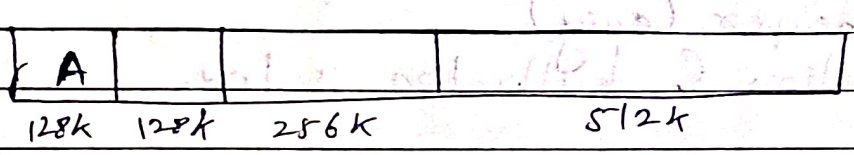
Eg. ① Request 128k (A)
② Request 256k (B)
③ Request 64k (C)
④ Request 250k (D)
⑤ Release B
⑥ Release A
⑦ Release C
⑧ Request 15k (E)
⑨ Release E
⑩ Release D

Ans:-



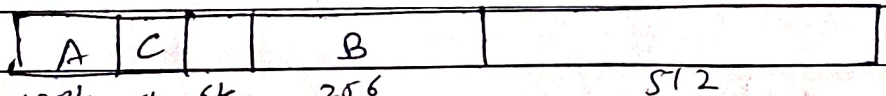
① 128k is too small for it. Therefore



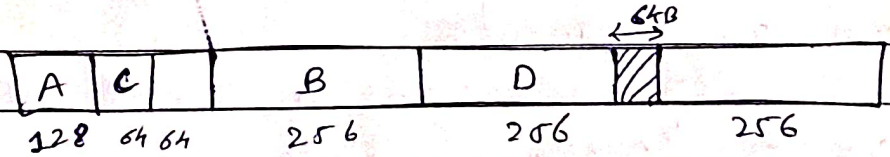


, IF=0, EF = 128+256+512k

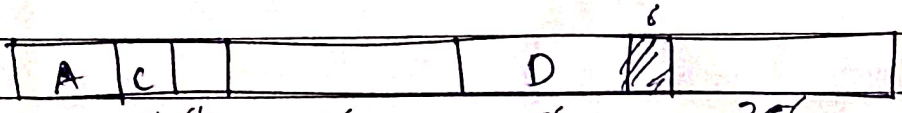
, IF=0, EF = 128+512k

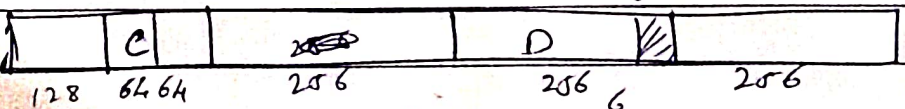


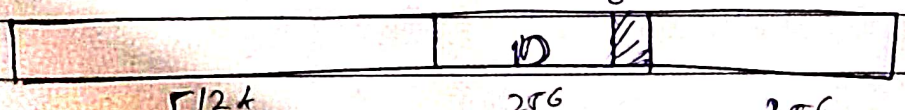
, IF=0, EF = 64+512k



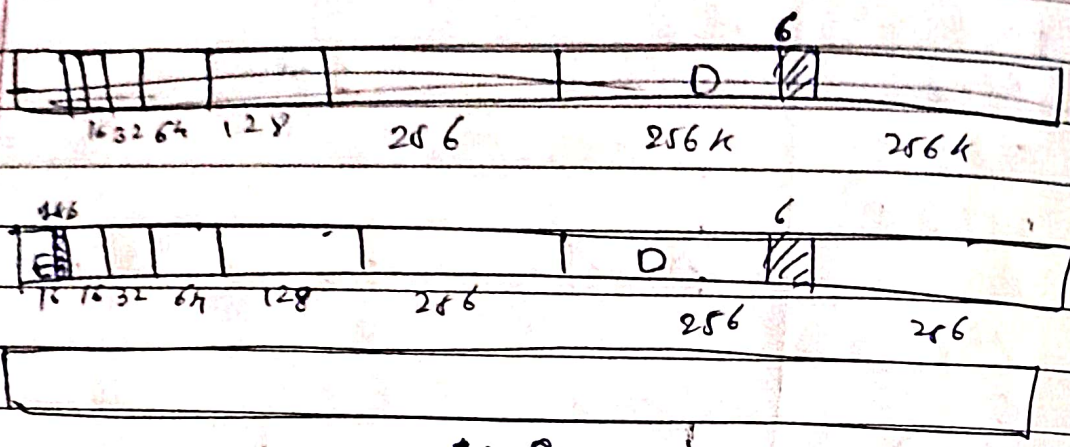
, IF=64B, EF = 64+256k







, IF=0, EF = 512+256



1MB

* Advantage
  - IF is less
  - EF is less
  - Merging is cheaper (easy)
  - cost of allocation & deallocation is low.

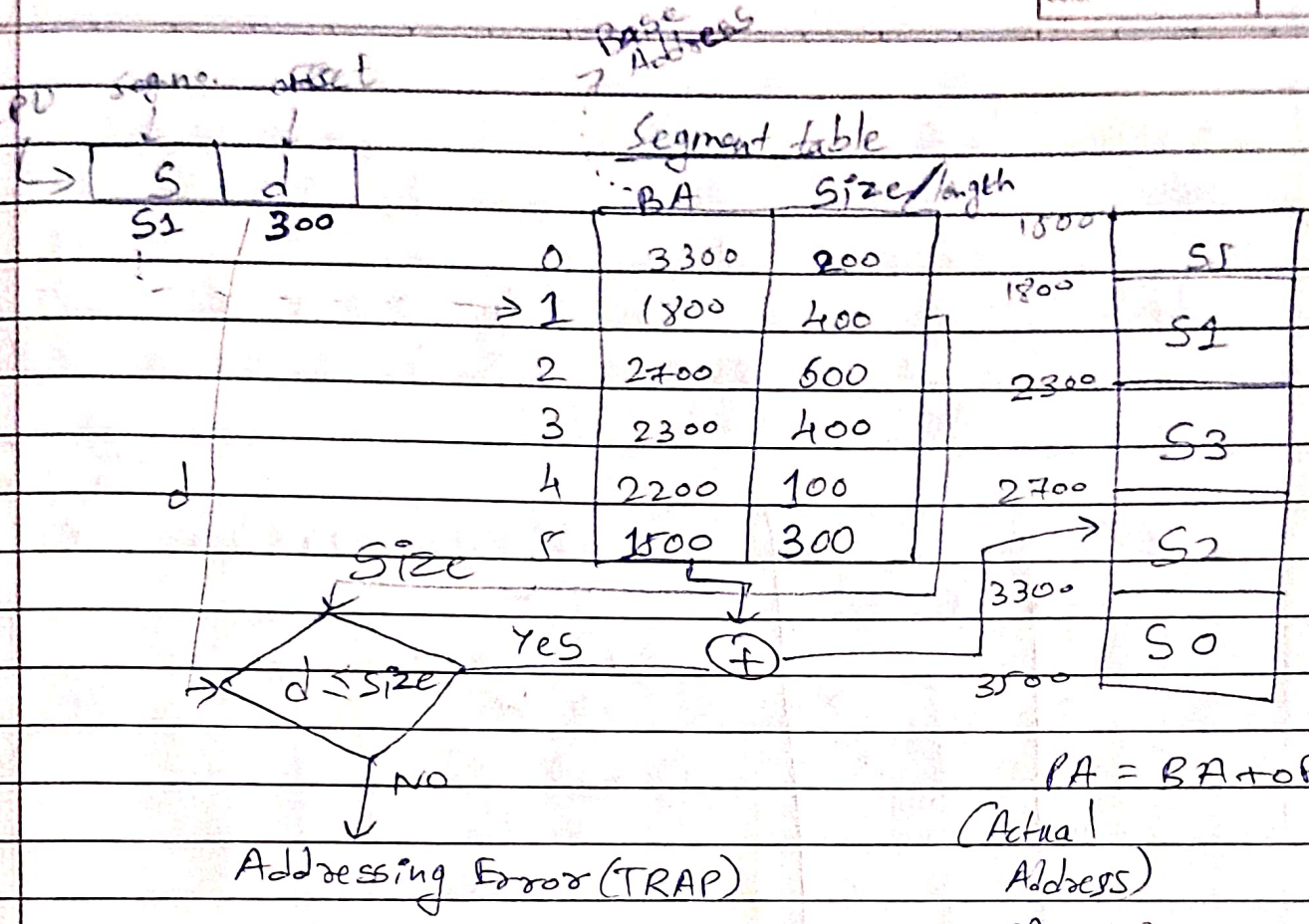
* Disadvantage
  - Time require to create two parts

14/08/25

* Segmentation

→ A process is divided into unequal size of chunks known as segments.
→ Segmentation is user view paging is OS view.
→ Stored in non-contiguous fashion.

LA | seg. no. | offset |



PA = BA + offset

(Actual Address)

PA = 1800 + 300 = 2100

* Page table Entry

Frame no.	control bits

Segment table entry (simple segmentation)

seg. no.	offset	control bits

Page table entry (paging with VM)

Frame No.	P	M	Protection	other control bits

P: Present - 1, Absent - 0

M: Modified - 1, Not modified - 0

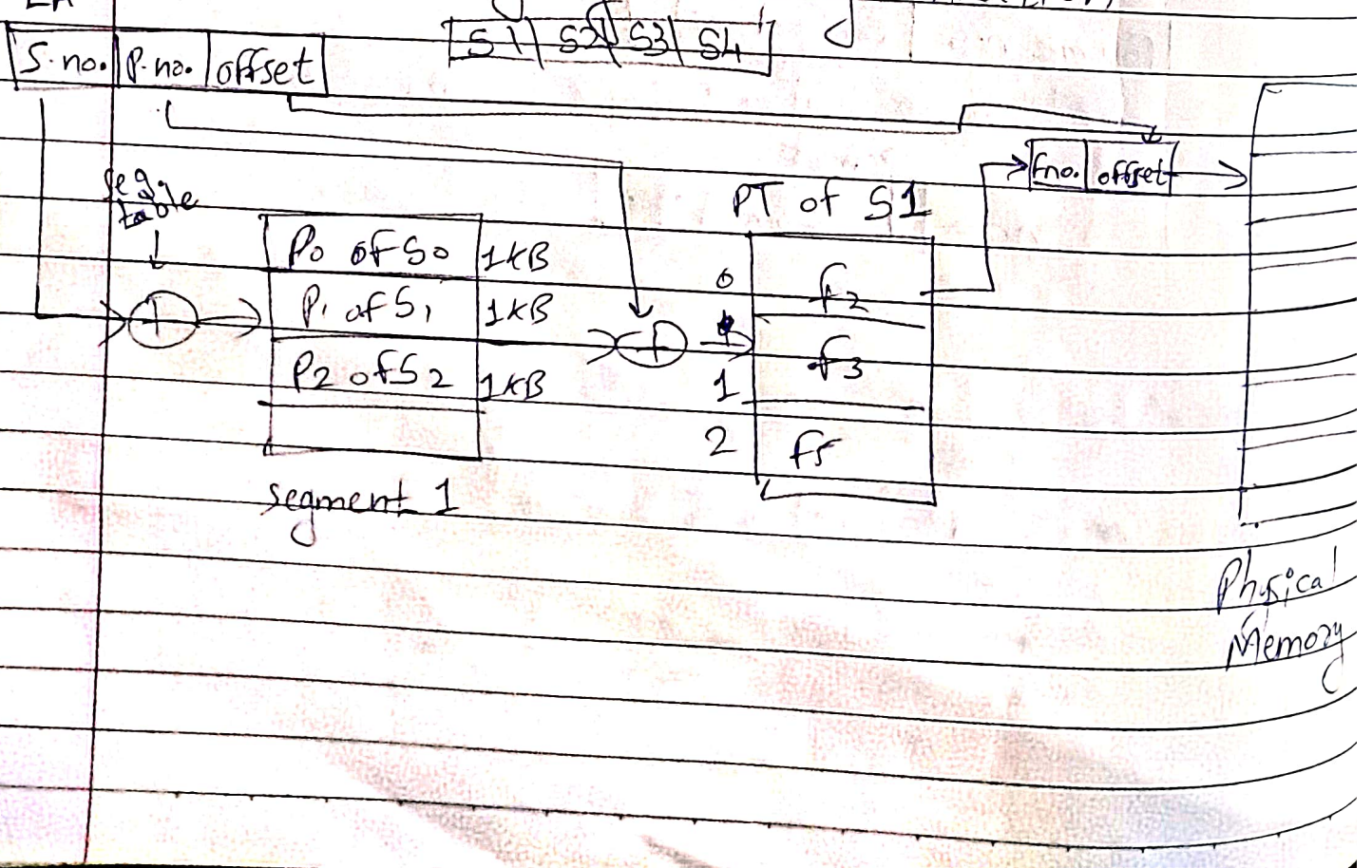
Seg. no.	BA	length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

① 0, 430 (0-699)
② 1, 11 (0-13)
③ 2, 100 (0-99)
④ 3, 425 (0-579)
⑤ 4, 95 (0-95)

Actual Address → Physical Address

① BA = 1219 + 430 = AA/PA
② ✓
③ TRAP
④ ✓
⑤ ✓

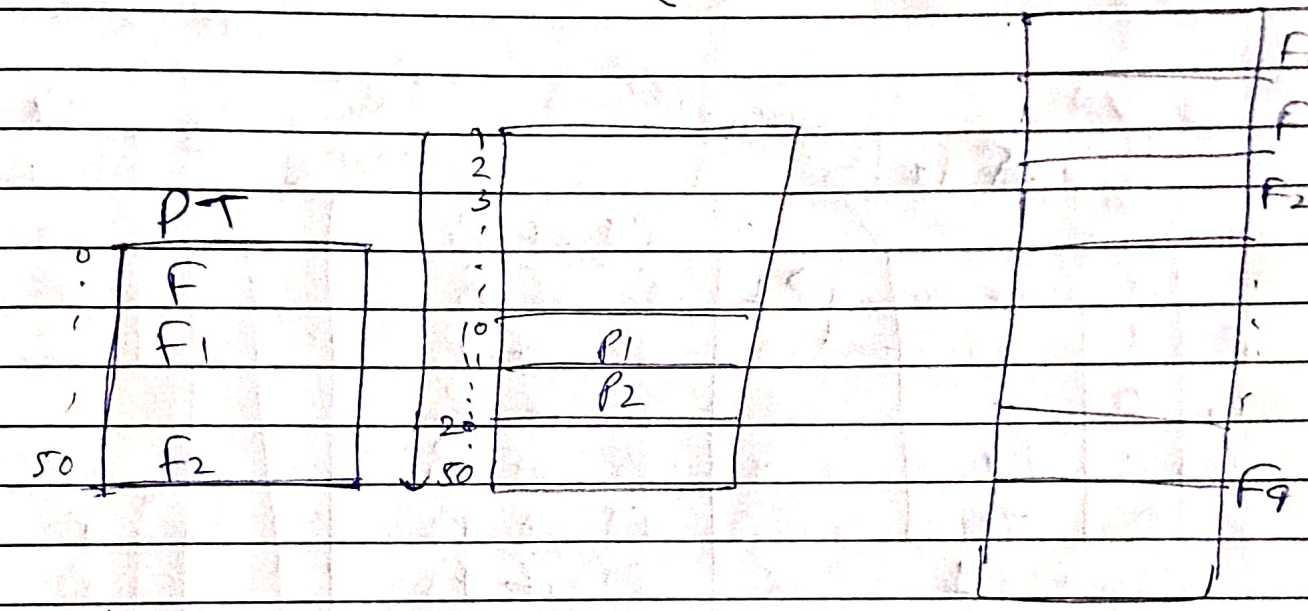
* Combined Paging and Segmentation



* Key points in Memory Mgmt.

(1) LA is translated into PA at runtime (dynamic)

(2) Process can be divided into parts which can be stored in ~~non-contiguous~~ non-contiguous manner.



- Resident set

↳ A portion of process that resides in Main memory at a given point.

~~Principle~~ Principle of locality / locality of reference

⇒ Tendency of a CPU to access same address over a short time period.

* Virtual Memory

↳ The S/m can be used as Primary which gives an illusion to user accessing P.M.

* Requirement of VM

(1) To execute the process size more than P.M. size.

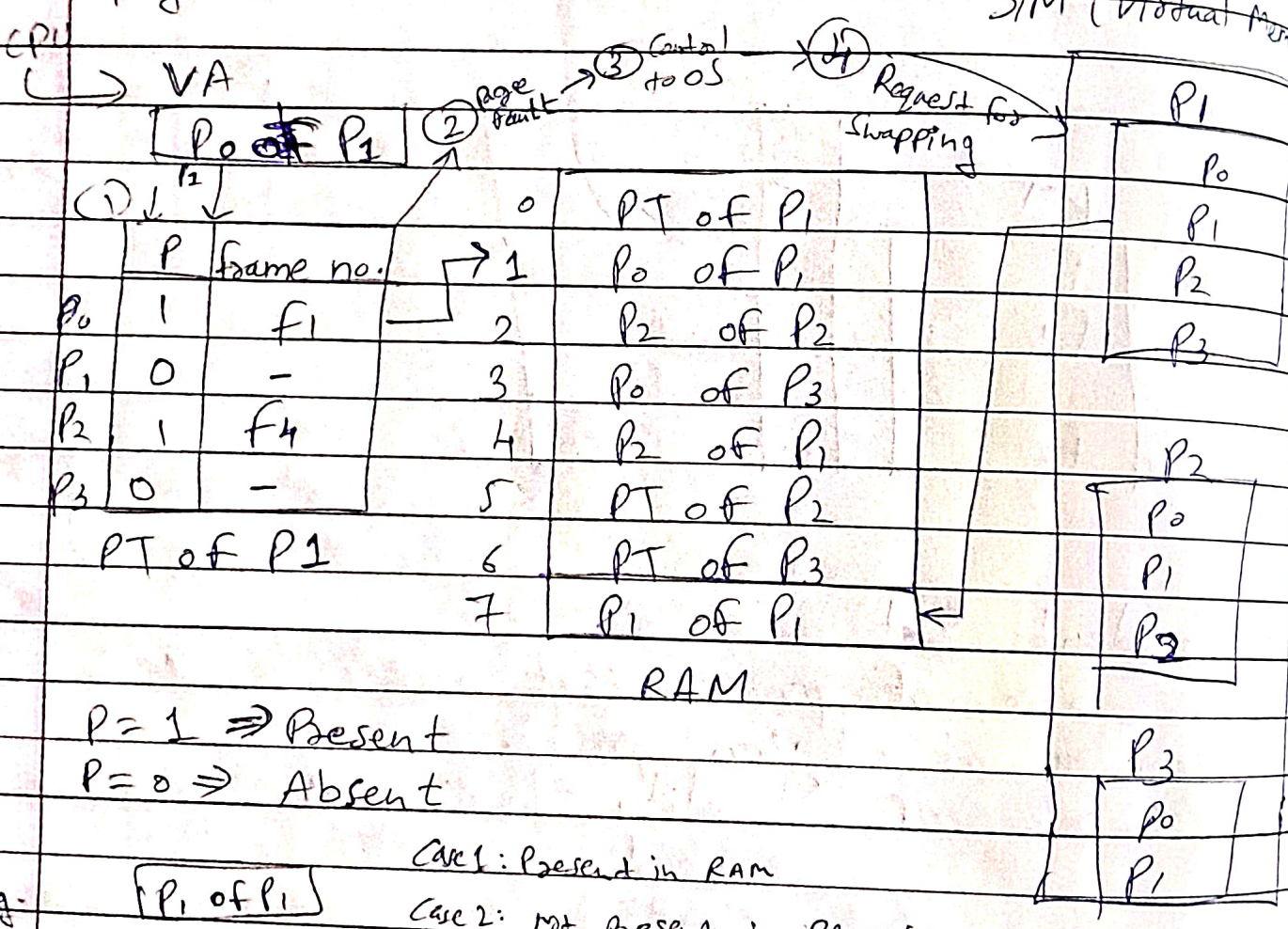
(2) To increase D.O.M.

* Execution of a process

(1) CPU generates virtual address (address in)

(2) As process starts its execution few pages are found in M/M.

(3) If page is not available in RAM, its PT ~~entry~~ entry is 0, which generates page fault.



P = 1 ⇒ Present

P = 0 ⇒ Absent

Case 1: Present in RAM

Case 2: Not present in RAM so Swap in RAM

E.g. P1 of P1

(1) P = 0 [Absent]

(2) page fault

(3) Control to OS [TRAP]

(4) Request for Swapping

⇒ It is called Page fault Service.

[We bring P1 of P1 in RAM (Swap-in)]

[Then change the page table]

	P	frame no.
P0	0 → 1	f1
P1	1	f7

- Case 3: Not present in RAM and also RAM is full so remove any one process and Swap-in our required process. And change page table accordingly. (Swap out)

- The solve the page fault and time taken in that is called Page fault Service Time.

- Points

(1) Search for the respective page in page table.

(2) check P bit in PT.
(i) if P=1 page is present in M/M.
(ii) if P=0 page is Absent in M/M.

(3) If page is present its physical address is calculated by CPU.

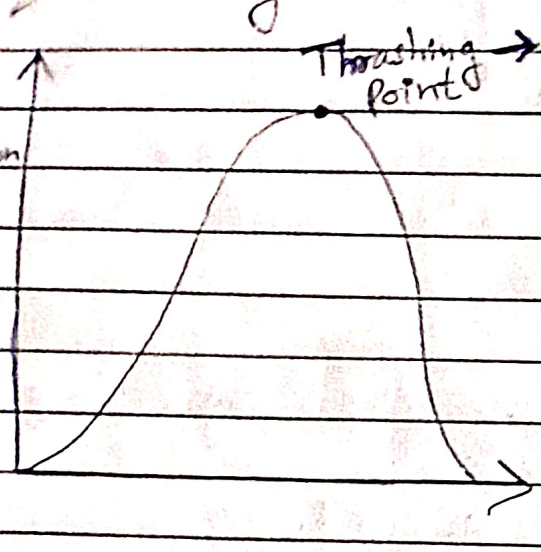
(4) If page is not found in RAM then go to step 5.

(5) Page fault is triggered.
↳ Trap generated by CPU will create a page fault to OS. CPU handover the control to OS.

(6) Serve the page fault.
Page fault service is a process to bring the page from Disk to RAM, called as Swap-in.
else go to step 7.
→ change the PT entry for page.

(7) Put the least used page back to disk called as swap-out. Now bring page to frame.

(8) Generate the physical address for the requested byte.



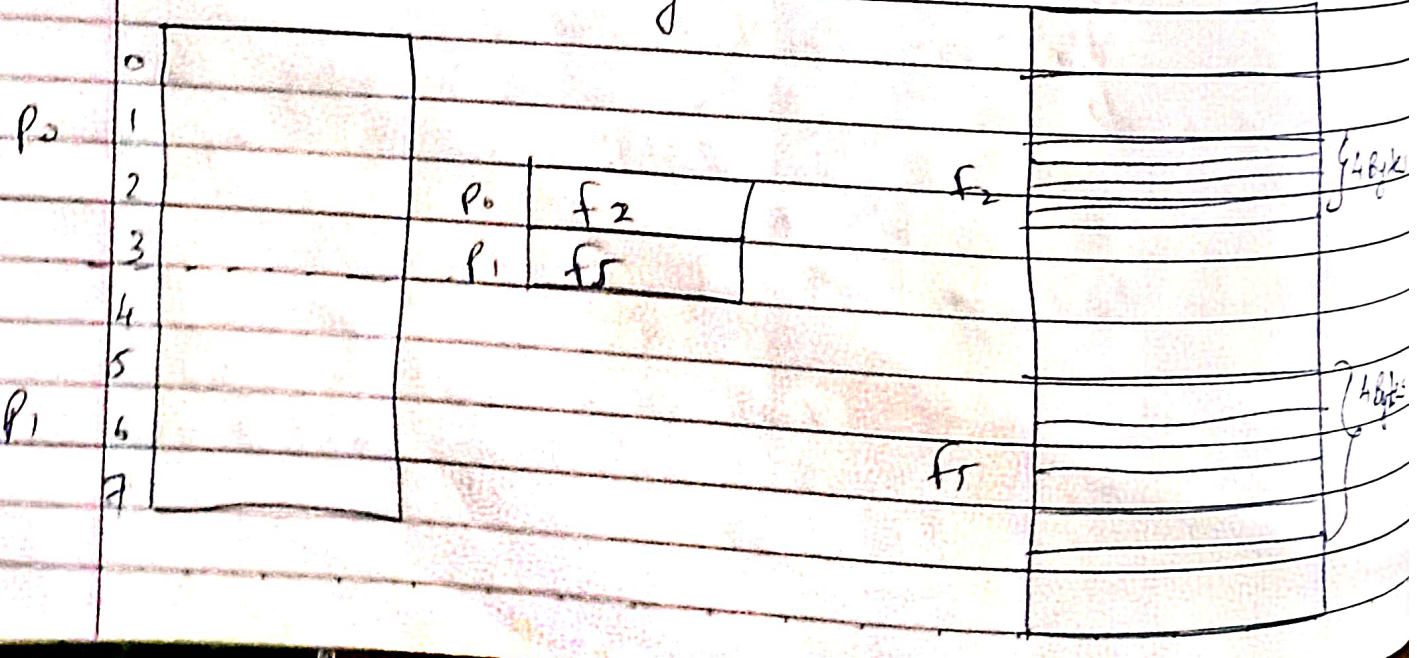
19/08/25

Q. Process size = 32 Bytes
Page size = 4 Bytes
Page Table entry = 1 Byte

→

- No. of pages = $\frac{32 \text{ Bytes}}{4 \text{ Bytes}}$ = 8

- Size of PT = 8 × 1 Byte
= 8 Bytes

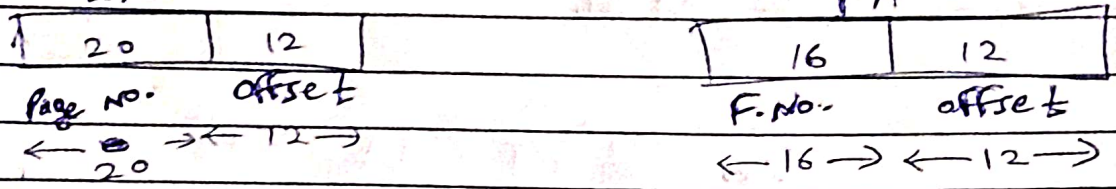


Q. PAS = 256 MB

LAS = 4 GB

Frame Size = 4 KB

→ 32 bit



No. of frames = 2^{16}

No. of pages = 2^{20}

Size of PT = No. of entries * No. of bit for each entry in PT

= $2^{20} \times 2^1$

= 2^{21} Bytes

PTE ⇒ 16 bits = $\frac{2^{20}}{2^{16}} = 2^4$ bits = 2^1 bytes

Size of PT = 2 MB [2^{21} Bytes]

Size of PT > Size of frame [∵ 2MB > 4KB]

Outer PT = $\frac{2MB}{4KB} = \frac{2 \times 2^{20}}{2^2 \times 2^{10}} = \frac{2 \times 2^{10}}{2^2} = \frac{2^2 \times 2^9}{2^2} = 2^9$

↑ Actual page size / page size

outer PT = 2^9

outer PT size = 2^{10}

inner PT = 2^{11}

Outer PT Size = $2^9 \times 2 = 2^{10}$

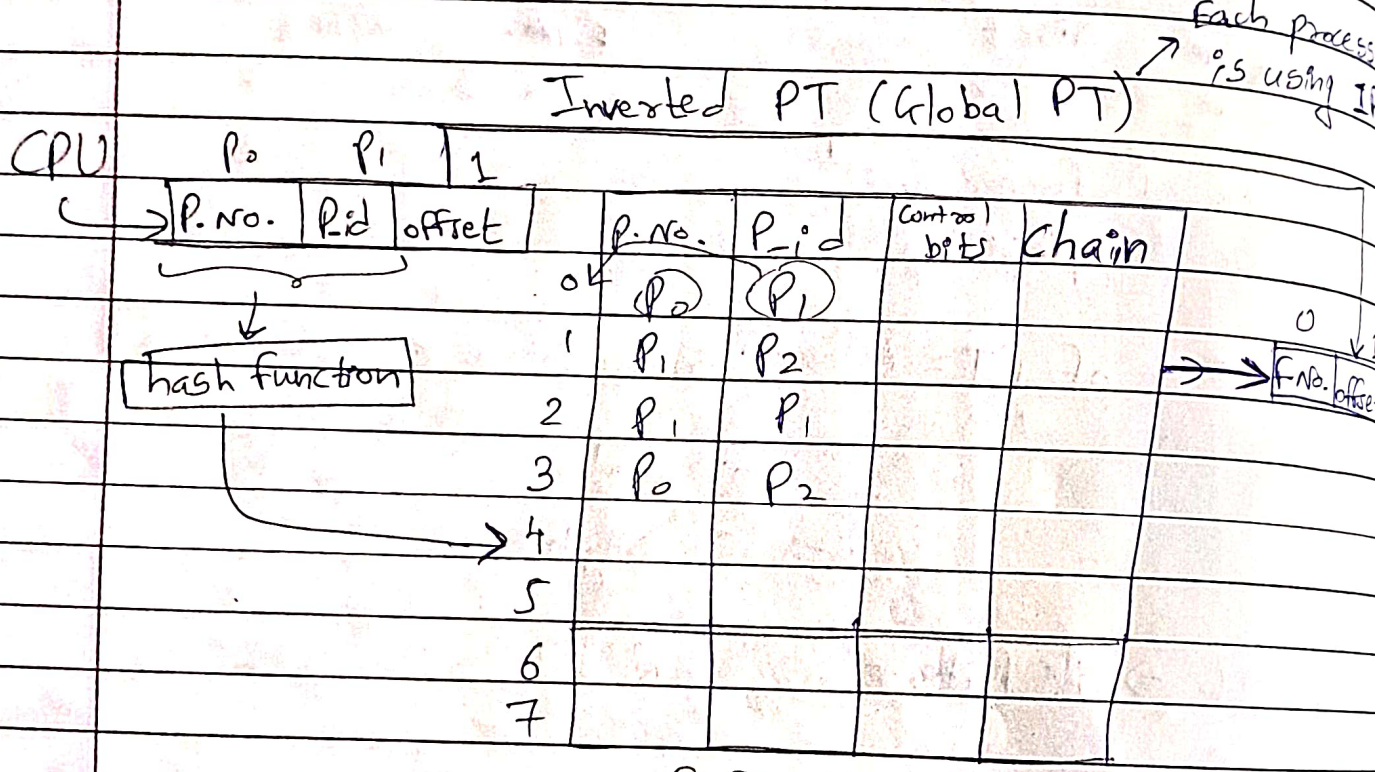
$\therefore 2^{10} < 2^{12}$

∴ No further division required.

* Inverted Paging

(1) Each process has its own PT.

(2) PT resides in M/M.



No. of Entries = No. of Frames

[Size of IPT is fixed

- IPT size is fixed.
- Linear searching (time consuming)
  Searching is time consuming

21/08/25

* Process synchronization

(*) Design goal of OS
(@) ↳ Managing multiple processes

(1) Multi-Programming
↳ Uni-processor (Interleaving)

(2) Multi-processing
↳ Multi-processors (overlapping)

(3) Distributed OS

* Difficulties in Concurrency
(1) Sharing of global resources. (Memory, I/O devices)

(2) Optimal management of allocated Resources

(3) Difficult to trace programming errors.

```
* void echo()
  {
1.  chin = getchar();
2.  chout = chin;
3.  putchar(chout);
  }
```

Shared procedure

- Uni-processor - Multiprogramming

P_1 (PCB)	PC = 2	P_2 (PCB)
1. chin = getchar() chin = 'a'		
2. P1 is interrupted		
		3. Chin = getchar(); 4. chin = 'b' 5. chout = 'b' display = 'b'
6. Chout = 'b' 7. display 'b'		

⇒ problem
↳ Shared variable/ procedure

⇒ Solution
↳ Permit only one process at a time.

✳ Multi-processing systems

P1 (Running on CPU1)	P2 (Running on CPU2)
(1) Chin = getchar(); Chin = a	
	(2) Chin = getchar(); chin = 'b'
(3) Chout = chin Chout = a	(3) Chout = chin chout = b
(4) putchar(chout) (b)	(4) putchar(chout) (b)

- Enforce

(1) P_1 invokes echo()
[EP1]?
(2) P_1 is interrupted after 1
(3) P_2 invokes echo()
P_2 blocked
(4) P_1 resumes and terminate
(5) P_2 process get into echo()

* problem
↳ single multiprocessor (shared resources)

* Solution
↳ Controlled access to shared resources

* Race condition
→ It occurs when multiple processes or threads read and write the data items and their final result depends in their order of execution.

e.g.: int x = 3 → global (shared) variable

P_1	P_2
read(x) = 1	
x = x + 1 = 2	
	read(x) = 2
	x = x + 1 = 3
display(x)	
3	
	display(x) x = 3

[Actual 2] [got]

22/08/25

* OS Concerns
1. Keep track of an processes
2. Allocation & deallocation of resources
3. protect data & resources
4. Maintain ~~memory~~ concurrency

* Process Interaction scenarios
↳ Degree of which they are aware abart each other's existance.
1. Process unaware of each other
   ↳ Competition
2. Process indirectly aware of each other
   ↳ co-operation by sharing
3. process directly aware of each other
   ↳ co-operation by sharing

* Process unaware of each other
↳ Multi-programming (Independent)

* Potential control problems
1. Mutual Exclusion
2. Deadlock
3. Starvation

* Mutual Exclusion
→ If two or more processes access to a single sharable resource, only one process should be allowed at a time.
- Critical Resource
  ↳ Sharable resource (printer)
- Critical Section

~~Deadlock~~

* General structure of each process
  Ra = Shareable resource

```
- void P1()                    - void P2()
  {                             {
    while (true)                  while (true)
    {                             {
      // preceding code
      enter(critical(Ra);         enter(critical(Ra);
      // critical section
      exit(critical(Ra);          exit(critical(Ra);
    }                             }
  }                             }
```

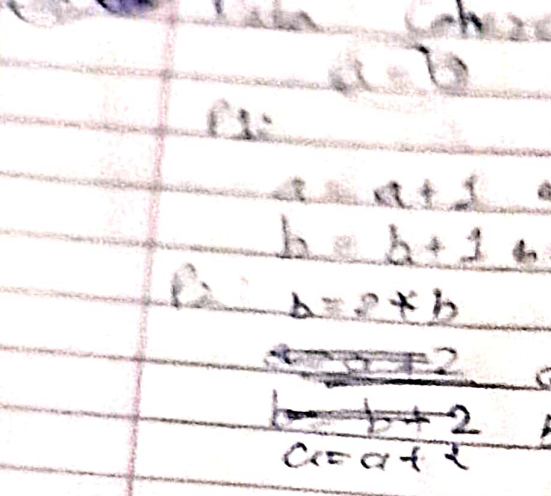
* Process are indirectly aware about each other's existence

→ Cooperation by sharing
→ data/file is shared
→ Integrity must be ensured

- int x = 1 //global

P_1	P_2
x++;	x = x+2
print(x) ⟷ Read	print(x)

- Potential control problems
① Mutual Exclusion (write operation)
② Deadlock
③ Starvation



* Requirements of Mutual Exclusion
1. Mutual exclusion must be enforced
2. A process that halts in ~~its~~ CS must do so without interfering other processes.
3. No Deadlock or starvation.
4. When CS is empty a process must be permitted without delay.
5. No assumptions are made for relative speed or no. of processes.
6. A process remains in CS for finite time.

* How to achieve the requirements of ME?
1. Leave the responsibility to processes
   ↳ High overhead
2. Use special Machine Instruction
   ↳ H/W support
3. Support of OS
   ↳ S/W support

Hardware Support - Mutual exclusion

(1) Interrupt disabling
↳ Uni processor allow interleaving
↳ Interleaved manner may interrupt the other running process.
→ disable interrupts
↳ guarantee ME

- while (true)
{
disabling interrupts
CS
enabling Interrupts

(2) Special Machine instruction
↳ Special machine instructions are carried out in atomic manner.
→ require 1 machine cycle not subject to interruption
(1) compare and swap
(2) exchange

- int compare and swap (int *word, int testval, int newval)
{
int oldval;
oldval = *word;
if (old val == testval) → Atomic
*word = newval; //check if CS is empty or not.
return oldval; → //CS mark as occupied.
}

```
const int n;
int bolt;
void P(int i)
{
    while (true)
    {
        while (compare and swap (bolt, 0, 1) == 1);
                CS
        bolt = 0;
        remainder code; [Remaining code]
    }
main ()
{
    bolt = 0;
    parbegin (P(1), P(2), ... P(n));
}
```

*
```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

-
```
int const n;
int bolt;
void P (int i)
{
    int key; = 1;
```

```
while (true)
{
    do exchange (key i, bolt);
    while (key != 0)
        CS;
    bolt = 0;
} remainder

main(){
    bolt = 0;
    P(1), P(2), ..... P(n)
}
```

```
while (true)
{
    do exchange (key, bolt);
    while (key != 0)
        cs;
    bolt = 0;
}: remainder

main(){
    bolt = 0;
    P(1), P(2), ..... P(n)
}
```

26/08/25

* Semaphore
  - ↳ software approach
  - ↳ Non-negative integer variable used for signaling among multiple processes.
  - ↳ Process co-operation using signals. (Sending & receiving signals)

* operations
  - (i) Initialize
  - (ii) Decrement (semwait)
  - (iii) Increment (Semsignal)

- struct semaphore
  {
    int count;
    QueueType queue; // waiting
  }

int S = 1;
S=1 //empty
S=0 //occupied

```
void semwait(semaphore S)
{
    S.count--;
    if (S.count < 0)
    {
        place this process in S.queue
        block this process
    }
}

void semsignal(semaphore S)
{
    S.count++;
    if (S.count <= 0)
    {
        remove a process from S.queue
        place in ready Q.
    }
}
```

* Types of Semaphore

(1) Binary
↳ Initialized with 0 or 1.

(2) Counting
↳ Initialized with N.

- struct semaphore
{
    enum { zero, one } count;
    queueType queue; (waiting)
}

void semWait (semaphore S)
{
    if(S.count == one) // CS empty (critical section)
        S.count = zero; // mark occupied
    else
        place the process in Queue;
}

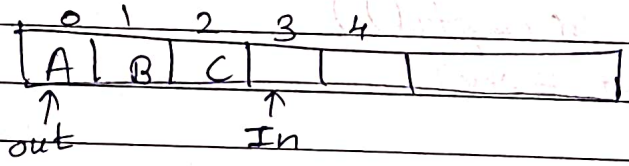
void semSignal (semaphore S)
{
    if (S.queue is empty())
        S.count = one;
    else
        remove a process from Queue;
}

28/08/25

* Producer - Consumer Problem
  ↳ one or more producers are generally items in shared buffer.
  ↳ one consumer is taking out the items from shared buffer.

problem:-
1. Both producer & consumer can not access the shared buffer at same time.
2. Producer can't data into full buffer
   consumer can't remove from empty buffer.

- Infinite Buffer



- In = Next index for producer to produce an item.
- out = next index consumber can consume item.

** Producer

```
                b=shared buffer
                in, out
while (true)
{
    b[in] = v;
    in++;
}
```

* Consumer

```
while (true)
{
    while (in <= out);   → buffer empty or not
    w = b[out];          → consumer blocked if
    out++;                 buffer is empty.
}
```

* Infinite buffer solution to PC

→ binary semaphore s = 1; (ME [Mutual Exclusion]) delay = 0; (consumer [stop] Empty)
int n; // No. of items in Buffer

- producer
  - produce() → local buffer
  - append() → add item buffer
- Consumer
  - take() → take item from bf
  - consume() → consume at cons.

```
void producer()
{
    while (true)
    {
        produce();
        semwait(s);
        append();      } CS [Critical section]
        n++;
        if (n==1) { semsignal(delay)};
        semsignal(s);
    }
}
```

```
void consumer()
{
    semwait(delay);
    while(true)
    {
        semwait(s)
        take();
        n--;
        semsignal(s);
        consume();
        if(n==0)
            semwait(delay);
    }
}
```

Incorrect sol^n
correct in Book.

* General solution using semaphore

Counting Semaphore

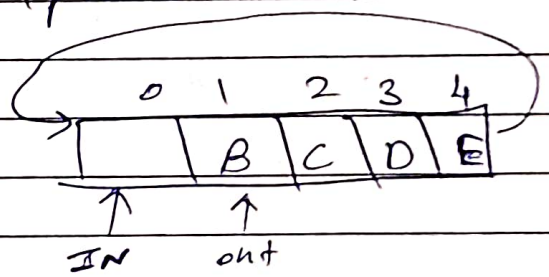
Semaphore n = 0; ⇒ No. of full slots/items in buffer
Semaphore s = 1;

```
void producer()
{
    while(true)
    {
        produce();
        semwait(s);
        append(a);
        semsignal(s);
        semsignal(n);
    }
}
```

- void consumer ()
{
while (true)
{ semwait(n);
semwait(s);
take();
semsignal(s);
consume();
}
}

* Bounded Buffer solution (producer - consumer)
n = size of buffer
in, out

0 to n-1



⇒ producer
while (true)
{
while ((in+1) % n == out); ← Full buffer
/* do nothing */
b[in] = v;
in = (in+1) % n

}

⇒ Consumer ()
while (true)
{
while (in == out);
/* empty buffer */

```
    w = b[out];
    out = (out+1) % n;
}
```

* Bounded Buffer Solution using Semaphore

e = no. of empty slots

n = no. of full slots

s = mutual exclusion