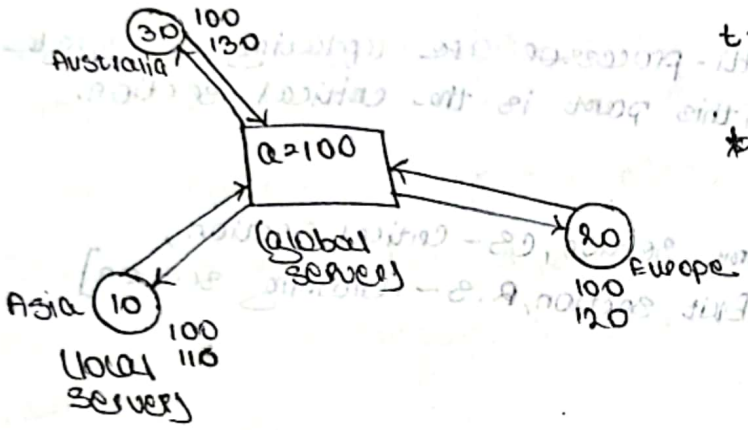
LECTURE 11 TUESDAY DATE: 25/07/23

PROCESS SYNCHRONISATION



t = 0 [All viewers are seeing 100 likes]

t = 1 [10 likes are added by Asia, so all viewers will view 110 likes. Also, 20 likes are added by Europe, so all viewers are supposed to see 130 likes, but due to lack of synchronization, 120 likes from Europe are viewed by all]

counter = 5
counter ++ → T_0, T_1
counter -- → T_2, T_3, T_4, T_5
(part of critical section)

OUTPUT: 5

producer:
T_0: register1 = counter
T_1: register1 = register1 + 1
T_2: counter = register1

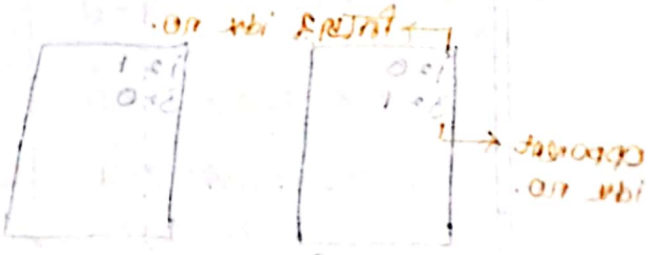
consumer:
T_3: register2 = counter
T_4: register2 = register2 - 1
T_5: counter = register2

If there is no synchronisation

5 = T_0 — counter ++
6 = T_1
6 = T_2
5 = T_3 — counter --
4 = T_4
4 = T_5

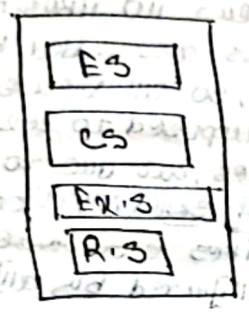
OUTPUT: 4

[miscalculation occurs] → Race condition occurs



CRITICAL SECTION

* Critical section: When multi-processor are updating a single variable, this part is the critical section.

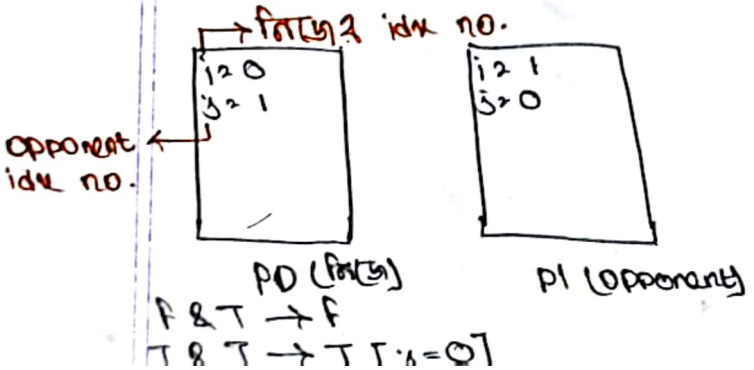


[ES - Entry section, CS - Critical section, Ex.S - Exit section, R.S - Remaining section]

* When there are two process side by side, their CS cannot be executed together. They are need to be mutually exclusive.
* Both the CS need to progress.
* Both the process need to wait for some time.
* This is preemptive process.

PEARSON'S SOLUTION [Software Based sol^n]

* turn:
* flag:
* Peterson's solution is applicable only for 2 processes.



F & T → F
T & T → T [i=0]

t = 0
t = 1

turn: 0

flag: T | F
0 1

flag: F | T
0 1

* F अगर synchronise हो.

```
do {
  flag[i] = True;
  turn = j
  while (flag[j] && turn = j);
  [CS]
  flag[i] = False;
  while (True);
```

Example

- Each process takes 2ms to execute. Process 1 gets executed first.
- Context switch will occur after 6ms
- Critical section contains 4 statements
- Remainder section contains 2 statements
- turn = 0
- Flag[0] = FALSE, Flag[1] = TRUE

time (ms)	Process 0 (i=0, j=1)	Process 1 (i=1, j=0)
0-6		Flag[1] = true turn = 0 while loop condition
6-12	Flag[0] = True turn = 1 stuck in while loop	
12-18		CS1 CS2 CS3
18-24	stuck in while loop	
24-30		CS Flag[1] = False RS1
30-36	while loop condition CS1 CS2	
36-42		RS2
42-48	CS3 CS4 Flag[0] = False	
48-54	RS1 RS2	

turn = 0 / 1

Flag = [F | F] (0, 1)

```
do {
Flag[i] = true;
turn = j;
while (Flag[j] && turn == j);
        CS
Flag[i] = false;
        RS
} while (true);
```

0-6: Process 1: T & F → F, while loop runs

6-12: Process 0: T & T → T, stuck in while

12-18: Process 1: CS → CS1, CS2, CS3

18-24: Process 0: stuck in while loop

24-30: Process 1: CS → CS4, Flag

30-36: Process 0: RS1, T & F → F, while loop runs

36-42: Process 1: RS2 → T

LECTURE 12

SUNDAY

DATE: 30/07/23

HARDWARE-BASED SOLUTION TO THE CRITICAL SECTION PROBLEM

* Locking : 2 Processes + CPU resources
* word: some unit of memory
* atomically: ஒரே single line ல modify & test செய்வது

TEST_AND_SET( )

```
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

```
do {
    while test_and_set(&lock);
        /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);
```

lock = T F F F F

P0 P1

* Initially, lock for P0 & P1 → F ஆக்கப்படும்
* pointer of (*target) for P0 is send to boolean test_and_set where the current boolean (F) is stored in rv and is returned into do. *target is updated to T.
* In do, the value from the address of the returned rv is used. Here, it is F.
* If the value of lock is F, then P0 goes into CS. During this execution, only P0 remains in CPU resource, P1 waits as *target becomes T, causing the while loop to run for infinite times until lock is updated to F and P1 is allowed to enter CPU.

Compare_and_swap()

```
int compare_and_swap
    (int *value, int expected,
     int new_value){
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

do{
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
        /* critical section */
    lock = 0;
        /* remainder section */
} while (true);
```

* Critical section
A process, lock = 1.
Otherwise,
lock = 0, so
that critical
section is visited
by

* Same as TEST_AND_SET()

lock = 0 × 0 × 0
(P0) (P1)

MUTEX LOCKS

* Mutex is a variable
* acquire lock — test and set
* release lock — set

```
acquire() {
  while (!available)
  ;/* busy wait */
  available = false;;
}
release() {
  available = true;;
}
```

* same as test_and_set()
* If available = True, then the while loop continues to run, and when the loop stops, available = false and the process enters the CS. After execution in CS, available = True allowing other process to enter the acquire lock.

```
do {
  acquire lock
    critical section
  release lock
    remainder section
}
while (true);
```

SEMAPHORE

* Used for multiple process synchronization
* wait( ) and signal( )

* Semaphore is a variable, which represents how many processes are present in a critical section for a given time.
* wait(s) → cs → signal(s)
* eg s = 3 → 3 cores

Semaphore

Binary semaphore
- s = 0/1
- 1 is set as default
- Mutex lock

Counting semaphore
- s = 0/1 ... n (n → total no. of resources)
- value of s is initialized using n [s = n]
- multiple threads can be operated at a time (adv)

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

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

* For Binary semaphore, initially the value of s is 1. When it enters wait( ), it doesn't enter the while loop, s is decremented by 1 and then it enters the while loop and the loop continues to run. process enters cs. After execution, s enters signal where it is incremented by 1

#S = 3

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

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

* When S enters wait, it is decreased by 1 S = S 2. 2< S, so, n threads can operate together in the while loop and execute two task in their respective CS.

* When S = 2 enters signal after execution of the processes. S is increased by 1 S = S 3. Thus, the process is executed in CS and leaves leaving free resource for other process to execute.

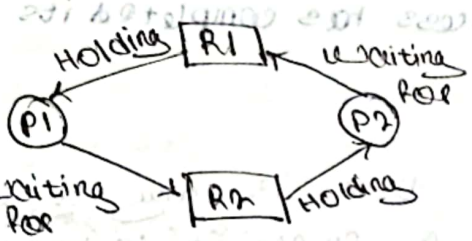
DEADLOCK & STARVATION

* When a process which is already using a resource and waiting sends requests to other resources, then deadlock occurs.

* If we remove processes from the list using LIFO, then processes wait indefinitely within the semaphore, it this is starvation.

LECTURE 13 TUESDAY DATE: 01/08/23

DEADLOCK

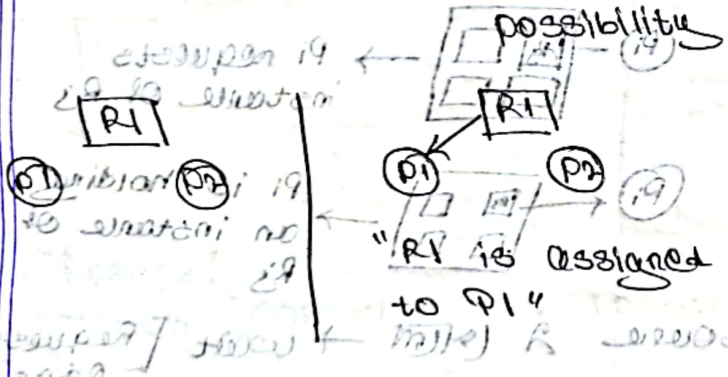


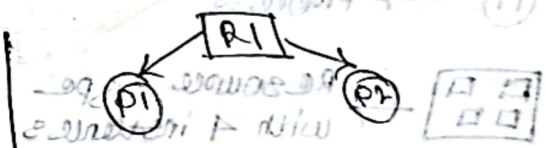
* Box — Resource → memory
* Circle — Process

* R1 is ~~not~~ used by P1 and R2 is used by P2.
* P1 is waiting for R2. But, since ~~the~~ P2 is using R2, P1 has to wait ~~till~~ until R2 leaves R1.

NECESSARY CONDITIONS FOR DEADLOCK

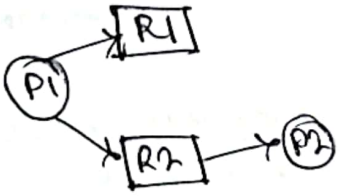
(1) Mutual exclusion: When two or more processes ~~hold~~ share a specific resource, then there is a possibility of deadlock.





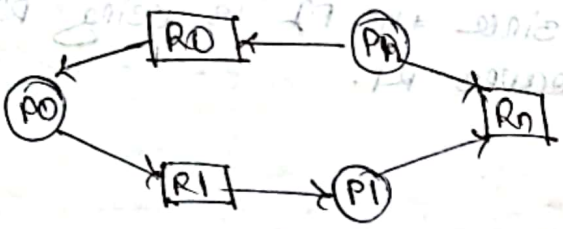
"R1 is shared between P1 & P2" — Deadlock occurs

(2) Hold and wait: When a process holds a resource and waits for another resource, deadlock may occur.



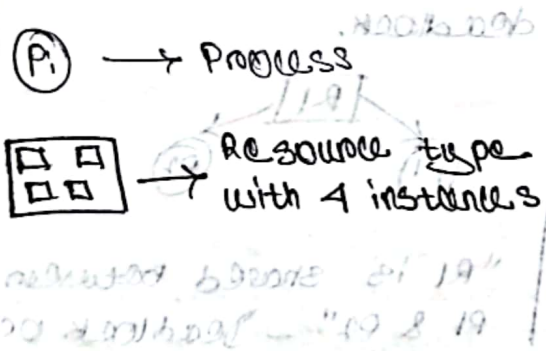
(3) No Preemption: No resource can be released by only voluntarily by the process holding it, after that process has completed its task.

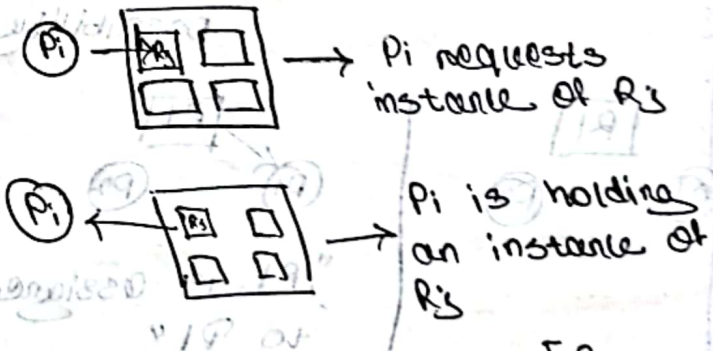
(4) Circular wait: We'll see if there is any cycle or not. If there's a cycle, it is has Deadlock.



* cycle না থাকলে → no Deadlock
* cycle থাকলে → possibility of Deadlock

RESOURCE ALLOCATION GRAPH





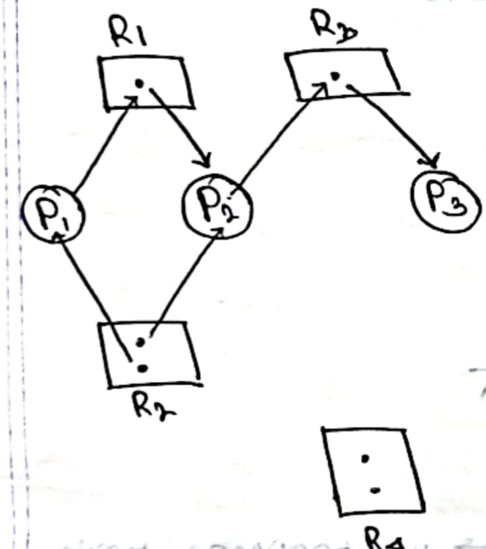
* Arrow process থেকে resource এ গেলে → wait [Request edge]
* Arrow resource থেকে process এ গেলে → hold [Assignment edge]

* Cycle + Wait & Hold থাকতে হবে to check for Deadlock.

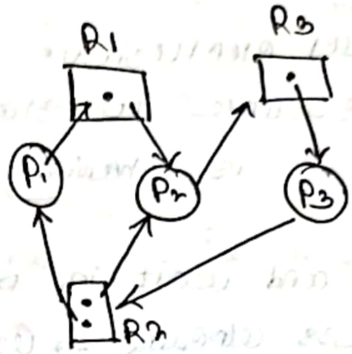
P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E = {P1 → R1, P2 → R3,
R1 → P2, R2 → P2,
R2 → P1, R3 → P3}



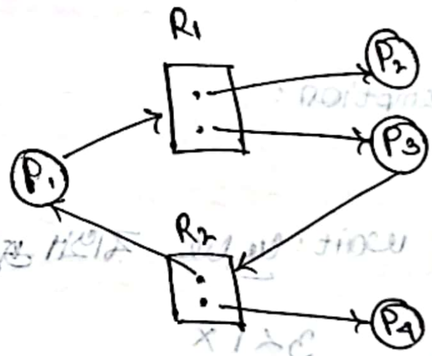
"Example"



Cycle 1: P1 → R1 → P2 → R3 → P3 → R2 → R1

Cycle 2: P2 → R3 → P3 → R2 → P2

DEADLOCK !



Cycle : P1 → R1 → P3 → R2 → P1 [*P2 & P4 are not in cycle]

NO DEADLOCK

METHODS FOR HANDLING DEADLOCKS

(1) Prevention: System will never enter a deadlock state

(2) Avoidance: Resource share করার আগে check করে নিবে whether or not the resource can be shared to the process.

(3) Detection and Recovery: A threshold is given. Check whether or not there is ~~multi~~ any deadlock. Either the process is terminated ~~and~~ or process is forcefully ~~terminated~~

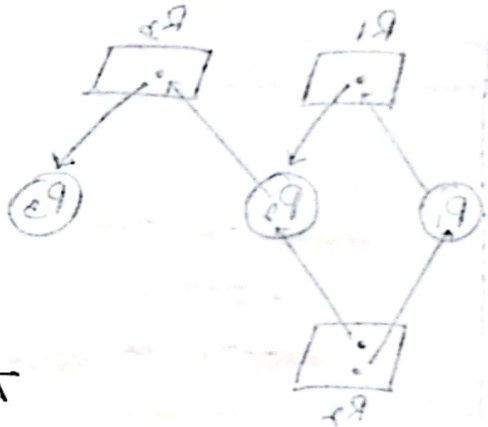
DEADLOCK PREVENTION

(1) Mutual exclusion: is prevented by not allowing a resource to share resources between processes when it is sharing to one process already.

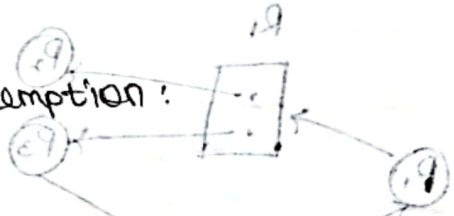
(2) Hold and wait is prevented by not allowing any process already is using a resource to wait for other resource.

Disadvantages: (1) Low resource utilization

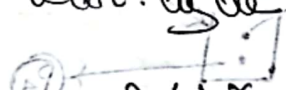
(2) Starvation



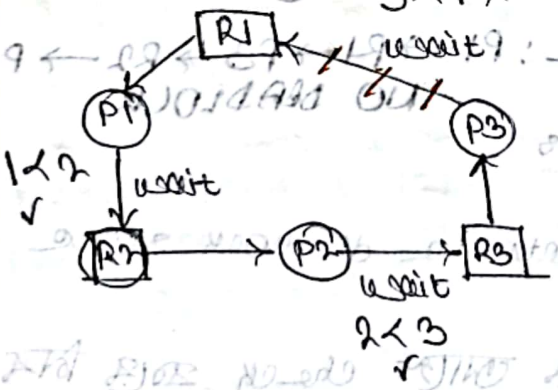
(3) No preemption:



(4) Circular wait:



$P_i < R_j$ → process wait for resource



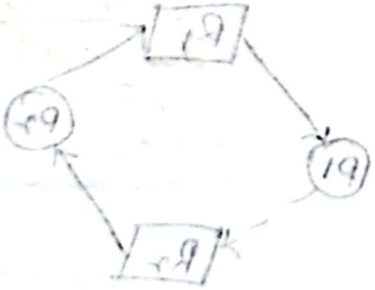
DEADLOCK AVOIDANCE

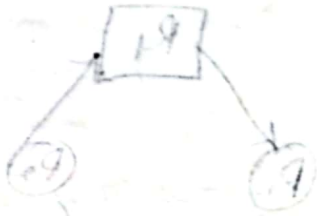
* Deadlock ಅನ್ನು detect ಮಾಡಿ and deadlock ಅನ್ನು avoid using algorithms to get a safe state where no deadlock occurs.

* Safe state : A state which can provide resources to each of the process available without any deadlock.

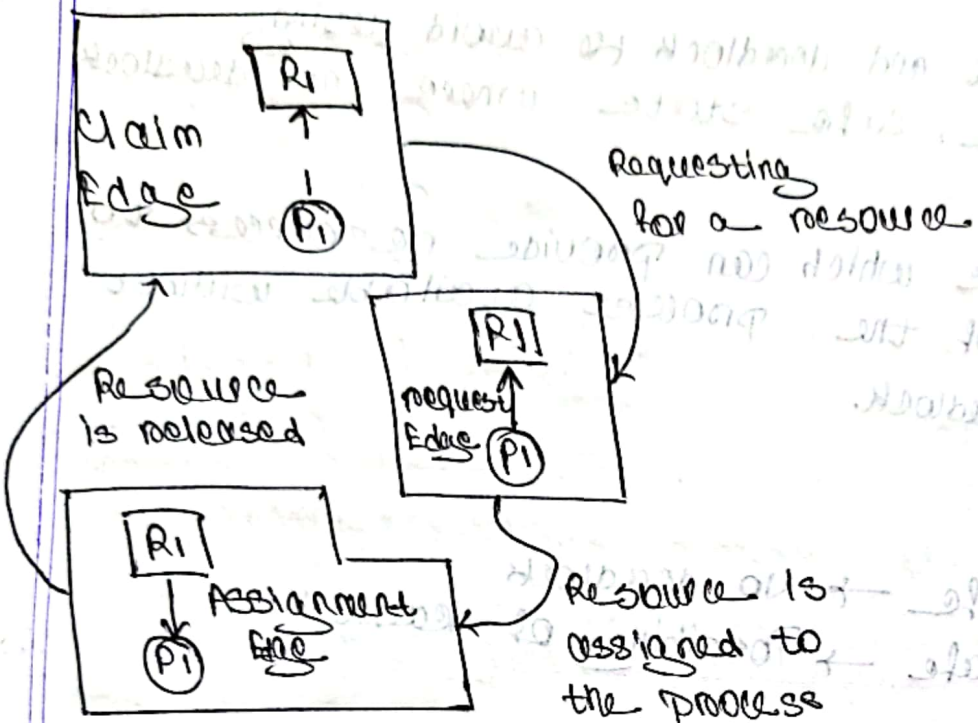
Safe → No deadlock
Unsafe → Possibility of deadlock

* Using Banker's Algorithm, we can achieve a safe state.

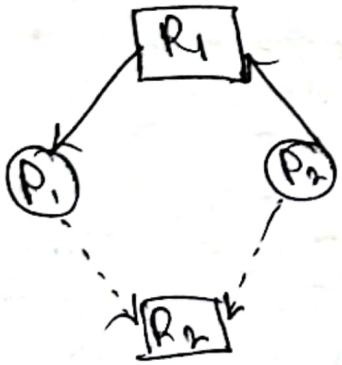




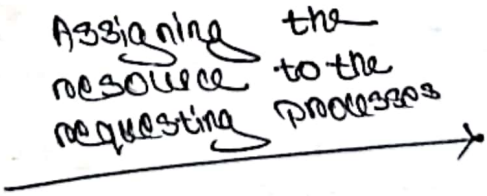
RESOURCE - ALLOCATION - GRAPH ALGORITHM

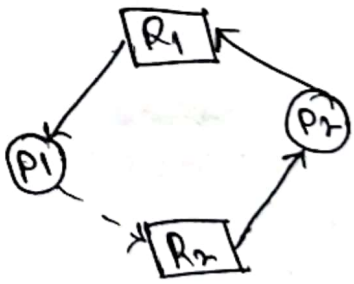


* Request Edge: Process requests for resources.
* Assignment Edge: Resource is used by process.
* Claim Edge: Process may request for resource.



Resource allocation graph for deadlock avoidance





An unsafe state in a resource allocation graph

LECTURE 14

SUNDAY

DATE: 06/08/23

BANKER'S ALGORITHM

* Banker's algorithm gives us whether we can execute the processes using the given resources.

	R1	R2	R3
available	2	2	3

R1 has 2 instances

Available – no. of instances available currently in the given resources

Max

	R1	R2	R3
P1	2	3	0
P2	6	5	2

Max – 2D matrix
– total no. of resources needed for process.

Allocation

	R1	R2	R3
P1	1	0	0
P2	2	3	2

Allocation – no. of resources currently given to a process

Need

P1	1	3	0
P2	4	2	0

Need = Max – Allocation
– No. of resources needed to complete execution of a given process.

① Allocation

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Need

	R_0	R_1	R_2
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

* for complete execution of a process, no. of instances of given resources in need for a specific process must be less than or equal to the instances in resources i.e. no. of in available.

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

* Available is incremented by the allocation of executed process

$P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0 \rightarrow P_2$ [safe sequence]

Available

3	3	2
2	0	0
5	3	3
2	1	1
7	4	3
0	0	2
7	4	5

0	1	0
7	5	5
3	0	2
10	5	7

Allocation

	R_0	R_1	R_2
P_0	1	0	1
P_1	2	1	2
P_2	3	0	0
P_3	1	0	1

Available

2	1	1

$+ \; 1 \; 0 \; 1$

$3 \; 1 \; 2$

$+ \; 3 \; 0 \; 0$

$6 \; 1 \; 2$

$+ \; 1 \; 0 \; 1$

$7 \; 1 \; 3$

Max:

	R_0	R_1	R_2
P_0	2	1	1
P_1	5	4	4
P_2	3	1	1
P_3	1	1	1

Need

	R_0	R_1	R_2
P_0	1	1	0
P_1	3	3	2
P_2	0	1	1
P_3	0	1	0

$P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow$ X

No safe sequence can can generated, so there's a possibility of deadlock

Extension of ①

*A request is given. we have to check whether the request will be accepted or not.

check for validity :-

request ≤ Need → True

request ≤ available → True

request

P1 → 1 0 2 (given)

[P1 needs extra resource which is taken from available, so available is decreased, allocation of P1 increases and need for P1 is updated]

Allocation

R0	R1	R2
0	1	0
~~2~~ 3	~~0~~ 0	~~0~~ 2
3	0	2
2	1	1
0	2	2

Max

R0	R1	R2
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Need

R0	R1	R3
7	4	3
~~1~~ 0	~~2~~ 2	~~2~~ 0
6	0	0
0	1	1
4	3	1

available

3 3 2

− 1 0 2 → request of P1

2 3 0

+ 3 0 2

5 3 2

+ 2 1 1

7 4 3

+ 0 0 2

7 4 5

+ 0 1 0

7 5 5

+ 3 0 2

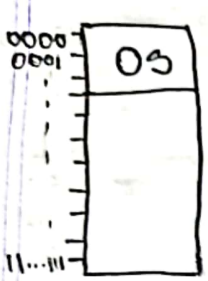
10 5 7

LECTURE 15

TUESDAY

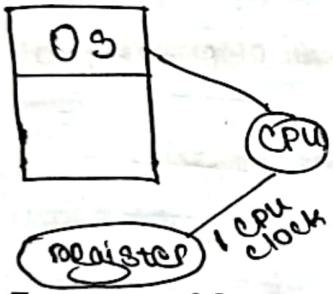
DATE: 07/08/23

MAIN MEMORY



main memory

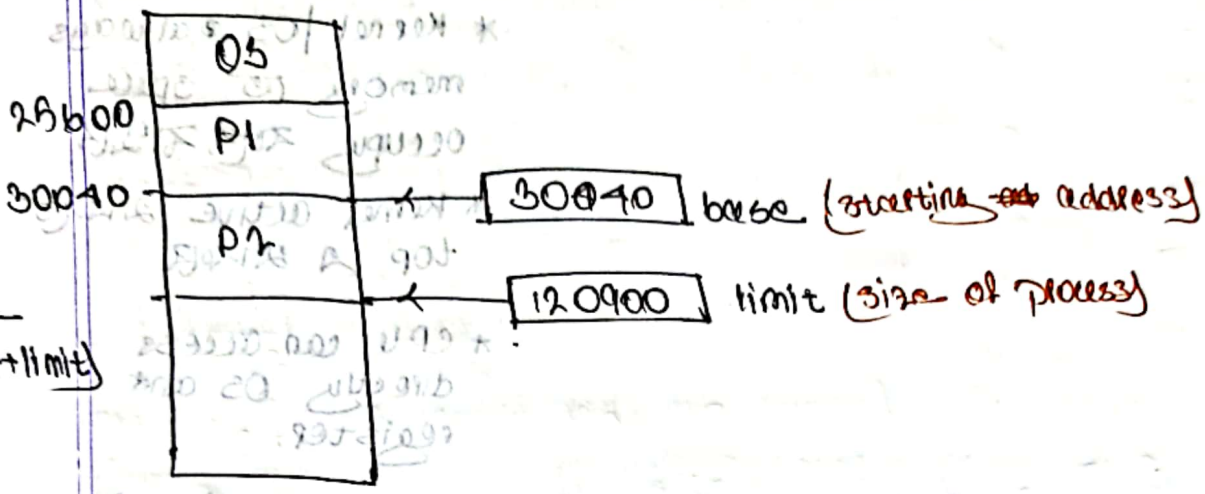
* Kernel /OS always memory তে space occupy করে থাকে
* Kernel active থাকতে top এ থাকে
* CPU can access directly OS and register.
* Access unit of register is 1 CPU clock.



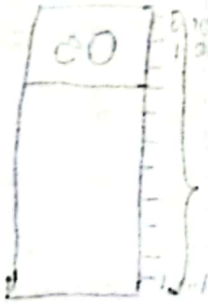
[ages to CPU first OS its faster than OS]

* Cache is slower than register but faster than OS
* একটা জিনিস বার বার RAM থেকে CPU তে load করা লাগলে, সেই জিনিসগুলো cache এ store করে রাখা হয়, লাগলে cache থেকে CPU তে load করে।
* Register > Cache > OS

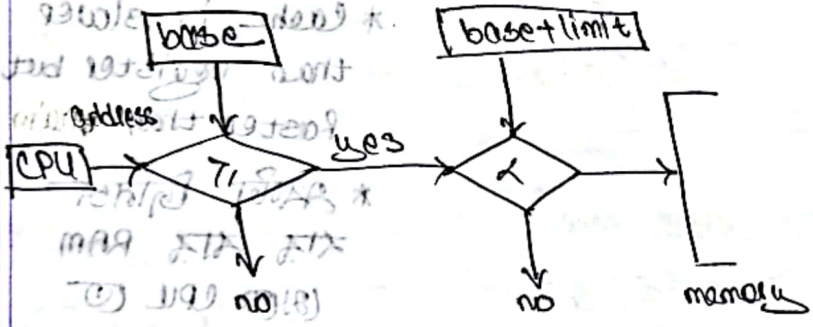
BASE AND LIMIT REGISTERS



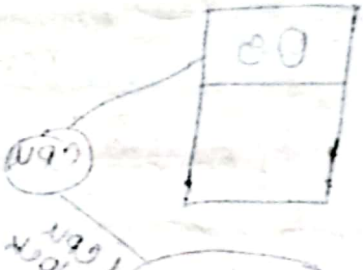
base – base+limit



HARDWARE PROTECTION



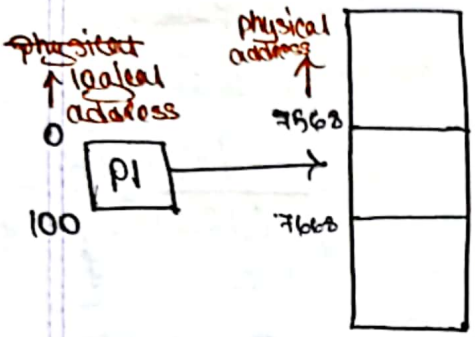
trap to OS monitor – addressing error

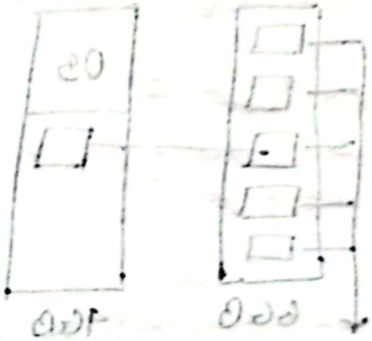


* Address from CPU ≥ base of process → YES
* Base+limit > Address → YES

→ satisfies → goes in memory

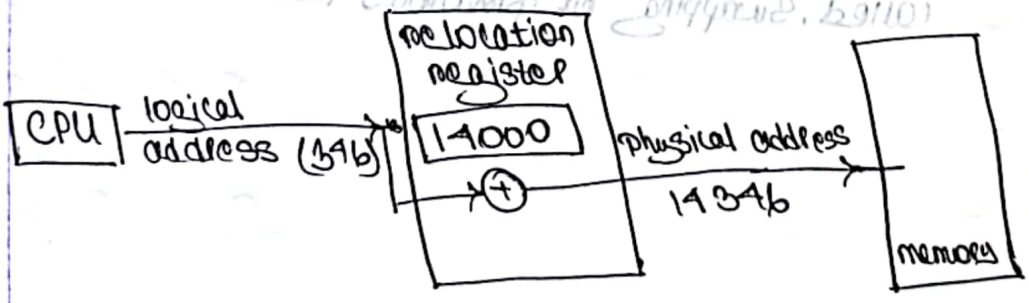
ADDRESS SPACE





Physical address = logical address + base

* If we want to load 10th line of P1 from logical address to main memory, physical address = 10 + 7568 = 7578
* MMU creates mapping between logical address & physical address. * MMU does mapping dynamically.
* Here, base is called relocation register.
* MMU তে কাজ করতে হলে RAM এ continuous allocation থাকতে হবে → (condition for MMU)



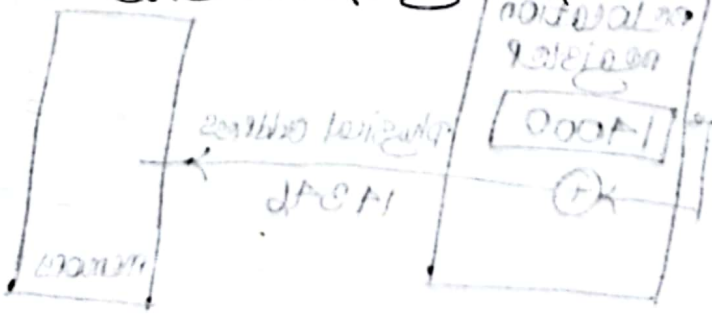
* CPU যখন process এর logical address দেয়, Relocation register এর মধ্যের address এর সাথে logical address যোগ হয়ে relocation address হয়। এই relocation address memory তে যায়।

DYNAMIC LOADING

routine. [6GB is divided into chunks]

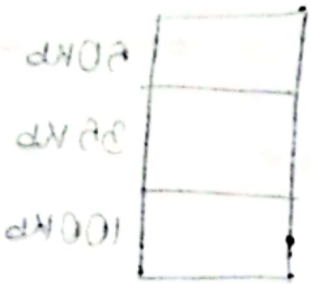
* যখন নতুন routine call হবে, তখন routine 4GB তে যাবে।
* যখন 4GB full and routine call হবে, existing routine in 4GB is swapped with incoming routine.

Dynamic loading: When a large process, tries to be allocated in a small storage memory, the large process is divided into routines, when the routines are called by OS, then that specific routine goes into OS. If the OS is full and another routine is called, swapping of previous routine occurs.



DYNAMIC LINKING

```
if (true) {
    func1(x);
} else {
    func2();
}
```

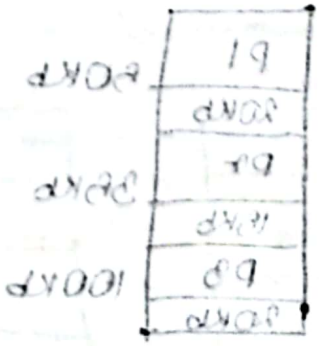


* Dynamically, ~~অথবা~~ func1 and func2 যখন এর তা লাগে তখন load হয় মানে func1 and func2 একসাথে load হয় না।

* Statically, func1 & func2 একসাথে load হয়।

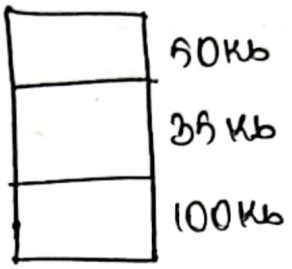
Contiguous Allocation: Process continuously RAM এ চলে যায়। sequentially.

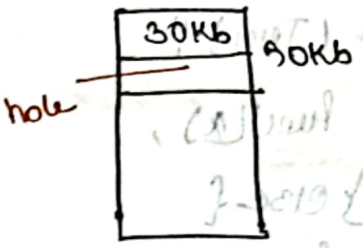
Disadv: Program বড় হলে, program contiguous allocation এ problem হয়ে যায়।



***MULTIPLE-PARTITION ALLOCATION

P1 → 30Kb
P2 → 20 Kb
P3 → 80Kb



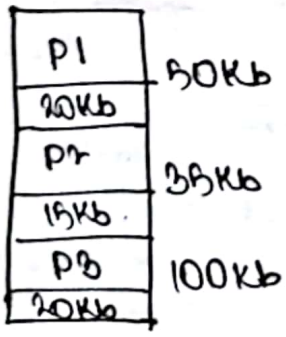


Fixed-sized partition:— No. of partition = No. of process
(static allocation)

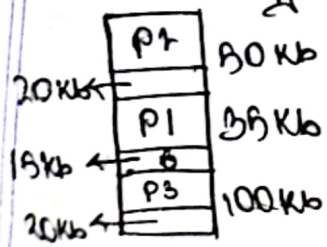
Variable-sized partition:— Each hole is treated as a
(Dynamic allocation) new segment which can be
used to store process.

Using fixed-sized partition :—

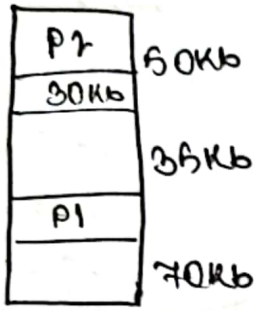
(1) First fit: প্রথমে যেখানে space পাবে সেখানে process store করবে।



(2) Best fit : minimum hole create করবে এমন space এ যেখানে সবচেয়ে ছোট use করবে



(3) Worst – fit : maximum hole create करेगा

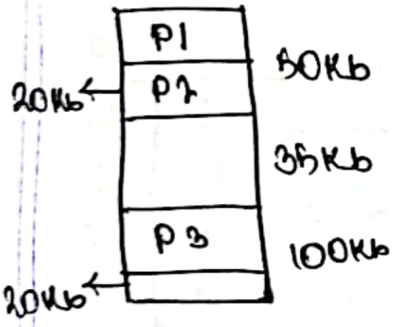


* memory में जितने भी hole exist करते sequentially process store करने possible नहीं

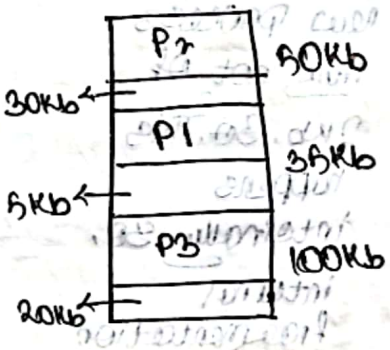
P3 cannot be allocated as there's no space remaining sequentially

Using variable partition:

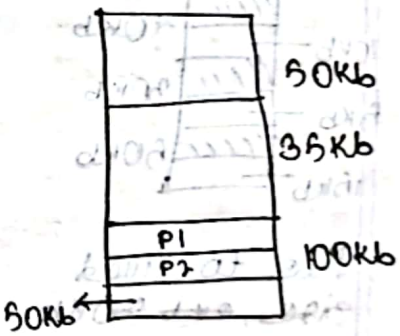
(1) First – fit :

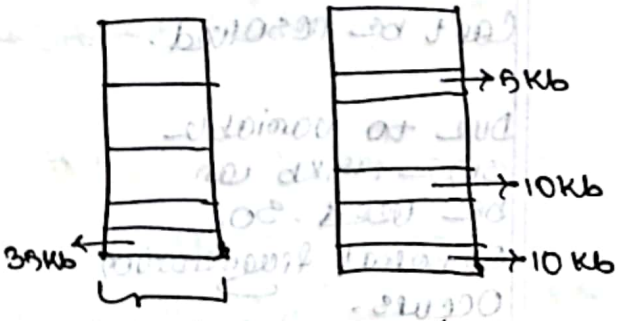


(2) Best – fit :



(3) Worst – fit





This is best applicable as if a new process comes, it can be loaded here.

LECTURE 16

SUNDAY DATE: 13/08/23

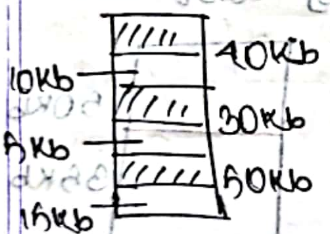
FRAGMENTATION

* Fragmentation - one kind of overhead

* 2 types of fragmentation
  - External
  - Internal

External

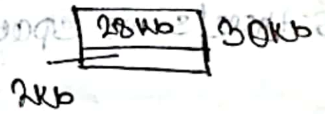
[Static allocation এ hole এ অনেক resource জায়গা থাকে না]



Due to fixed size, ~~ext~~ 30Kb is being lost. So, ~~the~~ So, external fragmentation occurs.
Can't be resolved.

Due to variable size, 15Kb can be used. So, external fragmentation occurs.
Can be resolved by placing all the remaining chunks together — compaction

Internal



2Kb is wasted. and ~~new~~ any new process may not of 2Kb. So, This happens internally. So, internal fragmentation occurs.

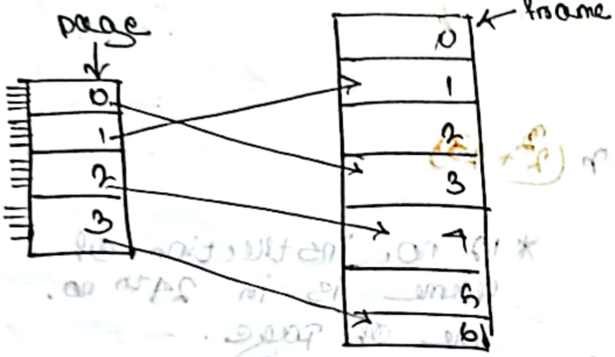
To reduce this, the remaining 2Kb is released.

PAGING

* Paging — contiguous

Characteristics:

(1) Non — contiguous



* sequentially না ও থাকতে পারে

* size of frame = size of page

P	F
0	3
1	1
2	4
3	6

Page Table

* Paging এর জন্য "page table" লাগে

* Page table এ address এর track রাখে

*

* Frame: logical memory কে divide করি into chunks

* Page: Physical memory কে divide করি into chunks.

Page Number		Page Offset	
1	0	1	1

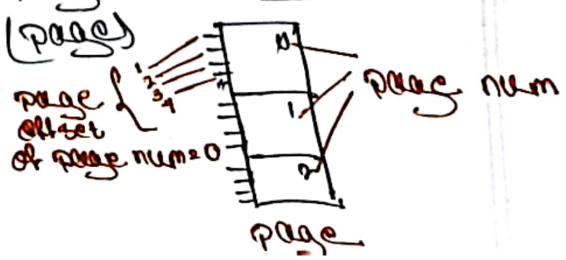
size of page = 4 = 2^2 → byte (2) → page offset

pa page offset

~~* Page Offset : Bit এর ঐ line~~ এর bit

* Page Offset : line no. in the page num.

* Page num: base address of each page in physical address (page)



0
1
2
3
4

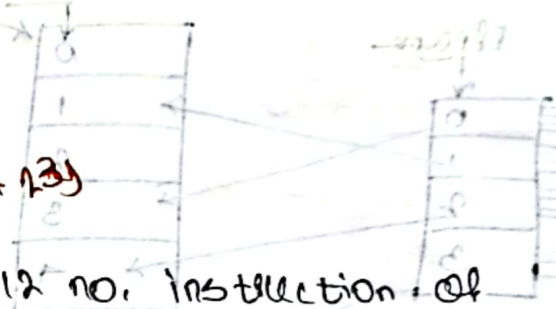
size = 4

n(frame) = 3

frame num = 3 (decimal)
= 011 (binary)
↳ 3 byte

frame num			page offset	
0	1	1	0	0

= 12 ($2^3 + 2^2$)



* 12 no. instruction of frame is in 24th no. line of page.

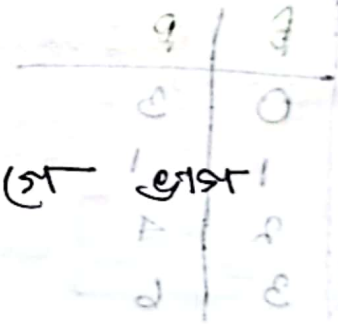
page num			page offset	
1	1	0	0	0

= 24 ($2^3 + 2^4$)

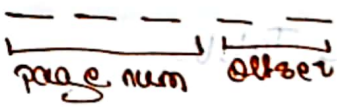
→ 2^5 → m=5 (5 byte total num)

32-byte memory } memory ⇒ $\frac{32}{4}$ = 8 ⇒ 8 pages
and 4-byte pages } ↳ 2^2 ⇒ n=2 (2 byte offset)

m=2, n=2

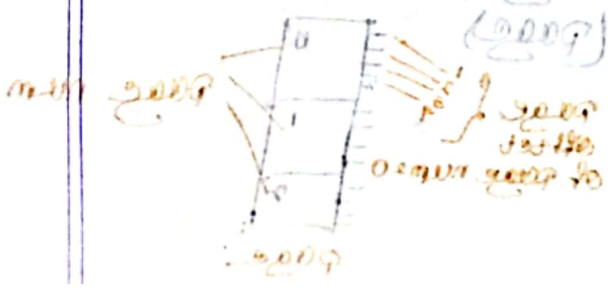


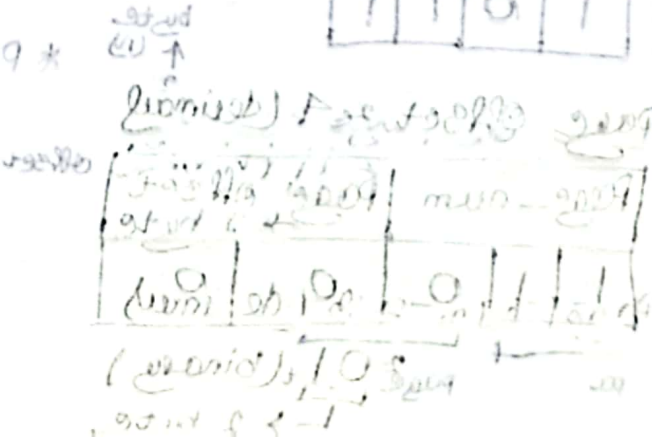
→ page size



2^m → no. of byte of entire ~~entire~~ ~~address~~ process

		0	1
1	1		





LECTURE 17

SUNDAY

DATE: 20/08/23

SIZE OF PAGE

P1

process size = 72,766

page size = 2048 byte

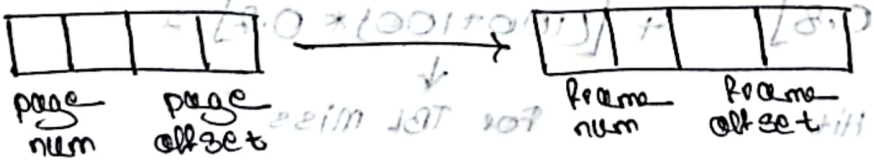
no. of page = $\frac{\text{process size}}{\text{page size}}$ = 35.53 ≈ 36

fraction ভাগফল internal fragmentation ভাগফল

35 page's completely filled = 35 * 2048 = 71,680

36 partially filled ভাগফল = 72,766 − 71,680 = 1086 (.53 bytes ভাগ করলে)

Internal fragmentation = 2048 − 1086 = 962



ASSOCIATIVE MEMORY

* CPU থেকে page table এ যেতে লাগবে, page table থেকে physical address এ যাবে। This takes time.

* TLB — কোনো কিছু খোঁজে পেলে cache এ রেখে দিব, Next time, TLB তে খোঁজে পেলে page table এ না গিয়ে, TLB hit হলে physical address এ চলে যাবে।
(↓ একটা cache memory) এর কাজে লাগে।

EFFECTIVE ACCESS TIME

Hit ratio = α → TLB Hit (কত সময় TLB পাওয়া যায়)

ε → Access করার time (TLB)

b → memory Access [logical address থেকে ~~TBL miss~~ page table (TBL), page table to physical address]

$\alpha = 80\%$, $\varepsilon = 20ns$, $b = 100ns$

80% → (100+20)
20% → (100+100)

EAT = [(100+20)*0.8] + [(100+100)*0.2] =

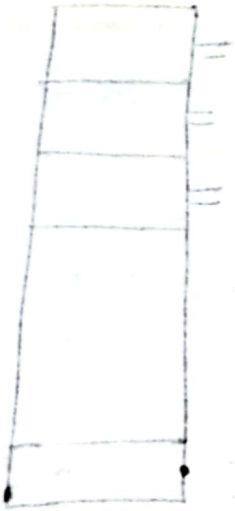
↓ For TBL Hit ↓ For TBL Miss

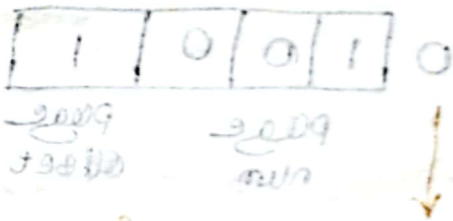
$$EAT = \alpha * \varepsilon + b + (1-\alpha) * 2b$$

SHARED PAGES EXAMPLE

* If multiple process has same data, then we can keep the data in a single place in the frame by updating the page number of the ~~for~~ the data in the page table.

* This reduces the space occupied by duplicates.





PRACTICE PROBLEM

(2) P1 — 10 byte → 5 ——— 5 এর জন্য 3 bytes assign করা লাগবে, so page num = 3

P2 — 6b → 3

P3 - 8b → 4

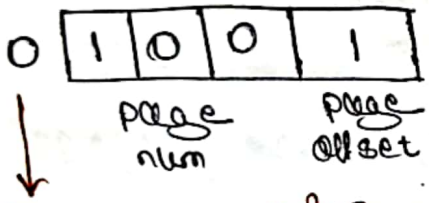
Page size — 2b ——— 1 bit assign করা লাগবে for page offset.

$2^n = 2$

$n = 1$

(একটা page এ 2 bit করে রাখা যায়)

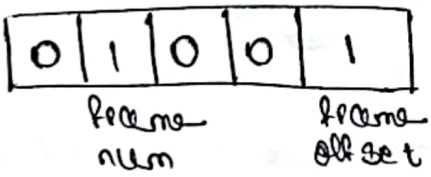
(a) P1

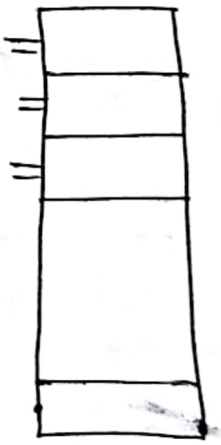


0 byte এ যাইছে

byte এ যাইছে।

Leading 0 থাকলে (no problem)

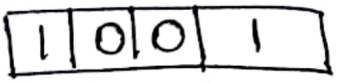




frame 32b

$\frac{32}{2} = 16$ frames

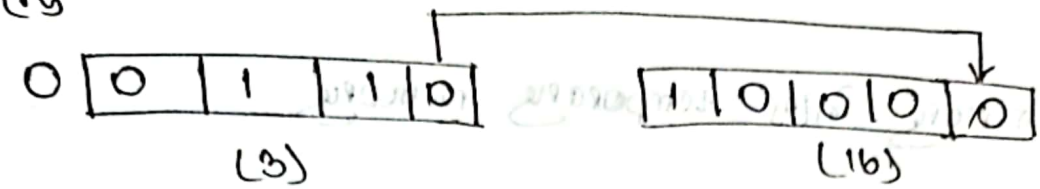
(b) 1



↳ leading 1, so invalid page num

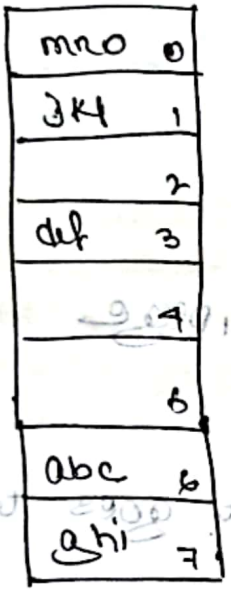
00110-P3

(4)

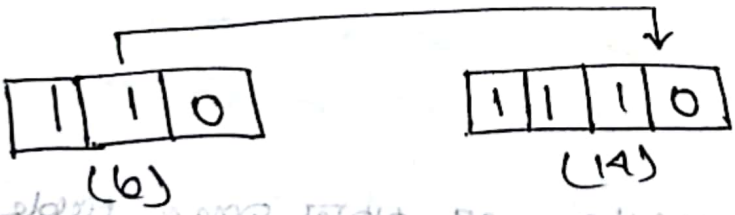


(3) page size = 4

Physical memory = 32



Total num of page = $\frac{32}{4} = 8$



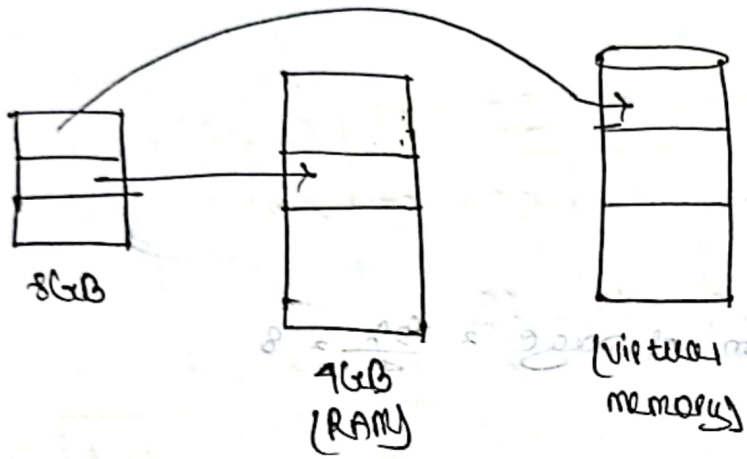
(5) $0.7 * 0.83 + 0.3 * 160 = 106.1$ ns

LECTURE 18

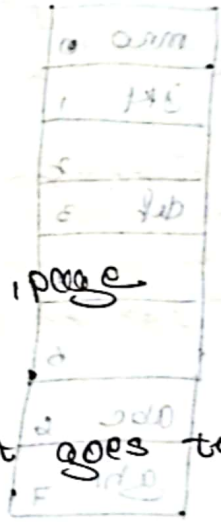
TUESDAY DATE: 22/08/23

VIRTUAL MEMORY

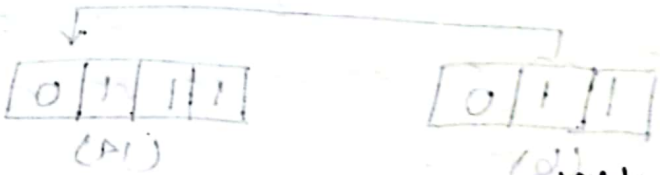
* Hardware থেকে memory নিয়ে temporary memory



* Page fault: Next Page number RAM এ না থাকলে, page fault হয়



* Process এর chunk RAM এ থাকে and rest goes to virtual memory



* Swap in & swap out operation এর সময় page table update হয়
* Valid হলে process RAM এ থাকে
* Invalid হলে abort করে দেয়,

Not in RAM হলে, virtual memory থেকে RAM এ নিয়ে আসে. if there's a free frame. Free frame না থাকলে, swap in & swap out হয়, page table update হয়।

FIFO

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2

P2	P2	P2	P2	P3	P5	P5	P5	P3	P3	P3	P3
	P3	P3	P3	P3	P2	P2	P2	P2	P5		
			P1	P1	P1	P4	P4	P4			
*	*	hit	*	*	*	*	hit	*	hit	*	*

hit = 3 | fault = 9

ratio = $\frac{3}{12}$ | ratio = $\frac{9}{12}$

* যে process first এ frame এ enter করেছে, সে আগে swap out হবে.

LRU (Least Recently Used)

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2

P2	P2	P2	P2	P2	P2	P2	P2	P3	P3	P3	P3
	P3	P3	P3	P5	P5	P5	P5	P5	P5	P5	P5
			P1	P1	P1	P4	P4	P4	P2	P2	P2
*	*	hit	*	*	hit	*	hit	*	*	hit	hit

hit = 5 | fault = 7

ratio = $\frac{5}{12}$ | ratio = $\frac{7}{12}$

* যে process আগের আগে frame এ enter করেছে, সে আগে swap out হবে.

Optimal

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2

P2	P2	P2	P2	P2	P2	P4	P4	P4	P2	P2	P2
	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3
			P1	P5	P5	P5	P5	P5	P5	P5	P5
*	*	hit	*	*	hit	*	hit	hit	*	hit	hit

hit = 6

hit ratio = $\frac{6}{12}$

fault ratio = $\frac{6}{12}$

* যে process এর future এ আসার chance সবচেয়ে কম বা অনেক পরে আসবে, তাকে swap করবে।

* The algorithm with the highest hit ratio & lowest fault ratio is the most effective.

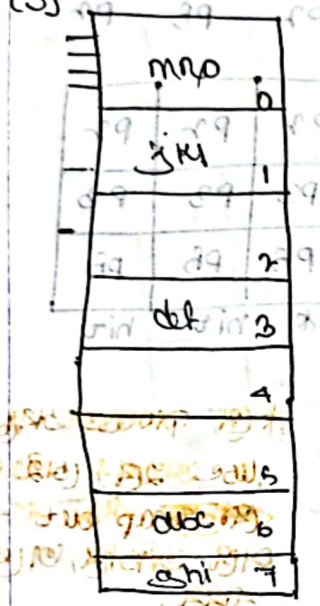
LECTURE 19

SUNDAY

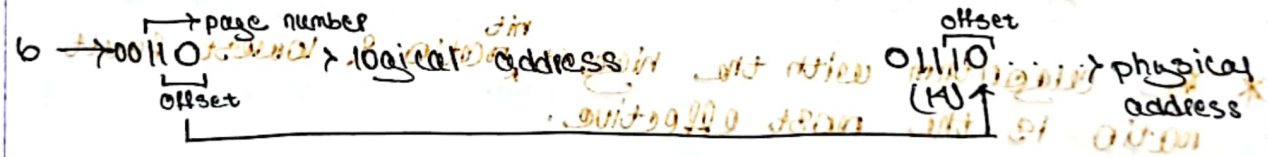
DATE : 27/08/23

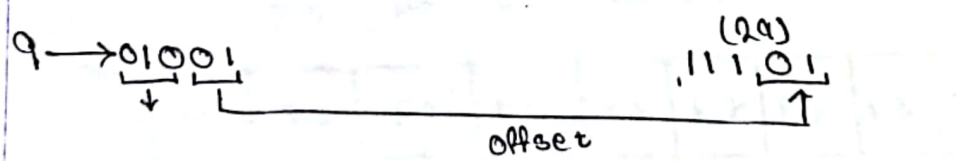
PRACTICE PROBLEM

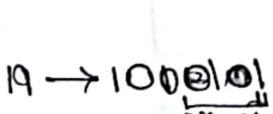
(3)



~~page no~~ frame = 8. To represent 8, we need 3 bits.
So, frame no. = 3







FINAL

(I) Process Synchronization
Peterson's Algorithm
Theory

(II) Scenario based Question

(III) Banker's Algorithm

(IV) Resource Allocation Graph

(V) Memory Access

(VI) Contagious Allocation
[Best fit, Worst fit, First fit]

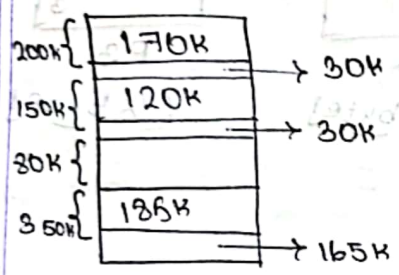
(VII) Memory - Theory

(VIII) Virtual memory

PRACTICE PROBLEMS ON MAIN MEMORY & VIRTUAL MEMORY

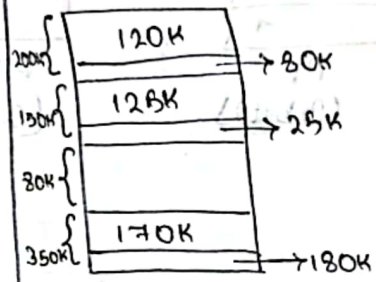
(1) Using fixed sized partition,

First fit



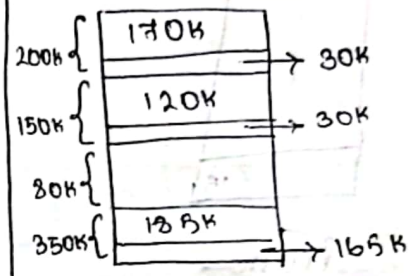
mem. = 305 Kb

Worst fit



mem. = 285 Kb

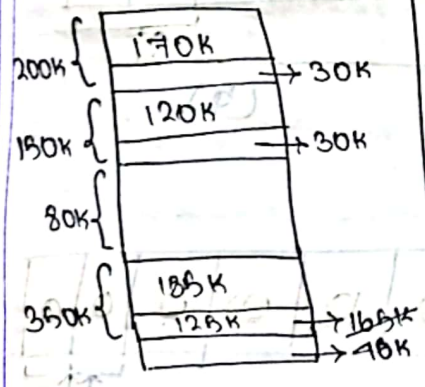
Best fit



mem = 305 Kb

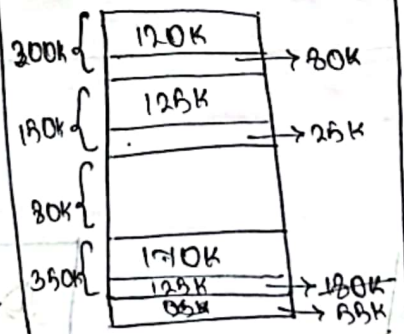
Using Variable Sized Partition

First fit



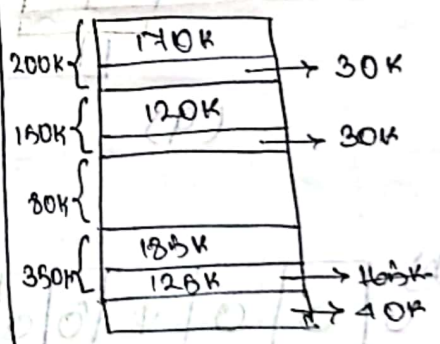
mem = 130Kb

Worst fit



mem = 160Kb

Best fit

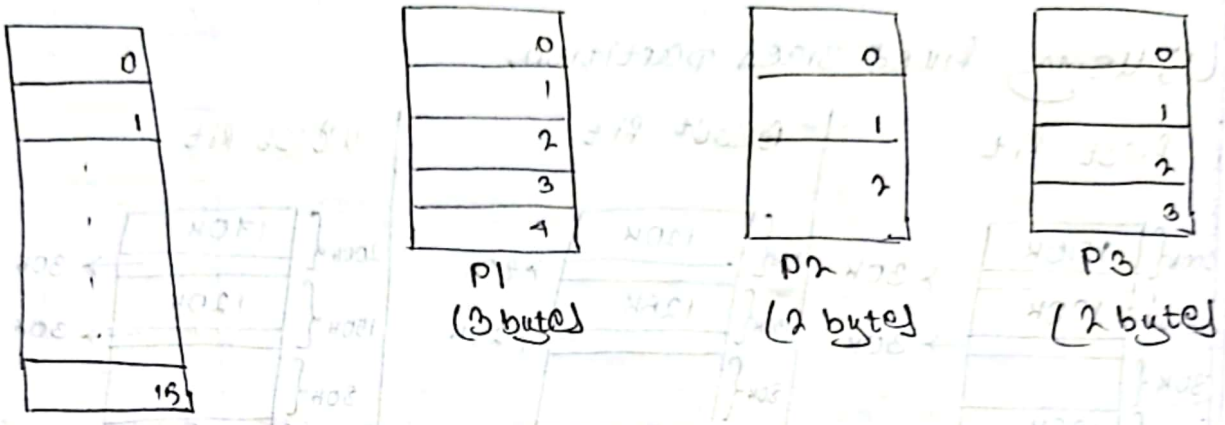


mem = 130Kb

First fit & Best fit.

10	1010	14	1110
11	1011	15	1111
12	1100		
13	1101		

(2)



frame

offset → $2^N = 2$ ∴ N = 1

logical address

physical address

(a) 0 | 1 | 0 | 0 | 1

(9)

0 | 1 | 0 | 0 | 1

(9)

(b) 0 | 0 | 1 | 0 | 0

(4)

0 | 0 | 0 | 1 | 0

(2)

(c) 0 | 0 | 1 | 1 | 1

(7)

0 | 1 | 0 | 1 | 1

(11)

logical address

physical address

(d) 0 0 1 → (1)

1 1 1 1 1 → (31)

(e) 0 0 1 0 0 → (4)

1 0 0 1 0 → (18)

(f) 1 0 1 0 1

page invalid due to leading 1

(g) 0 0 0 1 0 → (2)

0 1 1 0 0 → (12)

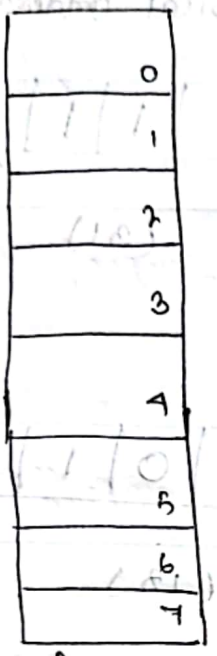
(h) 1 1 0 1 1

page invalid due to leading 1s

(i) 0 0 1 1 0 → (6)

1 0 0 0 0 → (16)

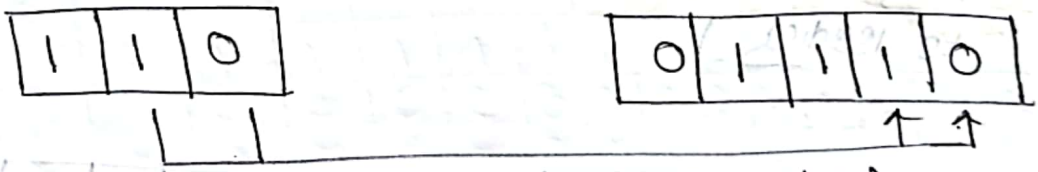
(3)



offset → $4 = 2^N$ ∴ N = 2

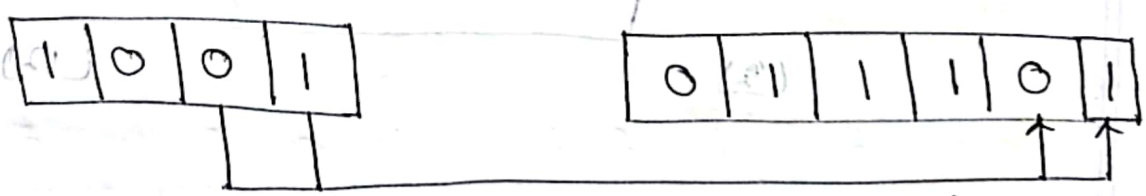
logical address | physical address

(a) (a)



(14)

(b) (9)



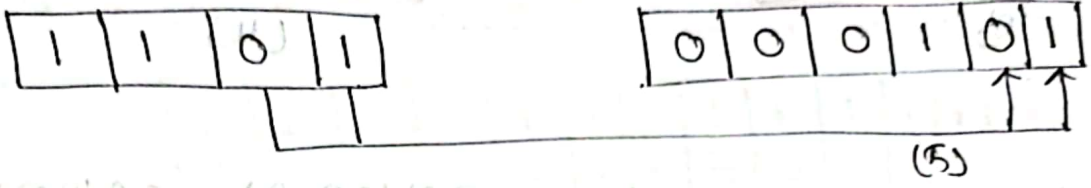
(29)

(c) (19)

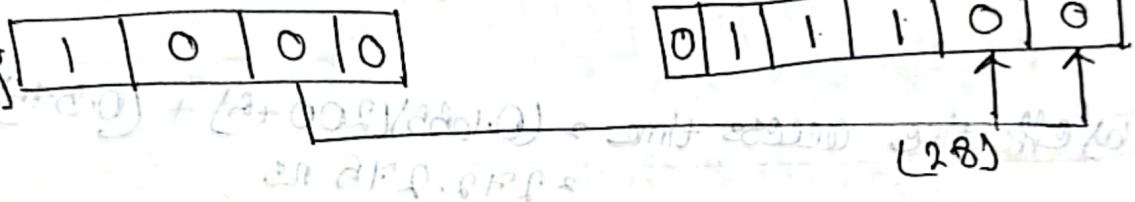
(3)

(d) logical address | physical address

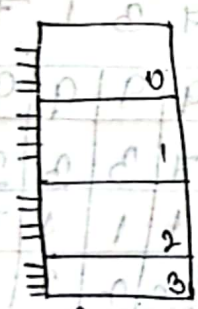
(d) (13)



(e) (8)

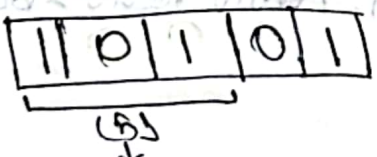


(4) $4 = 2^N \Rightarrow 2 \rightarrow$ offset

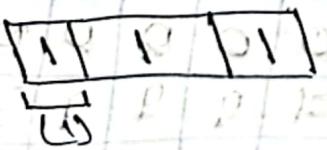


logical address | physical address

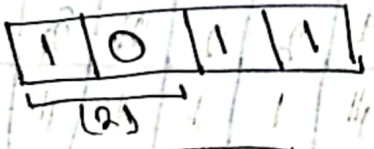
21



7

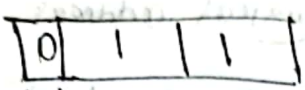


11



6

3



(10)

(11)

(5) effective access time = 0.7(80+3) + 0.3(80+80)
= 106.1 ns

(6) effective access time = (0.685)(200+3) + (0.345)(200+200)
= 272.278 ns

(7) FIFO

1 3 4 3 0 4 7 1 2 9 1 2 7 3 1 7

1	1	1	1	0	0	0	0	0	9	9	9	9	9	9	
	3	3	3	3	3	7	7	7	7	7	7	7	3	3	3
		4	4	4	4	4	1	1	1	1	1	1	1	1	7
			3	3	3	3	3	2	2	2	2	2	2	2	2
x	x	x	x	x	✓	x	x	x	x	✓	✓	✓	x	✓	x

Hit ratio = $\frac{5}{16}$ * 100% = 31.25%, fault ratio = 68.75%.

LRU

1 3 4 3 0 4 7 1 2 9 1 2 7 3 1 7

1	1	1	1	0	0	0	0	2	2	2	2	2	2	2	2
	3	3	3	3	3	7	7	7	7	7	7	7	7	7	7
		4	4	4	4	4	4	4	9	9	9	9	3	3	3
			3	3	3	3	1	1	1	1	1	1	1	1	1
x	x	x	x	x	✓	x	x	x	x	✓	✓	✓	x	✓	✓

Hit ratio = $\frac{6}{16}$ * 100%, = 37.5%, fault ratio = 62.5%.

Optimal

1 5 4 3 0 4 7 1 2 9 1 2 7 3 1 7

1	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	5	5	5	0	0	7	7	7	7	7	7	7	7	7	7
		4	4	4	4	4	4	2	2	2	2	2	3	3	3
			3	3	3	3	3	3	9	9	9	9	9	9	9
x	x	x	x	x	✓	x	✓	x	x	✓	✓	✓	x	✓	✓

hit ratio = $\frac{7}{16}$ * 100.1. = 43.75%. , fault ratio = 56.25%.

* Optimal performs better because it provides the maximum hit.