Method to Recover from Deadlock

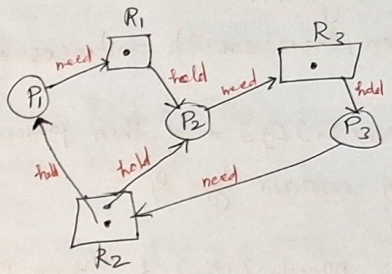
1. Process Termination → Abort all processes / Abort specific processes
2. Resource Preemption → Break the circular dependency / Rollback
3. Detection & Recovery
4. Deadlock avoidance.
5. Deadlock prevention

04/12/2024

Comprehensive 2024 (April QP with solutions)

① **Deadlock**: In a multiprogramming env. several processes may compete for a finite no. of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is k/a deadlock.

Ex:-



① $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Deadlock.

② $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Necessary & Sufficient condition for a deadlock / characteristics

*(a) **Mutual Exclusion**: At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.

*(b) **Hold & wait**: A process must be holding at least one resource & waiting to additional resource that is currently held by other processes.

*(c) **No preemption**: Resources allocated to a process can't be forcibly taken out from it unless it releases that resources after completing the task.

*(d) **Circular wait**: A set $\{P_0, P_1 \ldots P_n\}$ of waiting state/process must exists such that P_0 is waiting for a resources that is held by P_1, P_1 is waiting for the resource that is held by P_2 ..... P_{n-1} is waiting for resource that is held by P_n & P_n is waiting

for resources that is held by P_0.

Ex:- "Dining Philosophers" in deadlock.

② Banker's Algorithm: Deadlock avoidance algorithm ($m \times n^2$ time complexity)

Let $n = i$ → No. of processes, $m = j$ → No. of resources types.

* Available: Vector of length m.
If Available[j] = k, then, there are k instances of resource type R_j available.

* Max: n x m matrix; Max[i][j] = k, then process P_i may request at most k instances of resource type R_j.

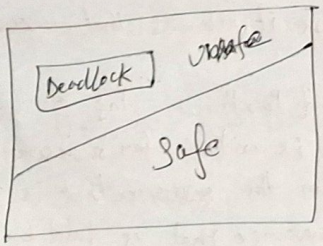
* Allocation: n x m matrix; Allocation[i][j] = k then P_i is currently allocated k instances of resources type R_j.

* Need: n x m matrix; Need[i][j] = k then P_i may need k more instances of R_j to complete its task.

$$Need[i][j] = Max[i][j] - Allocation[i][j]$$

* Finish: Boolean values, either TRUE or FALSE.
If finish[i] = TRUE ∀ i return safe state else unsafe.

Safety Algorithm:



Step (1): Let Available & Finish be vectors of length m & n respectively. Initialize:
Finish[i] = False ∀ i = 0, 1, 2 - - - n-1.

Step (2): Find an i such that both:
ⓐ Finish[i] = False
ⓑ Need ≤ Available
} If no such i exists, go to step (4).

Step (3): Available = Available + Allocation$_i$
Finish[i] = TRUE
Goto step (2).

Step (4): If Finish[i] == TRUE ∀ i
then system is in a safe state.

Resource-Request Algorithm

* Request$_i$ = Request vector for process P_i.
If Request[j] = k then process P_i wants k instances of resource type R_j.

Safe ① If Request$_i$ ≤ Need$_i$ } Go to step (2); otherwise raise error condition because its more than Max[i][j].

Unsafe ② If Request$_i$ ≤ Available } Go to step (3); otherwise P_i must wait since resources are not available

③ Pretend to allocate requested resources to P_i by
Available = Available - Request$_i$;
Allocation$_i$ = Allocation$_i$ + Request$_i$;
Need$_i$ → Need$_i$ - Request$_i$;

* If safe: Resources are allocated to P_i
* If unsafe: P_i must wait & the old resources allocation state is restored.

Ex:- Consider a system with five processes P_0 through P_4 & three resource type A, B, C. Resource type A has 10 instances, Resource B has 5 instances & resource type C has 7 instances. Suppose that the atime T_0, the following snapshot of system has been taken:

Process	Allocation			Max			Available			Need = Max − Allocation		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2	5	3	2	1	2	2
P_2	3	0	2	9	0	2	7	4	3	6	0	0
P_3	2	1	1	2	2	2	7	4	5	0	1	1
P_4	0	0	2	4	3	3	10	4	7	4	3	1

check safe/unsafe state

Available = 3 3 2

Need ≤ Available

P_1 Available$_1$ = Available + Allocation.

= 3 3 2 + 2 0 0

= 5 3 2

P_3 Available$_3$ = 5 3 2 + 2 1 1 = 7 4 3

P_4 Available$_4$ = 7 4 3 + 0 0 2 = 7 4 5

P_2 Available$_2$ = 7 4 5 + 3 0 2 = 10 4 7

P_0 Available$_0$ = 10 4 7 + 0 1 0 = 10 5 7

Hence, safe state < P_1, P_3, P_4, P_2, P_0 >.

Now Step(1): Request by P_1 = (1, 0, 2)

① → check feasibility: Request$_i$ ≤ Available.

i.e. (1, 0, 2) ≤ (3, 3, 2) True.

② → Update state.

Available = Available − Request$_i$ ⇒ (3, 3, 2) − (1, 0, 2) = (2, 3, 0)

Allocation$_i$ = Allocation$_i$ + Request$_i$ ⇒ (2, 0, 0) + (1, 0, 2) = (3, 0, 2)

Need$_i$ = Need$_i$ − Request$_i$ ⇒ (1, 2, 2) − (1, 0, 2) = (0, 2, 0)

③ Update Matrices

Process	Allocation (A, B, C)	Need (A, B, C)	Available (A, B, C)
P_0	(0, 1, 0)	(7, 4, 3)	(2, 3, 0).
P_1	(2, 0, 0)	(0, 2, 0)	
P_2	(3, 0, 2)	(6, 0, 0)	
P_3	(2, 1, 1)	(0, 1, 1)	
P_4	(0, 0, 2)	(4, 3, 1)	

③ check Safety Algorithm to verify if the system is in a safe state.

(a) Start with Available = (2, 3, 0)

(b) Find process Pi such that Needi ≤ Available

→ P_1 : (0, 2, 0) ≤ (2, 3, 0). safe, release resources:

Available = (3, 0, 2) + (2, 3, 0) = (5, 3, 2)

→ P_3 : (0, 1, 1) ≤ (5, 3, 2). safe, release resources:

Available = (2, 1, 1) + (5, 3, 2) = (7, 4, 3)

→ P_4 : (4, 3, 1) ≤ (7, 4, 3). safe, release resources:

Available = (0, 0, 2) + (7, 4, 3) = (7, 4, 5)

→ P_2 : (6, 0, 0) ≤ (7, 4, 5). safe, release resources

Available = (3, 0, 2) + (7, 4, 5) = (10, 4, 7)

→ P_0 : (7, 4, 3) ≤ (10, 4, 7). safe.

Hence Safe sequence < P_1, P_3, P_4, P_2, P_0 > satisfy the Safety condition. The System is safe.

Step(2): Request by P_4 = (3, 3, 0)

① check feasibility: Request4 ≤ Available

(3, 3, 0) ≤ (2, 3, 0) false.

Request is not granted.

Step(3): Request by P_0 = (0, 2, 0)

① check feasibility: Request ≤ Available

(0, 2, 0) ≤ (2, 3, 0) True

② Update state (Assume the request is granted):

Available = Available$_i$ − Request$_i$ ⇒ (2, 3, 0) − (0, 2, 0) = (2, 1, 0)

Allocation$_i$ = Allocation$_i$ + Request$_i$ ⇒ (0, 1, 0) + (0, 2, 0) = (0, 3, 0)

Needi = Needi − Requesti ⇒ (7, 4, 3) − (0, 2, 0) = (7, 2, 3)

Update matrices

Process	Allocation (A, B, C)	Need (A, B, C)	Available (A, B, C)
P_0	(0, 3, 0)	(7, 2, 3)	(2, 1, 0)
P_1	(3, 0, 2)	(0, 2, 0)	
P_2	(3, 0, 2)	(6, 0, 0)	
P_3	(2, 1, 1)	(0, 1, 1)	
P_4	(0, 0, 2)	(4, 3, 1)	

③ check safety Algorithm:

→ Start with Available = (2, 1, 0)

→ Find Pi such that Needi ≤ Available ∀. No process satisfies.

Hence, The system is unsafe after granting the request.

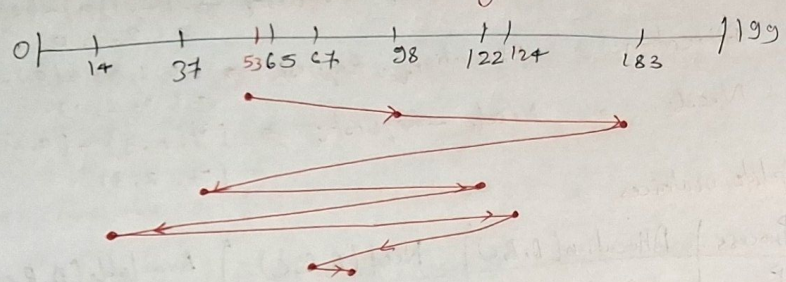
Hence, the request by P_0 cannot be granted.

③ Disck scheduling

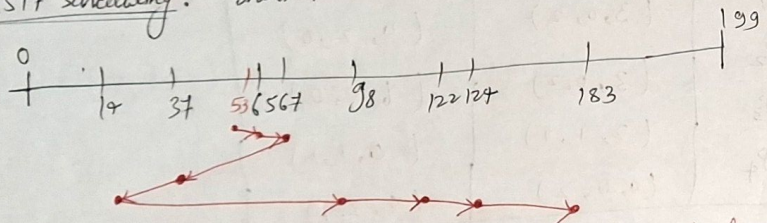
* Default head moves to the left
* If previous is provided then moves towards right. (or based on previous cylinder).

FCFS Scheduling: FIFO manner worked in sequential order.

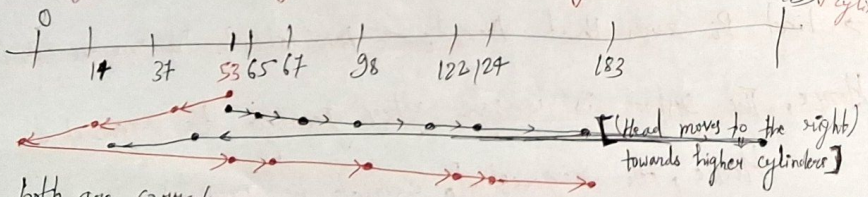
Ex:- Request queue (0-199): 98, 183, 37, 122, 14, 124, 65, 67
Consider now the head pointer is in cylinder 53.



SSTF scheduling: Shortest-seek-time-first

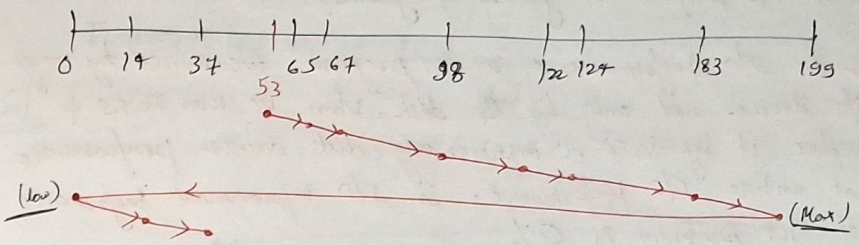


SCAN scheduling: (Elevator Algo.) The disk arm starts at one end of the disk, & moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. Head movement decided by the head cylinder.

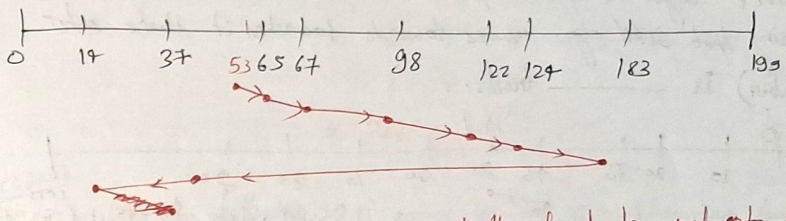


Or both are correct.

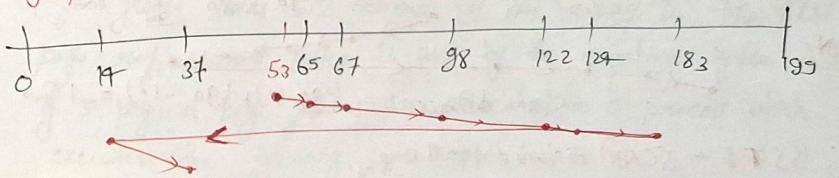
C-SCAN scheduling: The disk head moves towards higher-numbered cylinders, servicing requests. When it reaches the highest cylinder, it jumps directly to the lowest cylinder (without servicing any requests) and continues moving to the right.



Look: Similar to SCAN, but the head does not go to the extreme ends (highest or lowest cylinder). It only goes as far as the farthest request in each direction before reversing.



C-LOOK: Similar to C-SCAN, but the head does not go to the extreme ends. It serves all requests in one direction, then jumps to the other end. (where the requests start) & continues.



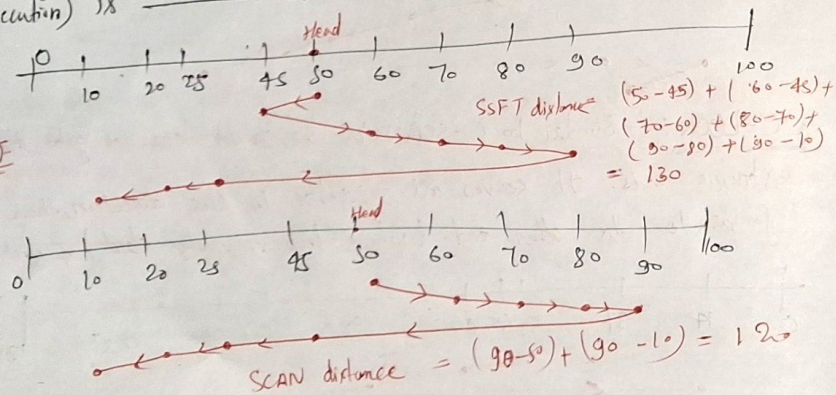
GATE 2004

Q. Consider an operating system capable of loading & executing a single sequential user process at a time. The disk head scheduling algorithm used is FCFS. If FCFS is replaced by SSTF, claimed by the vendor to given 50% better benchmark results, what is the expected improvement in the I/O performance of user programs?

Ans The I/O performance of a user program determined by many I/O devices not only by the disk. When we uses FCFS & replace it by SSTF it improves only disk driver performance not entire I/O performance. So I/O improvement performance of user program is 0%.

GATE 2015-Set-1 Q. Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given 45, 20, 90, 10, 50, 60, 80, 25, 70. Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the SSTF algo. is used compared to the SCAN (Elevator) algo. (assuming that SCAN algo. moves towards 100 when it starts execution) is _____ tracks.

Sol:



∴ (SSTF − SCAN) distance = 10 Ans.

GATE 2016-Set-1

Q. C-Look

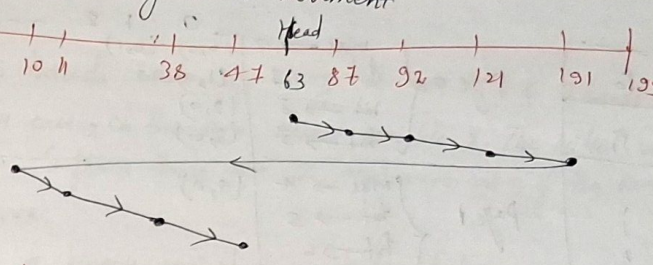
Given: The cylinders are numbered from 0 to 199

disk queue: 47, 38, 121, 191, 87, 11, 92, 10

The head is initially at cylinder no. 63, moving towards larger cylinder no. on its servicing pass.

Find ⇒ Total No. of head movement

Sol:



Total head movement = (87-63) + (92-87) + (121-92) + (191-121) + (191-10) + (11-10) + (38-11) + (47-38)

= 24 + 5 + 29 + 70 + 181 + 1 + 27 + 9

= 346 Ans.

GATE: 2023

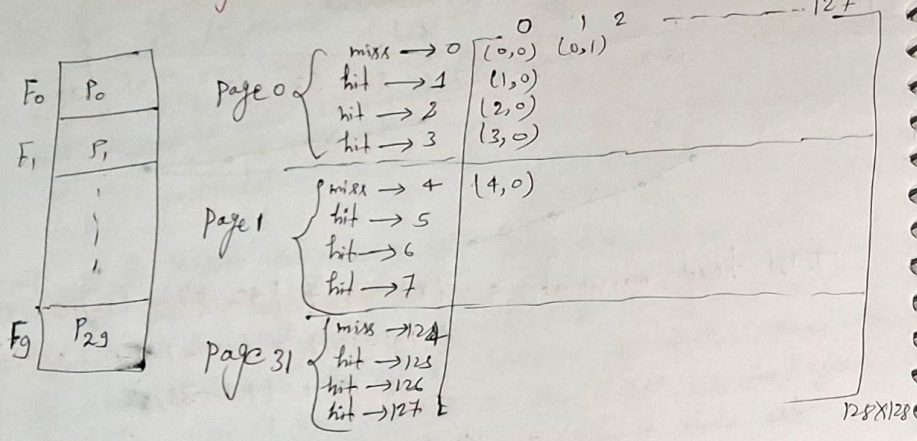
Q. Consider the following 2D array in the C programming language, which is stored in row-major order: int D[128][128];

Demand paging is used for allocating memory & each physical page frame holds 512 elements of the array D. The LRU page replacement policy is used by the operating system. A total of 30 physical page frames are allocated to a process which executes the following code snippet:

```
for (int i = 0; i < 128; i++)
    for (int j = 0; j < 128; j++)
        D[j][i] *= 10;
```

} The no. of page faults generated during the execution of this code snippet is ——

Sol^n: Given array dimension is D[128][128] which is stored in row major order. It's means that 128 rows & 128 columns



Given that, each physical frame holds 512 elements of the array D. 30 page frames are allocated to a process. Array access is in column major order [D[j][i] *=10].

At each physical frame holds (512) elements, the Ist 4 rows of the array belongs one page, we can see that, one page fault 4 elements. $\frac{512}{128} = 4$ elements

$\frac{128}{30} = 32$

Arrays contains 32 pages, but memory contains 30 page frames.

$$\therefore \text{No. of page faults} = \frac{128 \times 128}{4} = \frac{2^{14}}{2^2} = 4096$$

Ans.

GATE 2022

Consider the 2 func^n incr & decr shown below:

```
incr() {
    wait(s);
    X = X + 1;
    signal(s);
}

decr() {
    wait(s);
    X = X - 1;
    signal(s);
}
```

There are 5 threads each invoking incr once & 3 threads each invoking decr once, on the same shared variable X. The initial value of X is 10. Suppose there are 2 implementations of the semaphore s, as follows:

I_1: s is a binary semaphore initialized to 1.

I_2: s is a counting semaphore initialized to 2.

Let V_1 & V_2 be the values of X at the end of execution of all the threads with implementations I_1 & I_2 respectively. Which one of the following choices corresponds to the minimum possible values of V_1 & V_2 respectively?

(a) 15, 7 (b) 7, 7 (c) 12, 7 (d) 12, 8.

Sol^n: I_1: s is a binary semaphore initialized to 1. With binary semaphores there can't be 2 threads that can simultaneously access the critical section. Thus, each thread with can incr() or decr() one by one. Hence, only possible value of X at the end is 12.

I_2: With counting semaphore of value 2. Two processes at a time can access critical section.

X = X + 1
load X
Add #1, X
Store X

X = X - 1
load X
Sub #1, X
Store X

In order to minimume X value, we need to maximize the impact of decr() & minimize the impact of incr().

decr() X = X - 1
decr() X = X - 1
decr()
Load X
Hold X=7 (CS) → Sub #1, X
store X
Run incr() 5 times
→ Run last statement.

X = 7 Final.

OR

I1: Initially, X = 10, S = 1 (mutual exclusion) & it is binary semaphore. So after executing 5 incr(), X = 10 + 5 = 15 & after executing 3 decr(), X = 15 - 3 = 12

I2: Initially, X = 10 & S = 2 (counting semaphore), i.e. 2 processes can enter the CS at the same time.

15 & 7

i.e. 12 & 7 correct answer.

09/12/2024

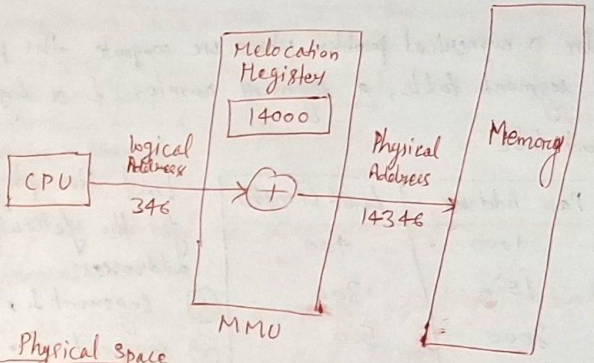
Comprehensive 2023 Qs with sol^n

① Necessary condition of Deadlock
- Mutual exclusion
- No preemption
- Hold & wait
- Circular wait

Method to recovery from Deadlock
- Deadlock avoidance
- Detection & Recovery
  - Rollback
  - Process Terminate?

② How logical address to physical address mapped. Explain in details.

Ans.



Logical vs. Physical space

→ An address generated by the CPU is commonly referred to as a logical address (LA) or a virtual address (VA) whereas an address seen by the main memory unit (MMU) is commonly referred as physical address. (PA)

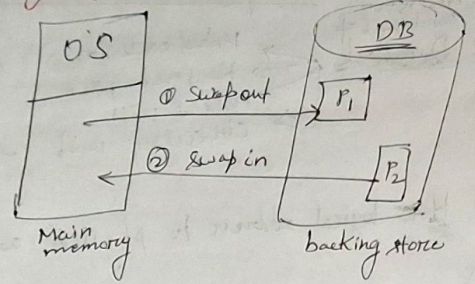
→ The set of all logical address generated by a program is a LA space & " " " " physical " " " " " " PA space.

→ Logical & physical addresses are the same in compile-time & load-time address binding scheme; logical (Virtual) & physical (actual) addresses differ in execution-time address binding scheme.

→ The Memory Management Unit is a H/w device that maps virtual to physical address. In MMU scheme, the value in the relocation register is added to every address generated by a user process

at time it is sent to memory as follows:

* ① Dynamic Loading
* ② Dynamic Linking
* ③ Overlays
* ④ Swapping → Roll out / Roll in



③ Let's solve a numerical problem where we compute the physical address given a segment table, a segment number & a logical address.

Given segment table

Seg. No.	Base Address	Limit (Length)
0	1000	400
1	1500	300
2	2000	500
3	3000	250

Find the physical address for the following logical addresses:

① Segment 1, offset 120
② Segment 2, offset 450
③ Segment 3, offset 300

Sol^n

① check if the offset is within the limit of the segment:
- If offset > limit; the logical address is invalid
- Otherwise; proceed to the next step.

② Compute the physical address:

// PA = Base Address of Segment + Offset. //

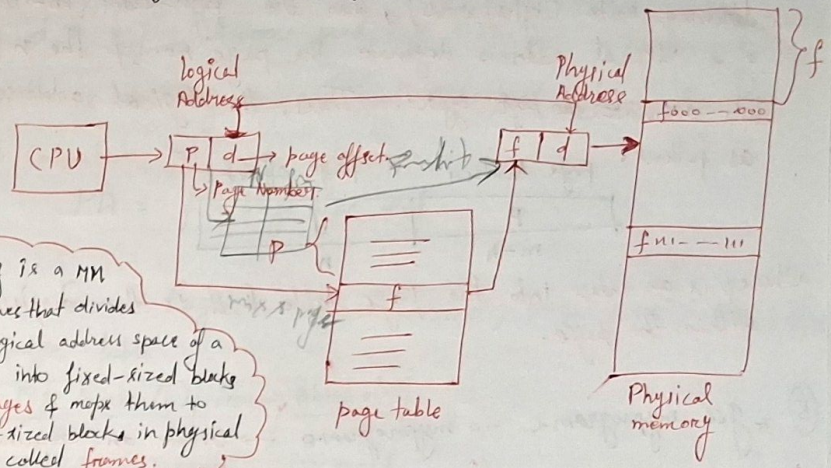
① Segment 1, offset 120 : ① $120 \leq 300$ (Valid)
② PA = 1500 + 120 = 1620

② Segment 2, offset 450 : ① $450 \leq 500$ (Valid)
② PA = 2000 + 450 = 2450

③ Segment 3, offset 300 : ① $300 \leq 250$ (Invalid)
Hence logical address is invalid.

④ How paging is mapped from logical address to physical address

→ Eliminate external fragmentation.



Paging is a MM technique that divides the logical address space of a process into fixed-sized blocks called pages & maps them to equally-sized blocks in physical memory called frames.

→ Main memory is divided into a number of equal-size blocks, are called frames.

→ Each process is divided into a no. of equal-size block of the same length as frames, are called Pages.

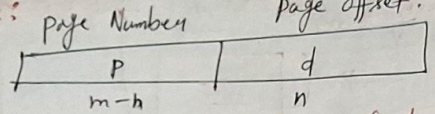
→ A process is loaded by loading all of its pages into available frames (may not be contiguous).

→ Process of translation from logical to physical address.

→ Every address generated by the CPU is divided into 2 parts → p, d
page Number (p) & page offset (d). The page number is used as an index into a page table.

→ The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical address that is sent to the memory unit.

↳ If the size of logical-address space is 2^m & a page size is 2^n addressing-units (bytes/words), then the high-order (m-n) bits of a logical address deginate the page no. & the n low-order bits deginate the page offset. Thus, the logical address is as follows:



where p is an index into the page table & d is the displacement within the page.

(5) * gcc myprogram.c -o myprogram.o → Create an executable named myprogram.o & file type executable.

* gcc -c myprogram.c -o myprogram.o → Create an object file named myprogram.o & file type → object file.
↳ c: flag to generate an object file.

~~* How paging Works~~:

① Logical address → Page No. (p), Page offset (d) } LA = p+d = Page No. + offset.

② Physical address → Frame No. (f), Offset (d) } PA = Frame No. * Page size + offset(d)

Ex:- Page size = 4KB = 2^{12} Bytes & Logical address = 32 bits

then p = 32 - 12 = 20 bits (page Number)

d = 12 (offset).

* Mapping Logical address to physical Address

↳ CPU generates Logical address

↳ Decompose logical address → p & d

p = [Logical Address / page size].

d = Logical Address mod Page size.

↳ Access the Page table: find corresponding frame f using p.

↳ Calculate physical address:

PA = f * Page size + d

Ex:- Given: Logical address = 1200, page size = 256 bytes & page table

Page Number (p)	Frame Number (F)
0	5
1	3
2	7

Sol^n ① Decompose logical address:

p = [1200 / 256] = 4

d = 1200 mod 256 = 176

② Access Page table: Page 4 is not mapped in the given table i.e. Invalid logical address.

If Logical Address = 400 then

① p = [400/256] = 1 & d = 400 mod 256 = 144

② page 1 maps to frame 3.

③ PA = 3 * 256 + 144 = 912.

Hence page table is key to translating logical address into PA.

⑦ Find the no. of child processes given the pseudo code with fork.

```
For (i=0; i<10; i++)
    if i%2 == 0:
        fork();
```

fork() called 5 times

Hence, No. of child process $= 2^n - 1 = 2^5 - 1$

$= (31)$ Ans.

⑧ Finding the best algorithm b/w FCFS-fit & Best-fit given the size of five memory blocks & size of 4 processes.

Given: Block size = {100, 500, 200, 300, 600}

Process size = {212, 417, 112, 426}

Soln: ① First-Fit Algo: Allocate the process to the first block that is large enough to accomodate it.

Process	Process size	Block Allocated	Remaining Block Size After Allocation
P_1	212	Block 2 (500)	500 − 212 = 288
P_2	417	Block 5 (600)	600 − 417 = 183
P_3	112	Block 3 (200)	200 − 112 = 88
P_4	426	No Block	Not Allocated.

P_1, P_2, P_3 are allocated but P_4 is not allocated.

Best ② Best-Fit Algo: Allocate the process to the smallest block that is large enough to accomodate it.

Process	Process size	Block Allocated	Remaining Block size after Allocation
P_1	212	Block 4 (300)	300 − 212 = 88
P_2	417	Block 5 (600)	600 − 417 = 183
P_3	112	Block 3 (200)	200 − 112 = 88
P_3	426	Block 2 (500)	500 − 426 = 74

③ Is there any fragmentation possible depending on the fixed size of block? When size of block is not fixed, is there possibility of no fragmentation? Give explanation with reason.

Ans: * Fragmentation with fixed Block sizes (static allocation) → Internal fragmentation

Ex: → Block size = 100kB → Process size = 70kB

→ Allocated block = 100kB → Wasted memory = 100 − 70 = 30kB. (Internal fragmentation)

Thus, fixed-size blocks always have the potential for "" because processes rarely fit perfectly into a block.

* Fragmentation with variable Block sizes → dynamic allocation (External fragmentation)

Ex:- → Initially, memory has 100kB, 300kB, & 200kB free blocks.

→ Process 1 (120kB) is allocated to the 300kB block, leaving 300 − 120 = 180kB as free block.

→ Now, there's 100kB, 180kB, & 200kB free.

→ Process 2 (250kB) cannot be allocated even though the total free memory is 100 + 180 + 200 = 480kB.

Fragmentation elimination techniques

① Paging → Eliminates external fragmentation (Internal fragmentation may be occur).

② Segmentation → Reduce internal fragmentation but can still lead to external

③ Compaction → Rearranges memory to group all free spaces together. Addresses external fragmentation but is time-consuming & impacted

④ Buddy System → Balanced internal & external fragmentation by combines fixed & variable-sized block allocation by splitting memory into power of two.

Galvin

(10) Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound task issue an I/O operation once for every millisecond of CPU computing & that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond & that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

(a) The time quantum is 1ms. (b) The time quantum is 10 ms.

Sol^{n}: What ever may be the Burst time of CPU-bound process (but it should be greater than TQ), it will run upto 1 TQ. after that already I/O bound process should come into ReadyQueue

→ Total time utilized = (1 * 1 * 1 ---- * 1 ten times of I/O bound proc) + 1 TQ for CPU-bound.

→ Extra time consumed = 1 context-switch for each process = n * 0.1 = 1.1 ms.

① If TQ = 1 ms ⇒ Efficiency = $\frac{10ms + 1ms}{10ms + 1ms + 1.1ms} = \frac{11}{12.1} = 90.9\%$

② If TQ = 10 ms ⇒ Efficiency = $\frac{10ms + 10ms}{10ms + 10ms + 1.1ms} = \frac{20}{21.1} = 94.78\%$

(6) Process synchronization & mutex related pseudo code.

→ A situation where several processes access & manipulate the same data concurrently & the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Producer - Consumer Problem

↳ Unbounded buffer places no practical limit on the size of the buffer.

↳ Bounded-buffer assumes that there is a fixed buffer size.

* Bounded-Buffer — Shared Memory Solution

→ Used to illustrate the power of synchronization primitives.

→ We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provided mutual exclusion for accesses to the buffer pool & is initialized to 1.

→ The empty & full semaphore count the no. of empty & full buffers respectively.

→ The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is

```
do{
produce an item in nextp
  ...
wait (empty);
wait (mutex);
  ...
add nextp to buffer
  ...
signal (mutex);
signal (full);
} while (1);
```

The code for the consumer process is

```
do {
wait (full);
wait (mutex);
  ...
remove an item from buffer to nextc
  ...
signal (mutex);
signal (empty);
  ...
consume the item in nextc
  ...
} while (1);
```

The Reader–Writers Problem

→ The semaphore mutex & wrt are initialized to 1;
→ readcount is initialized to 0.

The code for a writer process is

```
do {
 wait (wrt);
 ----
 Writing is performed
 ----
 signal (wrt);
} while (1);
```

The code for a reader process is

```
do {
wait (mutex);
readcount ++;
if (readcount == 1)
  wait (wrt)
signal (mutex);
---
reading is performed
---
wait (mutex);
readcount --;
if (readcount == 0)
signal (wrt);
signal (mutex);
} while (1);
```

Note: If a writer is in the critical section & n readers are waiting, then one reader is queued on wrt, & n-1 reader are ~~quet~~ queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a ~~signal~~ single waiting writer.

The Dining–Philosophers Problem

The structure of philosopher i & the semaphore chopstick[5]; where ~~all the elements that no two neighbours are eating simultaneously.~~

```
do {
wait (chopstick [i]);
wait (chopstick [(i+1) %.5]);
---
eat
---
signal (chopstick [i]);
signal (chopstick [(i+1) %.5]);
---
think
---
} while (1);
```

* where all the elements of chopstick are initialized to 1.
* Use an asymmetric sol^n that is an odd philosopher picks up first her left chopstick & then her right chopstick, whereas an even philosopher picks up her right chopstick & then her left chopstick.

Binary Semaphore:

→ binary semaphore S_1 & S_2;
int C;
Initially $S_1 = 1$, $S_2 = 0$ & the value of integer C is set to the initial value of the counting semaphore S.

* Wait operation on the counting semaphore S:

```
Wait (S1);
C--;
if (C<0){
signal (S1);
wait (S2);
}
signal (S1);
```

* Signal operation on the S:

```
wait (S1);
C++;
if (C <= 0) {
  signal (S2)
else
  signal (S1);
```

Peterson's solution:

is a s/w based sol^n to the critical sect^n problem.

→ Consider 2 processes P_0 & P_1. For convenience, when presenting P_i, we use P_j to denote the other process that is $j = 1-i$.

→ The process sharing 2 variables:

```
boolean flag[2];
int turn;
```

Initially flag[0] = flag[1] = false. & value of turn is immaterial (but either 0 or 1).

→ The structure of process P_i is show below:

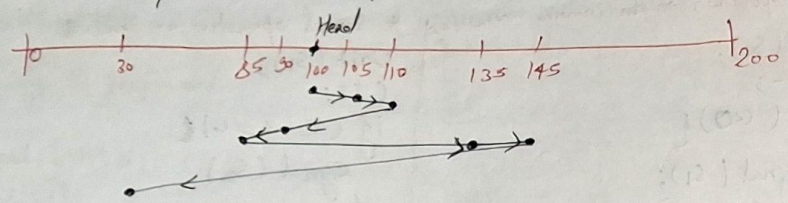
```
do {
    flag[i] = true
    turn = j
    while ( flag[j] && turn == j):
                Critical section;
    flag[i] = false
    ile (1); Remainder section
```

① Mutual exclusion is preserved.
② The progress requirement is satisfied
③ The bounded-waiting requirement is met.

05/12/2021

Comprehensive June 2021 QP with Solutions

(9) Given : disk = 201 cylinders, Head position = 100
Queue = 30, 85, 90, 100, 105, 110, 135, 145
SSTF algorithm & find sequence in which these requests are processed / serviced.



= (100 − 105) + (110 − 105) + (110 − 90) + (90 − 85) + (135 − 85) + (145 − 135)
= 05 + 5 + 20 + 5 + 50 + 115 + 10
= 210 Ans.

(8) Given Segment table

Seg. No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

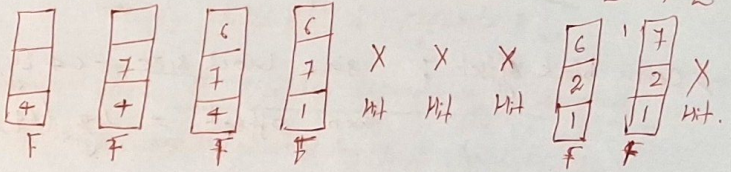
which of the following logical addresses will produce trap addressing error.
① 0, 430 ② 1, 11
③ 2, 100 ④ 3, 425
⑤ 4, 95

Sol": ① Valid offset check : @ offset < length (valid) else invalid.
@ offset > seg. length ⇒ trap addressing error.
② PA = Base Address + offset (If valid).

① 0, 430 : 430 < 700 (valid).
PA = 1219 + 430 = 1649.
② 1, 11 : 11 < 14 (valid)
PA = 11 + 2300 = 2311
③ 2, 100 : 100 ≮ 100 (invalid) ⇒ Trap addressing error.
④ 3, 425 : 425 < 580 (valid)
PA = 425 + 1327 = 1752
⑤ 4, 95 : 95 < 96 (valid)
PA = 1952 + 95 = 2047.

(7) page frame = 3 (Given), FIFO & find total No. of page faults & hit ratio & miss ratio.
Page reference string : 4, 7, 6, 1, 7, 6, 1, 2, 7, 2

Sol":



Total Page faults = 6 & Page Hits = 10 − 6 = 4.

* Hit Ratio = Page Hits / Total References = 4/10 = 0.4 or 40%

* Miss Ratio = Page Faults / Total References = 6/10 = 0.6 or 60%.

⑥ Given: Virtual address size = 46 bits
Physical address size = 32 bits
Size of page = ? in KB
Page table organization = 3-level paging.
PTE size = 32 bits (4 bytes)

* Cache properties: Cache size : 1 MB = 2^{20} bytes
Set associativity = 16-way
Block size = 64 bytes.

*cache indexing: Virtually indexed, physically tagged.

Sol^n
Now, No. of sets = $\frac{\text{cache size}}{\text{Associativity} * \text{Block size}} = \frac{2^{20}}{16*64}$
$= 2^{10} = 1024$ sets.

Hence, 10 bits are needed for cache index.

→ Cache block offset: since block size = 64 bytes
then offset = $\log_2(64) = 6$ bits.

→ Page offset = Block offset + cache index = 6+10 = 16 bits

→ Hence, Page size = 2^{16} bytes = 64 KB. Ans

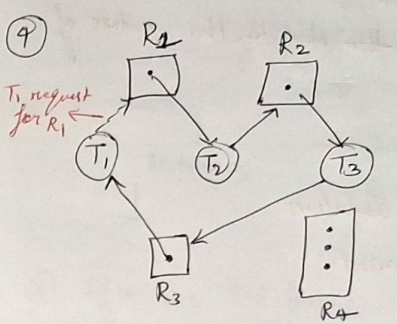
⑤ Given: VA = 32-bits, Page size = 4kB = 2^{12} bytes → 12 bits = offset
Tag = ? TLB = 128 page table entries & 4-way set associative.

No. of sets = $\frac{\text{Total TLB entries}}{\text{Associativity}} = \frac{128}{4}$ = 32 sets.

& cache index = $\log_2(32)$ = 5 bits.

Tag size = Total virtual address bits − (Page offset + cache index)
= 32 − (12+5) = 15 bits. Ans

④



To determine if a deadlock occurs, we analyze the scenario based on the 4 necessary & sufficient conditions for deadlock.

① Given: T_1 & T_2 → read-only
programs & resources
T_3 → write-only.
Resources → R_1, R_2, R_3 & R_4.

② Existing Allocation:
→ T_1 holds R_3 & Request R_1 (dotted arrow)
→ T_2 holds R_1 & requests R_2.
→ T_3 holds R_2 & Requests R_3.

③ If T_1 requests R_1: A circular dependency is created.
→ $T_1 \rightarrow R_1 \Rightarrow$ Requested by T_1 & held by T_2
→ $T_2 \rightarrow R_2 \Rightarrow$ Requested by T_2 & held by T_3
→ $T_3 \rightarrow R_3 \Rightarrow$ Requested by T_3 & held by T_1.

Deadlock Analysis

① Mutual Exclusion: Yes, resources R_1, R_2, R_3 & R_4 are assigned to one process at a time. This condition is satisfied.

② Hold & Wait: Yes, condition satisfied.

③ No preemption: condition satisfied.

④ Circular wait:
$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_2 \rightarrow T_3 \rightarrow R_3 \rightarrow T_1$
This condition is satisfied.

③ Threads share the heap but not the stack. What would happen if the threads used the same stack for execution?

Ans: → If threads used the same stack for execution, several critical problems would arise ~~thus~~ due to the nature of stack memory.

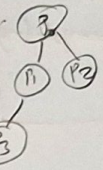
- → Data corruption
- → Loss of thread Isolation
- → Incorrect func^n behaviour
- → Race conditions.
- → Stack overflow

→ Why threads have separate stacks

- → Each thread is given its own stack so it can independently manage its local execution context (func^n calls, local variables, return addresses).
- → Threads share the heap for global or dynamically allocated data, which allows them to communicate & share state.

→ If threads were forced to share same stack, the system would be highly unstable & prone to crashes due to data corruption, loss of isolation, race conditions & undefined behavior. This is why modern operating systems & threading libraries ensure that each thread gets its own stack, preserving isolation & correctness.

②
```
int function (int level){
    int i=0;
    for (i=0; i< level; i++) {
        if (fork() && fork())
            break;
    }
    return 0;
}
```



1st iteration (i=0):

① The original process P_0 execute the first fork():
- → This create a new process (P_1)
- → Total process = P_0 & P_1

② P_0 & P_1 both proceed to execute the second fork():
- → P_0 create a new process (P_2).
- → P_1 creates a new process (P_3).
- → Total process = P_0, P_1, P_2, P_3.

③ After the 2nd fork():
- → only the parent processes (P_0 & P_1) satisfy the if (fork() && fork()) condition.
- → These parent processes execute the break stmt, exiting the loop.

Subsequent iterations (i = 1, 2 ---- level-1):
- → The break stmt ensures that no further iterations of the loop are executed.
- → Therefore, no additional processes are created after the first iteration.

Total No. of Processes:

1st iteration, the fork() calls result in $2^2 = 4$ processes.

For the entire program:
- → Initial Process (P_0) = 1 fork()
- → Additional Processes created in the first iteration : 3 new fork()s

∴ Total No. of process is = $2^{\min(2, level)} = 2^2 = 4$ Ans

Ex:-

① (a) Without Priority Inheritance

Time	Task Execution
0	TD
5	TA (preempts TD)
7	TB (preempts TD)
20	TC (preempts TD)
70	TD (Resumes & completes)
200	End

* Priority inversion might cause delays.

(b) With priority Inheritance

Time	Task Execution
0	T_D
5	T_A
7	T_D (Inherits T_B's priority briefly
8	T_B
20	T_C
70	T_D (Resumes & completes
200	End

* Priority inheritance mitigates the issues, ensuring smoother execution.

Comprehensive Dec 2024 Qs

* 50 MCQ + 1 descriptive questions.
* Related of OS basics & GATE PYQs.
* Simple linux commands.
* 1 Question related C-LOOK
* 1 Question related Memory management
* 1 Question Deadlock, conditions & How to prevent from deadlock methods/techniques explaination with example.
* 1 Question Dining philosopher & it's effectiveness.
* 1 Context switching related.
* 1 Question LRU related in partitions.