2 MARKS :

1) The *TEST and SET lock algorithm* is a simple synchronization technique:

- It uses atomic TEST and SET operations to manage access to a shared resource.

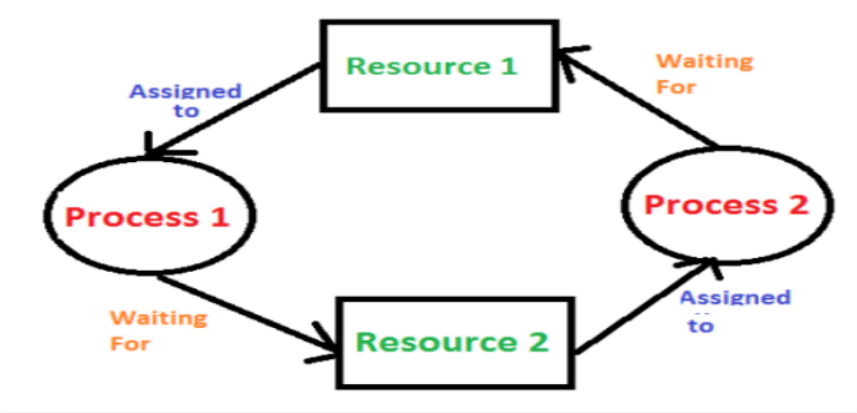
- Threads repeatedly TEST and SET a boolean variable.

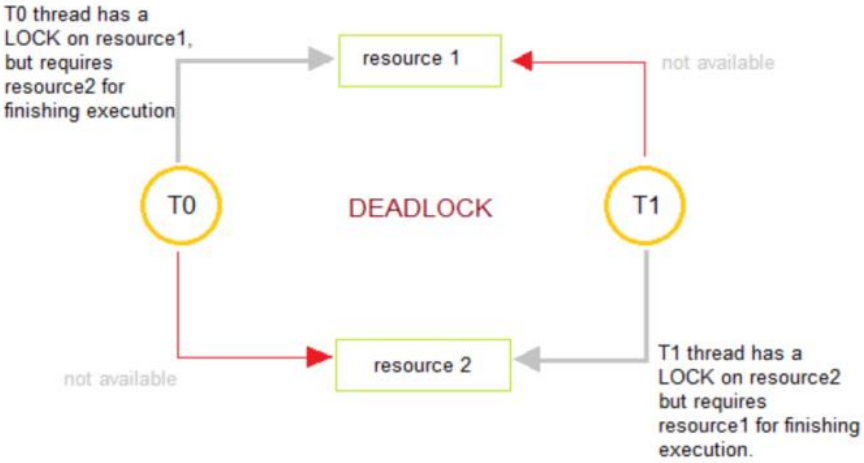
- If the variable is set (true), the thread spins or blocks; if not (false), it sets the variable and proceeds.

- It's simple but can lead to busy-waiting and lacks fairness guarantees.

2) *Process synchronization* is a concept in operating systems and concurrent programming that ensures multiple processes or threads work together correctly and efficiently when accessing shared resources or communicating. It prevents issues like data races and deadlocks by coordinating their execution through synchronization primitives like locks, semaphores, and barriers.

3) A deadlock is a situation in concurrent computing where two or more processes are unable to proceed because they are each waiting for the other(s) to release a resource. It creates a circular wait condition. Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process





4) *External Fragmentation* and *Internal Fragmentation* are both concepts related to memory management, particularly in the context of allocating and using memory efficiently.

External fragmentation occurs when free memory blocks are scattered throughout the memory space, making it challenging to allocate a contiguous block of memory for a process, even when the total free memory is sufficient.

Internal fragmentation happens when memory allocated to a process is larger than what the process actually needs, resulting in wasted memory within the allocated block.

5) Swapping is a memory management technique used in operating systems for various reasons to optimize memory usage and system performance. Here are some key reasons for employing the swapping technique:

1. *Increased Process Size*: Swapping allows processes to be larger than the physical memory (RAM) available. When a process is not actively running, parts of it can be swapped out to disk, freeing up physical memory for other processes.

2. *Multiprogramming*: Swapping enables a higher degree of multiprogramming, where multiple processes can be in memory simultaneously. It allows the operating system to efficiently manage a large number of processes, improving overall system throughput.

3. *Priority-Based Execution*: Swapping allows the operating system to prioritize processes. High-priority processes can be kept in memory while low-priority ones are swapped out, ensuring that critical tasks are executed promptly.

4. *Resource Utilization*: Swapping helps maximize the utilization of available resources. It prevents scenarios where a process cannot be loaded into memory because of fragmentation or insufficient space.

5. *Virtual Memory*: Swapping is a fundamental component of virtual memory systems. It enables the illusion that each process has access to a large, contiguous block of memory, even if the physical memory is limited.

6. *Fault Tolerance*: Swapping can play a role in fault tolerance. Critical data and processes can be swapped out to non-volatile storage (like a hard disk) in case of system crashes or power failures, allowing for recovery upon system restart.

7. *Load Balancing*: Swapping can help distribute the load among different storage devices (e.g., multiple hard drives) by moving less frequently used processes or data to less utilized storage, thus improving overall system performance.

8. *Cache Management*: Swapping can be used to manage caches effectively. Frequently accessed data can be kept in memory, while less frequently used data can be swapped to slower storage devices.

9. *Resource Isolation*: Swapping can isolate resource-intensive or misbehaving processes. If a process consumes too much memory, the operating system can swap it out to prevent it from affecting the overall system's stability.

10. *Memory Defragmentation*: Swapping can be used as a mechanism to defragment memory. By swapping out processes and then compacting the remaining memory, the operating system can reduce fragmentation issues.

In summary, swapping is a crucial technique in memory management that allows efficient use of memory resources, enhances system performance, and enables the management of processes and data that exceed the physical memory capacity. It plays a vital role in modern operating systems and their ability to handle diverse workloads.

6) *Belady's Anomaly* is a phenomenon in page replacement algorithms, particularly in the context of demand-paged virtual memory systems. It occurs when, counterintuitively, increasing the number of available page frames (physical memory) can lead to an increase in the number of page faults (data transfers between disk and RAM) instead of reducing them. In other words, the algorithm exhibits worse performance as more memory is allocated to it. This anomaly challenges the idea that more memory always results in better performance and highlights the importance of choosing the right page replacement algorithm for efficient memory management.

7.)*Paging* and *segmentation* are both memory management techniques used in computer systems, but they have distinct characteristics:

Paging:

- Divides physical memory and processes into fixed-size blocks called "frames."

- Divides logical memory (processes) into fixed-size blocks called "pages."

- Simplifies memory allocation and management but can lead to internal fragmentation.

- Provides efficient memory utilization but may not preserve the logical structure of processes.

Segmentation:

- Divides logical memory (processes) into variable-sized segments based on the program's structure (e.g., code, data, stack).

- Allows for more flexible memory allocation and can reduce internal fragmentation.

- Preserves the logical structure of processes, making it suitable for complex program organization.

- Can result in external fragmentation when segments of different sizes are allocated.

In summary, paging uses fixed-size blocks for both physical and logical memory, while segmentation uses variable-sized segments based on the program's structure. Each has its advantages and disadvantages, and some systems use a combination of both, known as "paged segmentation" or "segmented paging," to leverage the benefits of both approaches.

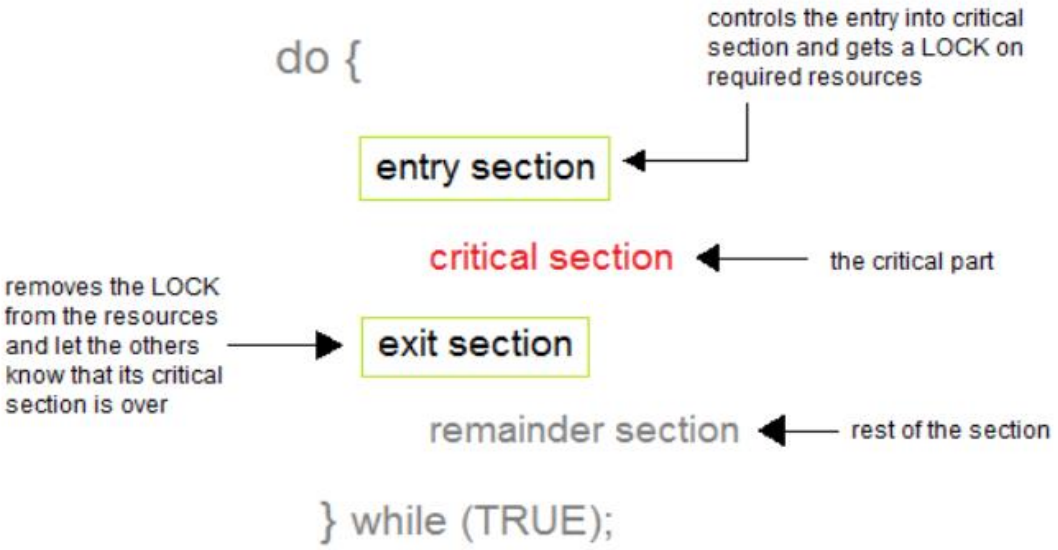
LONG ANSWERS :

8) THE CRITICAL SECTION PROBLEM:

I) A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action.

II) It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.

III) If any other process also wants to execute its critical section, it must wait until the first one finishes.



THE SOLUTION TO THE CRITICAL SECTION PROBLEM:

The main solution for the critical section problem is based on the three main ways.

1. MUTUAL EXCLUSION:

I)Out of a group of cooperating processes, only one process can be in its critical section at a given point of time

II)If process P1 is executing in its critical section, then no other processes can be executing in their critical sections.

III) IMPLICATIONS:

- Critical sections better be focused and short.
- Better not get into an infinite loop in there.

o If a process somehow halts/waits in its critical section, it must not interfere with other processes.

2. PROGRESS:

I) If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

II) If only one process wants to enter, it should be able to.

III)If two or more want to enter, one of them should succeed.

3. BOUNDED WAITING:

I) After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted.

II) So after the limit is reached, system must grant the process permission to get into its critical section.

TWO PROCESS SOLUTIONS:

ALGORITHM 1:

```
do {
while (turn != 1);
 critical section
 turn = j;
remainder section
 }while (1);
```

This Algorithm satisfies Mutual exclusion whereas it fails to satisfy progress requirement since it requires strict alternation of processes in the execution of the critical section.

For example, if turn == 0 and p1 is ready to enter its critical section,p1 cannot do so, even though p0 may be in its remainder section.

ALGORITHM 2:

```
do{
flag[i] =true;
 while (flag[j])
critical section
flag[i]= false;
 remainder section
```

}while(1);

In this solution, the mutual exclusion is met. But the progress is not met. To illustrate this problem, we consider the following

Execution Sequence:

To: P0 sets Flag[0] = true

T1: P1 sets Flag[1] = true

Now P0 and P1 are looping forever in their respective while statements.

ALGORITHM 3:

By combining the key ideas of algorithm 1 and 2, we obtain a correct solution.

Do{

Flag[i] = true;

Turn = j;

While (flag[j] && turn == j);

Critical section

Flag[i] = false;

Remainder section

}while(1);

The algorithm does satisfy the three essential criteria to solve the critical section problem. The three criteria are mutual exclusion, progress, and bounded waiting.

9)

producer consumer problem:

I)This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.

II) Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.

III) Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

IV) The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

do {

```
// produce an item in nextp
wait(empty);
 wait(mutex);
// add the item to the buffer
 signal(mutex);
 signal(full);
 }while(TRUE);
```

The structure of the Producer Process

```
do {
wait(full);
wait(mutex);
// remove an item from buffer into nextc
signal(mutex);
signal(empty);
// consume the item in nextc
}while(TRUE);
```

The structure of the Consumer Process

EXAMPLE CODE:

```
#include <stdio.h>
#include <cono.h>
int main()
{
int s,n,b=0,p=0,c=0;
clrscr();
 printf("\n producer and consumer problem");
Do
 {
 printf("\n menu");
 printf("\n 1.producer an item");
```

```
printf("\n 2.consumer an item");
printf("\n 3.add item to the buffer");
printf("\n 4.display status");
printf("\n 5.exit");
printf("\n enter the choice");
scanf("%d",&s);
switch(s)
{ case 1:
p=p+1;
 printf("\n item to be produced");
break;
 case 2:
 if(b!=0)
 { c=c+1;
b=b-1;
 printf("\n item to be consumed");
} else
{ printf("\n the buffer is empty please wait...");
 } break;
 case 3:
 if(b<n)
{
If p!=0
{
B=b+1;
printf("\n item added to buffer");
 }
else
printf("\n no.of items to add...");
```

```
} else
printf("\n buffer is full,please wait");
break;
case 4:
 printf("no.of items produced :%d",p);
printf("\n no.of consumed items:%d",c);
printf("\n no.of buffered item:%d",b);
 break;
case 5:
exit(0);
}
while(s<=5);
 getch();
return 0
}
```

2}THE READERS WRITERS PROBLEM:

I) In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading, or instead of reading it.
II) There are various type of readers-writers problem, most centered on relative priorities of readers and writers.

SOLUTION:

I) From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

II) Here, we use one mutex m and a semaphore w. An integer variable sread is used to maintain the number of readers currently accessing the resource. The variable swrite is initialized to 0. A value of 1 is given initially to m and w.

III) Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the sread variable.

EXAMPLE CODE :

```
#include<stdio.h>
```

```
#include<conio.h>
#include <process.h>
void main()
{
typedef int semaphore;
semaphore sread=0,
swrite=0; int ch,r=0;
clrscr();
printf("\nReader writer");
do { printf("\nMenu");
printf("\n\t 1.Read from file");
printf("\n \t2.Write to file");
printf("\n \t 3.Exit the reader");
printf("\n \t 4.Exit the writer");
printf("\n \t 5.Exit");
printf("\nEnter your choice:")
scanf("%d",&ch);
switch(ch) {
case 1:
if(swrite==0)
{ sread=1; r+=1;
printf("\nReader %d reads",r);
}
Else
{ printf("\n Not possible");
}
break;
case 2:
if(sread==0 && swrite==0)
```

```
 { swrite=1;
 printf("\nWriter in Progress");
 }
 else if(swrite==1)
 { printf("\nWriter writes the files");
 }
 else if(sread==1)
{ printf("\nCannot write while reader reads the file");
 }
 Else
 printf("\nCannot write file");
 break;
case 3:
 if(r!=0)
 {
 printf("\n The reader %d closes the file",r);
 r-=1;
 }
else if(r==0)
 {
printf("\n Currently no readers access the file");
 sread=0;
}
 else if(r==1)
 {
 printf("\nOnly 1 reader file");
 }
 else printf("%d reader are reading the file\n",r);
break;
```

```
case 4:
 if (swrite==1)
{
 printf("\nWriter close the file");
swrite=0;
 }
 else printf("\nThere is no writer in the file");
 break;
 case 5:
 exit(0);
 }
 }
 while(ch<6);
getch();
}
```

10. BANKERS ALGORITHM:

Process	Allocation			maximum			Available		
	A	B	C	A	B	C	A	B	C
P_0	1	1	2	4	3	3	2	1	0
P_1	2	1	2	3	2	2			
P_2	4	0	1	9	0	2			
P_3	0	2	0	7	5	3			
P_4	1	1	2	1	1	2			

(i) content of need matrix

$Need(i,j) = max(i,j) - Allocation(i,j)$

Need of $P_0 = (4,3,3) - (1,1,2) = (3,2,1)$

Need for $P_1 = (3,2,2) - (2,1,2) = (1,1,0)$

Need for $P_2 = (9,0,2) - (4,0,1) = (5,0,1)$

Need for $P_3 = (7,5,3) - (0,2,0) = (7,3,3)$

Need for $P_4 = (1,1,2) - (1,1,2) = (0,0,0)$

Need matrix

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

(iii) check if the system is safe

to check system safe if need(i,j) is less than or equal to available(i,j) then

work = Allocation(i,j) + Available(i,j)

$P_0 \rightarrow (3,2,1) \leq (2,1,0)$ ✗

$P_1 \rightarrow (1,1,0) \leq (2,1,0)$ ✓

work = (2,1,0) + (2,1,2)

= (4,2,2) (Available)

$P_2 \rightarrow (5,0,1) \leq (4,2,2)$ ✗

$P_3 \rightarrow (7,3,3) \leq (4,2,2)$ ✗

$P_4 \rightarrow (0,0,0) \leq (4,2,2)$ ✓

work = (4,2,2) + (1,1,2)

= (5,3,4)

Avaible = (5,3,4)

$P_0 \rightarrow (3,2,1) \leq (5,3,4)$ ✓

work = (5,3,4) + (1,1,2)

= (6,4,6)

$P_2 \rightarrow (5,0,1) \leq (6,4,6)$ ✓

work = (6,4,6) + (4,0,1)

= (10,4,7)

$P_3 \rightarrow (7,3,3) \leq (10,4,7)$ ✓

= (10,4,7) + (0,2,0)

= (10,6,7) ✓

Available = (10,6,7)

∴ Safe :→ $P_1 \rightarrow P_4 \rightarrow P_0 \rightarrow P_2 \rightarrow P_3$

(ii) Determine total sum of each type of resou

- total Allocation of A = 1+2+4+1 = 8

  total Allocation of B = 1+1+2+1 = 5

  total Allocation of C = 2+2+1+2 = 7

  Avaible of A = 2

  Available of B = 1

  Available of C = 0

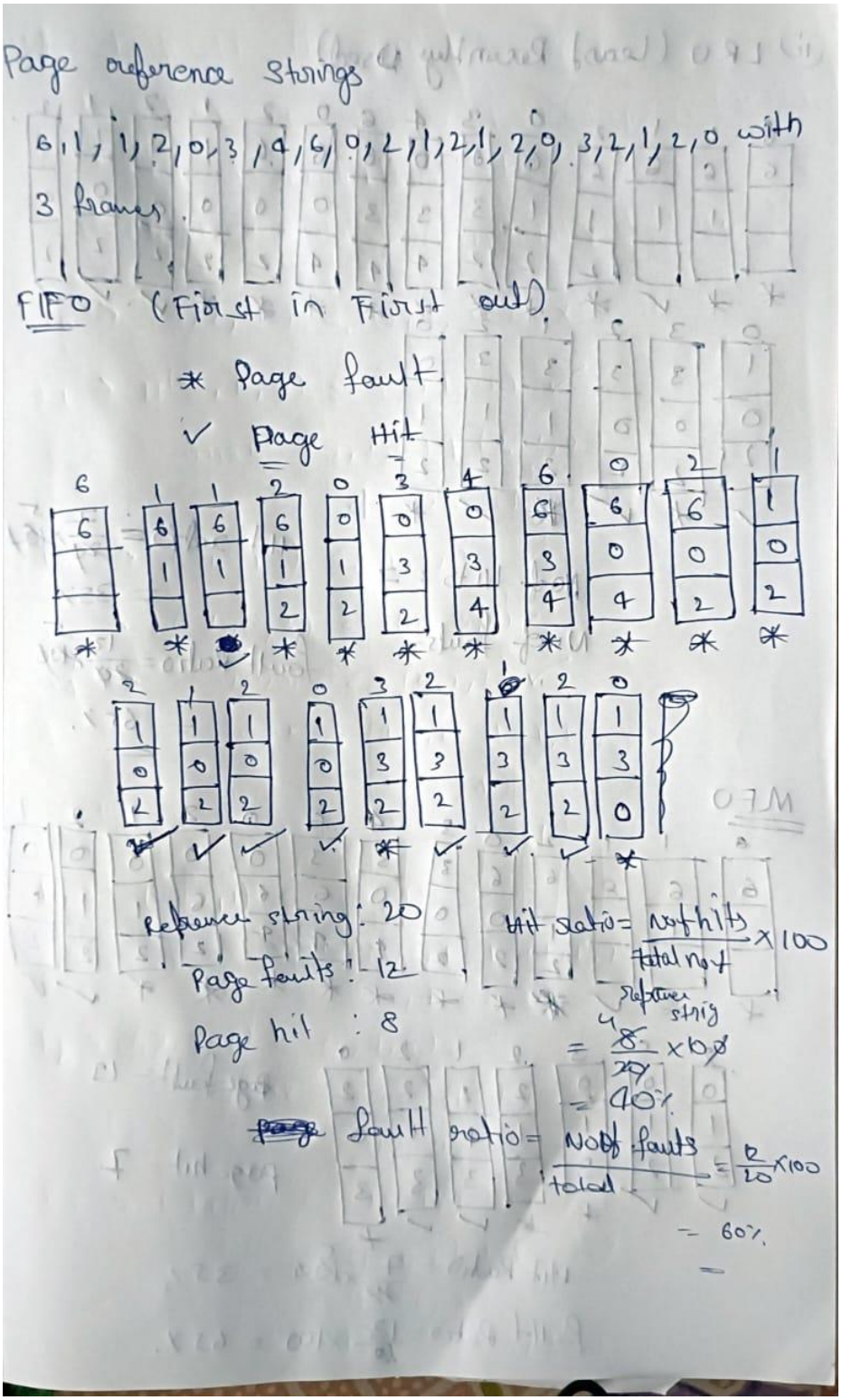
∴ total sum of A = 8+2 = 10

" " " B = 5+1 = 6

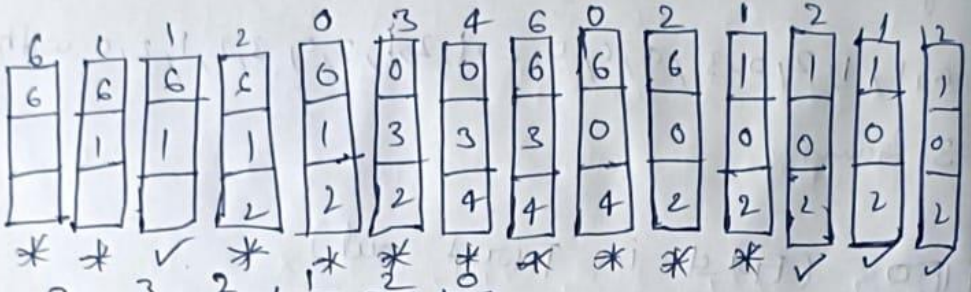
" " " C = 7+0 = 7

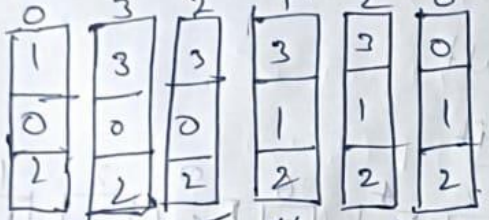
∴ total sum (A, B, C) = (10, 6, 7) ✓

13. **PAGE REFERENCE STRING:**



(ii) LRU (Least Recently Used)





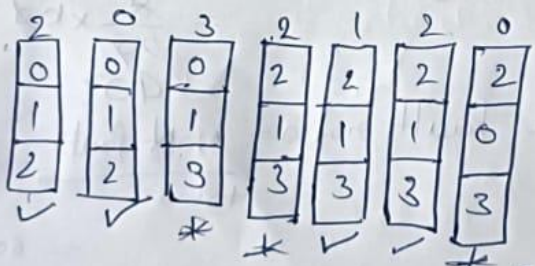
No of Hits = 7

No of faults = 13

Hit ratio = $\frac{7}{20} \times 100 = 35\%$.

Fault Ratio = $\frac{13}{20} \times 100 = 65\%$.

MFO



Page fault = 13

page hit = 7

Hit Ratio = $\frac{7}{20} \times 100 = 35\%$.

Fault Ratio = $\frac{13}{20} \times 100 = 65\%$.