REG.NO.:

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
FALL SEMESTER 2024-2025

SLOT: B1+TB1

Programme Name & Branch : **B.Tech. CSE**
Course Code and Course Name : **BCSE303L – Operating Systems**
Date of Examination : **14.10.2024**
Exam Duration : **90 minutes** **Maximum Marks: 50**

General Instruction(s): Answer All Questions

Q. No	Question
1.	Consider a system with 5 processes (P0, P1, P2, P3, P4) and 4 resource types (A, B, C, D). The Allocation, Max, and Available matrices are given as follows:

	Allocation				Max				Available			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	2	0	0	1	4	2	1	2	3	3	2	1
P1	3	1	2	1	5	2	5	2				
P2	2	1	0	3	2	3	1	6				
P3	1	3	1	2	1	4	2	4				
P4	1	4	3	2	3	6	6	5				

Using the Banker's Algorithm, answer the following questions:

1. Safety Check: Determine if the system is currently in a safe state. Show the sequence of processes and work vectors used to verify this. **(5 Marks)**
2. Resource Request: Suppose process (P1) makes a request for resources: (1, 1, 0, 0). Check if this request can be granted by: **(5 Marks)**
   - Verifying if the request is less than or equal to the process's remaining need.
   - Verifying if the request is less than or equal to the available resources.
   - Simulating the allocation and then performing a safety check to ensure the system remains in a safe state.

Answer:

1. Need Matrix (calculated as Max - Allocation):

	Need			
	A	**B**	**C**	**D**
P0	2	2	1	1
P1	2	1	3	1
P2	0	2	1	3
P3	0	1	1	2
P4	2	2	3	3

Initialize:

- Work = Available = [3, 3, 2, 1]
- Finish = [False, False, False, False, False]

Sequence of processes:

Check P0: [2, 2, 1, 1] <= [3, 3, 2, 1] (True)

- Work = Work + Allocation[i] = [3, 3, 2, 1] + [2, 0, 0, 1] = [5, 3, 2, 2]
- Finish[0] = True

Check P1: [2, 1, 3, 1] <= [5, 3, 2, 2] (False)
Check P2: [0, 2, 1, 3] <= [5, 3, 2, 2] (False)
Check P3: [0, 1, 1, 2] <= [5, 3, 2, 2] (True)

- Work = Work + Allocation[i] = [5, 3, 2, 2] + [1, 3, 1, 2] = [6, 6, 3, 4]
- Finish[3] = True

Check P4: [2, 2, 3, 3] <= [6, 6, 3, 4] (True)
- Work = Work + Allocation[i] = [6, 6, 3, 4] + [1, 4, 3, 2] = [7, 10, 6, 6]
- Finish[4] = True

Check P1: [2, 1, 3, 1] <= [7, 10, 6, 6] (True)
- Work = Work + Allocation[i] = [7, 10, 6, 6] + [3, 1, 2, 1] = [10, 11, 8, 7]
- Finish[1] = True

Check P2: [0, 2, 1, 3] <= [10, 11, 8, 7] (True)
- Work = Work + Allocation[i] = [10, 11, 8, 7] + [2, 1, 0, 3] = [12, 12, 8, 10]
- Finish[2] = True

All processes are finished, so the system is in a safe state.
Safe sequence: [P0, P3, P4, P1, P2]

2. Resource Request:

P0 requests (1, 1, 0, 0)

Verify if Request[1] ≤ Need[1]:
- Request[1] = [1, 1, 0, 0]
- Need[1] = [2, 1, 3, 1]
- [1, 1, 0, 0] ≤ [2, 1, 3, 1] (True)

Verify if Request[1] ≤ Available:
- Available = [3, 3, 2, 1]
- [1, 1, 0, 0] ≤ [3, 3, 2, 1] (True)

Simulate the allocation:
- Available = Available - Request = [3, 3, 2, 1] - [1, 1, 0, 0] = [2, 2, 2, 1]
- Allocation[P1] = Allocation[P1] + Request = [3, 1, 2, 1] + [1, 1, 0, 0] = [4, 2, 2, 1]
- Need[P1] = Need[P1] - Request = [2, 1, 3, 1] - [1, 1, 0, 0] = [1, 0, 3, 0]

New:

	New_Allocation				**New_Max**				**New_Available**				**New_Need**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
P1	4	2	2	1	5	2	5	2					1	0	3	1
P2	2	1	0	3	2	3	1	6					0	2	1	3
P3	1	3	1	2	1	4	2	4					0	1	1	2
P4	1	4	3	2	3	6	6	5					2	2	3	3

Initialize:
- Work = Available = [2, 2, 2, 1]
- Finish = [False, False, False, False, False]

Sequence of processes:

Check P0: [2, 2, 1, 1] <= [2, 2, 2, 1] (True)

	- Work = Work + Allocation[i] = [2, 2, 2, 1] + [2, 0, 0, 1] = [4, 2, 2, 2] - Finish[0] = True Check P1: [1, 0, 3, 1] <= [4, 2, 2, 2] (False) Check P2: [0, 2, 1, 3] <= [4, 2, 2, 2] (False) Check P3: [0, 1, 1, 2] <= [4, 2, 2, 2] (True) - Work = Work + Allocation[i] = [4, 2, 2, 2] + [1, 3, 1, 2] = [5, 5, 3, 4] - Finish[3] = True Check P4: [2, 2, 3, 3] <= [5, 5, 3, 4] (True) - Work = Work + Allocation[i] = [5, 5, 3, 4] + [1, 4, 3, 2] = [6, 9, 6, 6] - Finish[4] = True Check P1: [1, 0, 3, 1] <= [6, 9, 6, 6] (True) - Work = Work + Allocation[i] = [6, 9, 6, 6] + [4, 2, 2, 1] = [10, 11, 8, 7] - Finish[1] = True Check P2: [0, 2, 1, 3] <= [10, 11, 8, 7] (True) - Work = Work + Allocation[i] = [10, 11, 8, 7] + [2, 1, 0, 3] = [12, 12, 8, 10] - Finish[2] = True **All processes are finished, so the system is in a safe state.** **Safe sequence: [P0, P3, P4, P1, P2]** **Result:** - System is currently in a safe state. - The resource request by P1 for [1, 1, 0, 0] can be granted.
2.	a) A university library has a study room that can accommodate up to 20 students at a time. Students can enter the study room only if there is space available. Once inside, students can request to use a shared whiteboard for group discussions. Only one group can use the whiteboard at a time. If the whiteboard is in use, other groups must wait until it becomes available. Design a synchronization mechanism using semaphores to manage the entry of students into the study room and the use of the shared whiteboard. Provide the pseudo code for your solution and explain how it ensures mutual exclusion and proper coordination.
	Answer: **Need for Process Synchronization** • **Mutual Exclusion**: Ensure that only one group uses the whiteboard at a time. • **Coordination**: Manage the entry and exit of students to ensure the study room does not exceed its capacity. • **Order**: Control the access to the whiteboard so that groups use it in the order they request it. **Semaphores Used (1 Mark)** • **study_room_capacity**: Controls the number of students in the study room (maximum 20). • **whiteboard**: Ensures that only one group can use the whiteboard at a time. • **mutex**: Ensures mutual exclusion when students enter or exit the study room. **Pseudo Code (4 Marks)** // Initialize semaphores

```
Semaphore study_room_capacity = 20; // Maximum 20 students
Semaphore whiteboard = 1;         // Only one group can use the whiteboard at a time
Semaphore mutex = 1;              // Mutual exclusion for entering/exiting the study room

// Student process
void student() {
   while (true) {
      // Enter the study room if there is space
      wait(study_room_capacity);
      wait(mutex);
      enter_study_room();
      signal(mutex);

      // Request to use the whiteboard
      wait(whiteboard);
      use_whiteboard();
      signal(whiteboard);

      // Exit the study room
      wait(mutex);
      exit_study_room();
      signal(mutex);
      signal(study_room_capacity);
   }
}

// Functions representing actions
void enter_study_room() {
   // Student enters the study room
   printf("Student enters the study room.\n");
}

void use_whiteboard() {
   // Student uses the whiteboard
   printf("Student uses the whiteboard.\n");
}
```

```
void exit_study_room() {
  // Student exits the study room
  printf("Student exits the study room.\n");
}
```

Explanation (2 Marks)

1. **Initialization**:
   - study_room_capacity is initialized to 20, representing the maximum number of students allowed in the study room.
   - whiteboard is initialized to 1, ensuring that only one group can use the whiteboard at a time.
   - mutex is initialized to 1, ensuring mutual exclusion when students enter or exit the study room.
2. **Student Process**:

Entering the Study Room:

- The student waits on the study_room_capacity semaphore to ensure there is space available.
- The student then waits on the mutex semaphore to ensure mutual exclusion while entering the study room.
- After entering, the student signals the mutex semaphore to allow other students to enter or exit.

Using the Whiteboard:

- The student waits on the whiteboard semaphore to ensure that no other group is using the whiteboard.
- After using the whiteboard, the student signals the whiteboard semaphore to allow other groups to use it.

Exiting the Study Room:

- The student waits on the mutex semaphore to ensure mutual exclusion while exiting the study room.
- After exiting, the student signals the mutex semaphore to allow other students to enter or exit.
- The student then signals the study_room_capacity semaphore to indicate that a space has become available in the study room.

b) Describe Peterson's solution for mutual exclusion. How does it ensure that only one process enters the critical section at a time?

Answer: Peterson's Solution

Peterson's solution uses two shared variables:

- flag[2]: An array of boolean values where flag[i] indicates if process Pi wants to enter the critical section.
- turn: An integer variable that indicates whose turn it is to enter the critical section.

How It Ensures Mutual Exclusion (2 Marks)

	• Mutual Exclusion: The critical section is only entered if the other process is not interested (flag[j] == false) or if it is the current process's turn (turn != j). This ensures that only one process can be in the critical section at a time. • Progress: If no process is in the critical section, the process that wants to enter will eventually be able to do so. The turn variable ensures that if both processes want to enter the critical section simultaneously, they will take turns. • Bounded Waiting: Each process will enter the critical section within a bounded number of attempts. The turn variable ensures that each process gets a fair chance to enter the critical section. **Example Scenario (1 Mark)** • If P0 wants to enter the critical section, it sets flag[0] = true and turn = 1. If P1 is not interested (flag[1] = false) or it is P0's turn (turn != 1), P0 enters the critical section. • If P1 also wants to enter, it sets flag[1] = true and turn = 0. Now, if P0 is still interested (flag[0] = true) and it is P0's turn (turn == 0), P1 will wait. Once P0 exits the critical section and sets flag[0] = false, P1 can enter. This way, Peterson's solution ensures that only one process enters the critical section at a time, maintaining mutual exclusion and preventing race conditions.
3.	a) In a study group, there are five scholars who need to use shared resources: pens and notebooks. Each scholar requires both a pen and a notebook to take notes. However, only one scholar can use a pen or a notebook at a time. If a scholar cannot acquire both a pen and a notebook, they must wait until both are available. The goal is to ensure that no two scholars are using the same pen or notebook simultaneously and to avoid deadlock where scholars are waiting indefinitely for resources. Design a synchronization mechanism using the principles of the Dining Philosophers problem to manage the allocation of pens and notebooks to scholars. Provide the pseudo code for your solution and explain how it ensures mutual exclusion, prevents deadlock, and allows scholars to take notes in an orderly manner.

	Answer:
	Need for Process Synchronization
	• **Mutual Exclusion**: Ensure that only one scholar can use a pen or a notebook at a time.
	• **Coordination**: Manage the allocation of pens and notebooks to prevent deadlock and ensure that scholars can take notes without indefinite waiting.
	• **Order**: Control the access to pens and notebooks so that scholars can take notes in an orderly manner.
	Semaphores Used (1 Mark)
	• **pen[i]**: Controls access to each of the five pens.
	• **notebook[i]**: Controls access to each of the five notebooks.
	• **mutex**: Ensures mutual exclusion when scholars try to acquire pens and notebooks.
	Pseudo Code (4 Marks)
	// Initialize semaphores
	Semaphore pen[5] = {1, 1, 1, 1, 1}; // Each pen is initially available
	Semaphore notebook[5] = {1, 1, 1, 1, 1}; // Each notebook is initially available

```
Semaphore mutex = 1;              // Mutual exclusion for acquiring resources

// Scholar process
void scholar(int i) {
  while (true) {
    // Think (prepare for taking notes)
    prepare_notes();

    // Try to acquire both a pen and a notebook
    wait(mutex);
    wait(pen[i]);
    wait(notebook[i]);
    signal(mutex);

    // Critical section: Take notes
    take_notes();

    // Release both the pen and the notebook
    signal(pen[i]);
    signal(notebook[i]);
  }
}

// Functions representing actions
void prepare_notes() {
  // Scholar is preparing to take notes
  printf("Scholar %d is preparing to take notes.\n", i);
}

void take_notes() {
  // Scholar is taking notes
  printf("Scholar %d is taking notes.\n", i);
}
```

Explanation (2 Marks)

Initialization:

- pen[5] is initialized to {1, 1, 1, 1, 1}, indicating that all five pens are initially available.
- notebook[5] is initialized to {1, 1, 1, 1, 1}, indicating that all five notebooks are initially available.
- mutex is initialized to 1, ensuring mutual exclusion when scholars try to acquire pens and notebooks.

Scholar Process:

Preparing for Taking Notes:

- The scholar prepares for taking notes before attempting to acquire a pen and a notebook.

Acquiring Resources:

- The scholar waits on the mutex semaphore to ensure mutual exclusion while acquiring resources.
- The scholar then waits on the pen[i] and notebook[i] semaphores to acquire the required pen and notebook.
- After acquiring both resources, the scholar signals the mutex semaphore to allow other scholars to attempt to acquire resources.

Taking Notes:

- Once the scholar has acquired both resources, they enter the critical section and take notes.

Releasing Resources:

- After taking notes, the scholar signals the pen[i] and notebook[i] semaphores to release the resources, making them available for other scholars.

b) Compare shared memory and message passing in Interprocess Communication (IPC). (Any four comparisons 4 Marks)

Feature	Shared Memory	Message Passing
Definition	Memory that can be accessed by multiple **processes**.	Communication through sending and receiving messages.
Speed	Generally faster due to direct memory access.	Slower due to the overhead of message handling.
Complexity	More complex to implement synchronization mechanisms.	Simpler as it abstracts synchronization.
Data Size	Suitable for large amounts of data.	Better for smaller amounts of data.
Synchronization	Requires explicit synchronization (e.g., semaphores, mutexes).	Implicit synchronization through message queues.
Scalability	Can be less scalable due to synchronization overhead.	More scalable as it decouples processes.
Use Cases	High-performance applications needing fast data access.	Distributed systems, network communication.
Resource Management	Requires careful management of shared resources.	Easier resource management with message queues.
Fault Tolerance	Less fault-tolerant; issues in one process can affect others.	More fault-tolerant; isolated process failures.

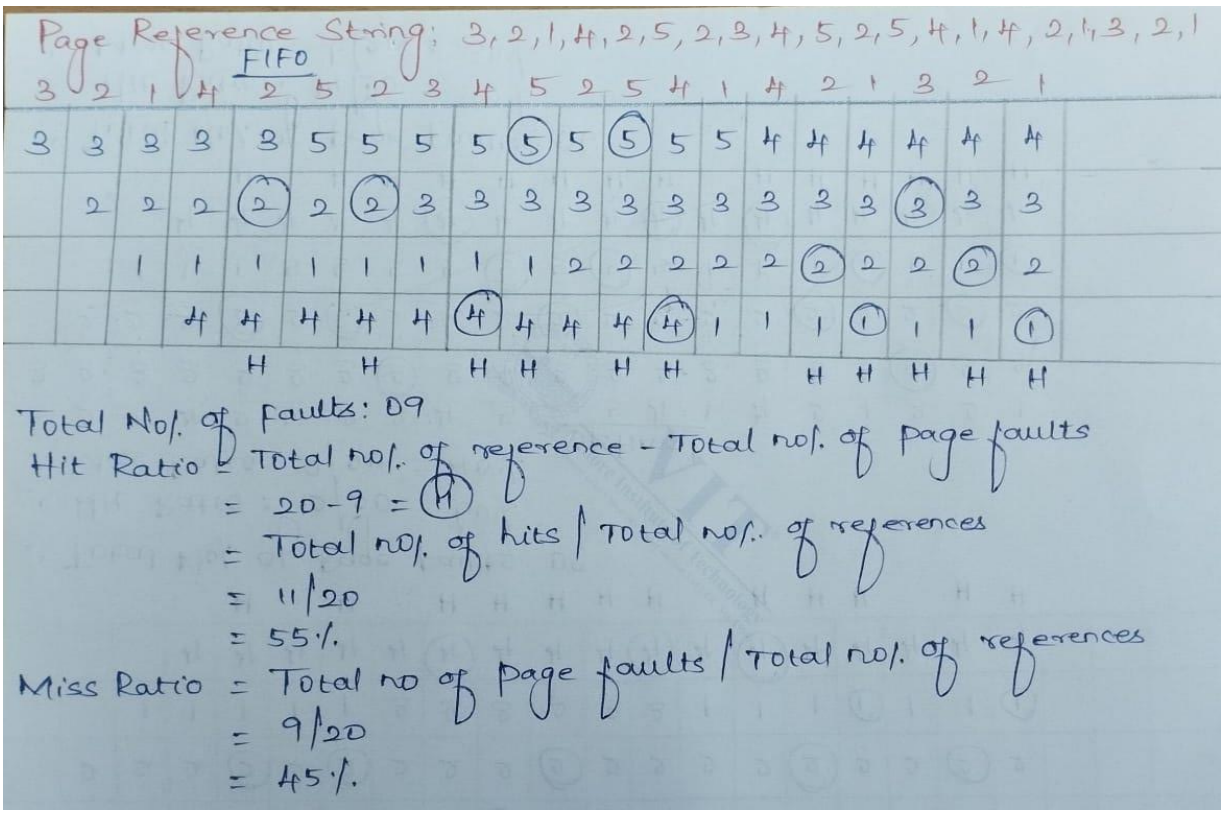
4. Consider a system with 4 page frames and the following page reference string: 3, 2, 1, 4, 2, 5, 2, 3, 4, 5, 2, 5, 4, 1, 4, 2, 1, 3, 2, 1.
Using the FIFO, LRU, and Optimal page replacement algorithms:

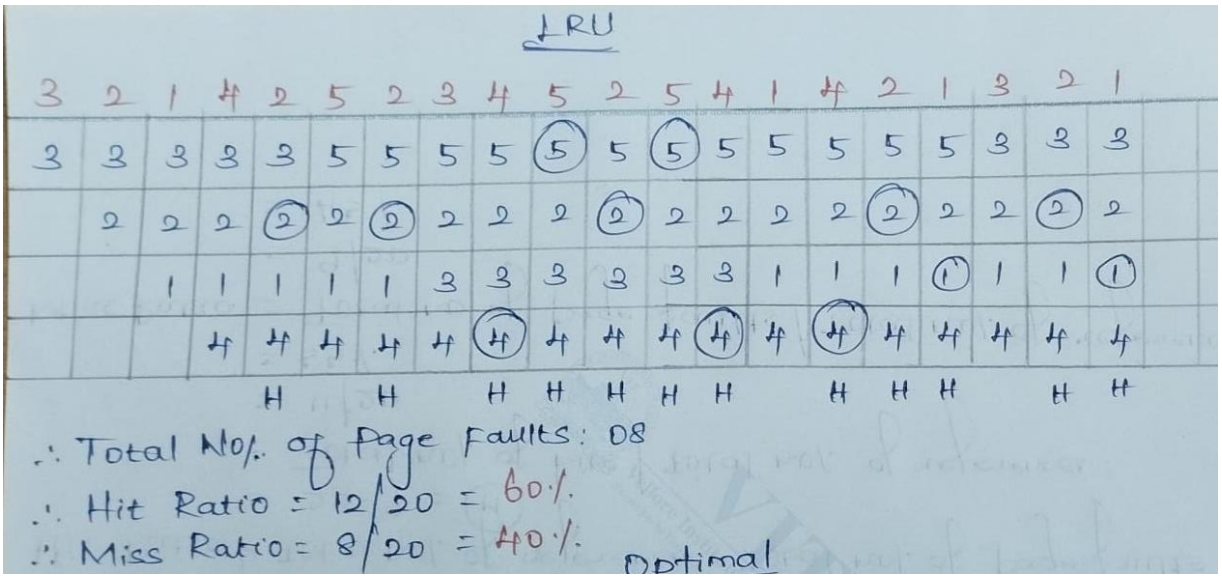
- Clearly show the page replacement activities for each algorithm.

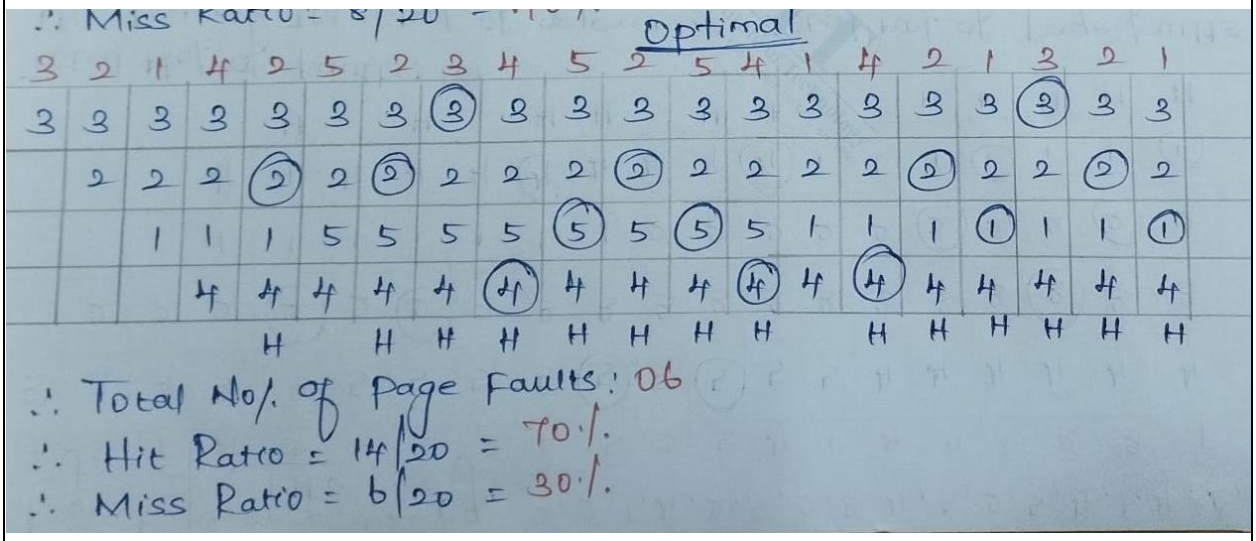
- Calculate the number of page faults for each algorithm.
- Determine the hit ratio and miss ratio for each algorithm.

Answer:

(FIFO: 3 Marks, LRU:3.5 Marks, Optimal:3.5 Marks)







5. A computer system has a total memory capacity of 1000 KB, divided into the following free memory blocks (in KB): 200, 350, 150, 400, and 250. Eight processes require memory allocations as follows:

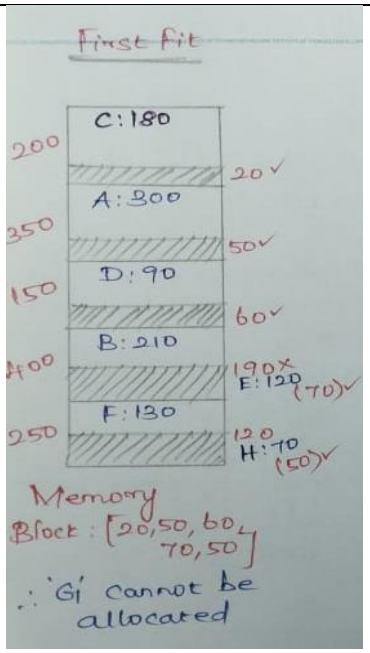
- Process A: 300 KB
- Process B: 210 KB
- Process C: 180 KB
- Process D: 90 KB
- Process E: 120 KB
- Process F: 130 KB
- Process G: 160 KB
- Process H: 70 KB

Using the First Fit, Best Fit, and Worst Fit allocation algorithms, determine how the memory will be allocated to each process. Illustrate the state of the memory blocks after each allocation for each algorithm. Analyze the efficiency of each algorithm based on the final state of the memory blocks.

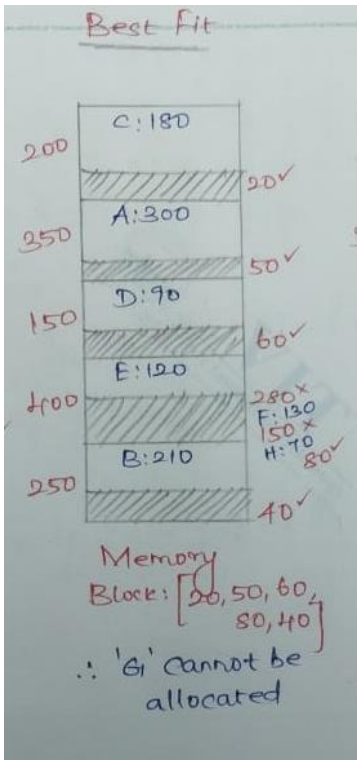
Answer:
(First: 3 Marks, Best:3 Marks, Worst:3 Marks, Analyze: 1 Mark)

First Fit Allocation
1. Process A (300 KB):
Allocated to the first block that fits: 350 KB.
Memory blocks: [200, 50, 150, 400, 250]
2. Process B (210 KB):
Allocated to the first block that fits: 400 KB.
Memory blocks: [200, 50, 150, 190, 250]
3. Process C (180 KB):
Allocated to the first block that fits: 200 KB.

Memory blocks: [20, 50, 150, 190, 250]
4. Process D (90 KB):
Allocated to the first block that fits: 150 KB.
Memory blocks: [20, 50, 60, 190, 250]
5. Process E (120 KB):
Allocated to the first block that fits: 190 KB.
Memory blocks: [20, 50, 60, 70, 250]
6. Process F (130 KB):
Allocated to the first block that fits: 250 KB.
Memory blocks: [20, 50, 60, 70, 120]
7. Process G (160 KB):
Cannot be allocated (no block large enough).
8. Process H (70 KB):
Allocated to the first block that fits: 120 KB.
Memory blocks: [20, 50, 60, 70, 50]

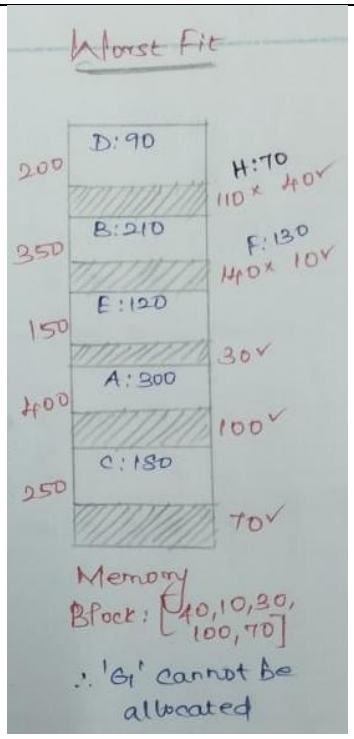


Best Fit Allocation
1. Process A (300 KB):
Allocated to the smallest block that fits: 350 KB.
Remaining blocks: [200, 50, 150, 400, 250]
2. Process B (210 KB):
Allocated to the smallest block that fits: 250 KB.
Remaining blocks: [200, 50, 150, 400, 40]
3. Process C (180 KB):
Allocated to the smallest block that fits: 200 KB.
Remaining blocks: [20, 50, 150, 400, 40]
4. Process D (90 KB):
Allocated to the smallest block that fits: 150 KB.
Remaining blocks: [20, 50, 60, 400, 40]
5. Process E (120 KB):
Allocated to the smallest block that fits: 400 KB.
Remaining blocks: [20, 50, 60, 280, 40]
6. Process F (130 KB):
Allocated to the smallest block that fits: 280 KB.
Remaining blocks: [20, 50, 60, 150, 40]
7. Process G (160 KB):
Cannot be allocated (no block large enough).
8. Process H (70 KB):
Allocated to the smallest block that fits: 150 KB.
Remaining blocks: [20, 50, 60, 80, 40]



Worst Fit Allocation
1. Process A (300 KB):
Allocated to 400 KB block.
Remaining blocks: [200, 350, 150, 100, 250]

2. Process B (210 KB):
Allocated to 350 KB block.
Remaining blocks: [200, 140, 150, 100, 250]
3. Process C (180 KB):
Allocated to 250 KB block.
Remaining blocks: [200, 140, 150, 100, 70]
4. Process D (90 KB):
Allocated to 200 KB block.
Remaining blocks: [110, 140, 150, 100, 70]
5. Process E (120 KB):
Allocated to 150 KB block.
Remaining blocks: [110, 140, 30, 100, 70]
6. Process F (130 KB):
Allocated to 140 KB block.
Remaining blocks: [110, 10, 30, 100, 70]
7. Process G (160 KB):
Cannot be allocated (no block large enough).
8. Process H (70 KB): Allocated to 110 KB block.
Remaining blocks: [40, 10, 30, 100, 70]



Summary of Memory States:

- First Fit: [20, 50, 60, 70, 50]
- Best Fit: [20, 50, 60, 80, 40]
- Worst Fit: [40, 10, 30, 100, 70]

****