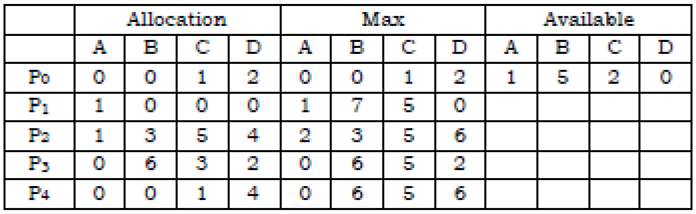
Department of Computer Science & Engineering

IAT2 Answer Scheme and Solution

BCS303-Operating System

Sub:	Operating System					Sub Code:	BCS303	Branch : CSE
Date:	14.12.2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	III/ A, B, C	OBE

	Answer any FIVE FULL Questions	**MARKS**	CO	RBT
1	What is paging? Explain paging hardware with TLB with neat diagram.	10	CO4	L2
2	What is resource allocation graph? Consider an example to illustrate how it useful to avoid deadlock situation?	10	CO3	L2
3	Consider a disk with 200 tracks and the queue has random requests from different processes in the order: 55,58,39,18,90,160,150,38,184. Initial arm head is at 100.Find the average seek time using FIFO,SCAN,SSTF?	10	CO4	L3
4	Consider the following snapshot of the system:  Answer the following questions using Banker's algorithm: i)Is the system in a safe state? If yes, mention the safe sequence. ii)If a request from process P1 arrives for (0,4,2,0) can the request be granted immediately?	10	CO3	L3
5	Explain paging and segmentation?	10	CO5	L2

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

6	a) Distinguish between single level directory structure and two level directory structure? b) Explain. Various file operations?	5+5	CO5	L3

Solution

1.

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory and thus eliminates problems of fitting varying sized memory chunks onto the backing store. Paging divides the virtual address space of a process into fixed-size blocks called pages, which are mapped to physical memory frames.

Paging Hardware with TLB

Components of Paging Hardware:

1. Page Table: A data structure used by the virtual memory system to store the mapping between virtual addresses and physical addresses. Each entry in the page table holds the base address of a page in physical memory.
2. Page Table Entry (PTE): Contains information about a page, including the frame number and status bits (valid, read/write permissions, etc.).
3. Translation Lookaside Buffer (TLB): A small, fast cache that stores recent translations of virtual page numbers to physical frame numbers to speed up the memory access process.
4. Memory Management Unit (MMU): A hardware component responsible for handling all memory and caching operations associated with the CPU.

Steps in Address Translation using TLB:

1. Virtual Address Generation: The CPU generates a virtual address (VA) that consists of a virtual page number (VPN) and an offset.
2. TLB Lookup: The MMU checks if the VPN is present in the TLB.
    - If there is a TLB hit, the physical frame number (PFN) is retrieved from the TLB.
    - If there is a TLB miss, the MMU retrieves the PFN from the page table in the main memory.

3. Page Table Lookup (if TLB miss): The VPN is used to index into the page table to find the corresponding PFN.
4. Update TLB (if TLB miss): The MMU updates the TLB with the new VPN-to-PFN mapping.
5. Physical Address Formation: The PFN retrieved from either the TLB or the page table is combined with the offset from the VA to form the physical address (PA).
6. Memory Access: The physical address is used to access the actual memory location.

2.

A Resource Allocation Graph (RAG) is a directed graph used to represent the allocation of resources to processes in a system. It is a tool used in operating systems to detect and avoid deadlocks. The graph consists of two types of nodes:

Process Nodes (P): Represent processes, typically shown as circles.

Resource Nodes (R): Represent resources, typically shown as squares.

Edges in the graph indicate the allocation and request of resources:

Request Edge: A directed edge from a process to a resource (P → R) indicates that the process is requesting the resource.

Assignment Edge: A directed edge from a resource to a process (R → P) indicates that the resource has been allocated to the process.

Deadlock Detection and Avoidance

In a Resource Allocation Graph, deadlock can occur if there is a cycle in the graph. If a cycle exists, it indicates that a set of processes are in a circular wait condition, each waiting for a resource held by another process in the cycle.

To avoid deadlock, the system can use the Resource Allocation Graph in the following ways:

Cycle Detection: By regularly checking the RAG for cycles, the system can detect potential deadlocks. If a cycle is found, the system can take corrective actions, such as resource preemption, process termination, or rolling back processes to a safe state.

Safe State Verification: Before granting a resource request, the system can check if the new state would still be a safe state, i.e., a state where no cycle would form. If granting the request results in a cycle, the request is denied, ensuring that the system remains deadlock-free.A Resource Allocation Graph (RAG) is a directed graph used to represent the allocation of resources to processes in a system. It is a tool used in operating systems to detect and avoid deadlocks. The graph consists of two types of nodes:

1. Process Nodes (P): Represent processes, typically shown as circles.
2. Resource Nodes (R): Represent resources, typically shown as squares.

Edges in the graph indicate the allocation and request of resources:

- Request Edge: A directed edge from a process to a resource (P → R) indicates that the process is requesting the resource.
- Assignment Edge: A directed edge from a resource to a process (R → P) indicates that the resource has been allocated to the process.

Deadlock Detection and Avoidance

In a Resource Allocation Graph, deadlock can occur if there is a cycle in the graph. If a cycle exists, it indicates that a set of processes are in a circular wait condition, each waiting for a resource held by another process in the cycle.

To avoid deadlock, the system can use the Resource Allocation Graph in the following ways:

1. Cycle Detection: By regularly checking the RAG for cycles, the system can detect potential deadlocks. If a cycle is found, the system can take corrective actions, such as resource preemption, process termination, or rolling back processes to a safe state.

2. Safe State Verification: Before granting a resource request, the system can check if the new state would still be a safe state, i.e., a state where no cycle would form. If granting the request results in a cycle, the request is denied, ensuring that the system remains deadlock-free.

3.

1. FIFO (First In, First Out)

FIFO processes requests in the order they arrive.

Initial head position: 100 Request queue: 55, 58, 39, 18, 90, 160, 150, 38, 184

Seek operations:

1. From 100 to 55: |100−55|=45|100 - 55| = 45|100−55|=45
2. From 55 to 58: |55−58|=3|55 - 58| = 3|55−58|=3
3. From 58 to 39: |58−39|=19|58 - 39| = 19|58−39|=19
4. From 39 to 18: |39−18|=21|39 - 18| = 21|39−18|=21
5. From 18 to 90: |18−90|=72|18 - 90| = 72|18−90|=72
6. From 90 to 160: |90−160|=70|90 - 160| = 70|90−160|=70
7. From 160 to 150: |160−150|=10|160 - 150| = 10|160−150|=10
8. From 150 to 38: |150−38|=112|150 - 38| = 112|150−38|=112
9. From 38 to 184: |38−184|=146|38 - 184| = 146|38−184|=146

Total seek time: 45+3+19+21+72+70+10+112+146=49845 + 3 + 19 + 21 + 72 + 70 + 10 + 112 + 146 = 49845+3+19+21+72+70+10+112+146=498

Average seek time for FIFO: 4989≈55.33\frac{498}{9} \approx 55.339498≈55.33

2. SCAN (Elevator Algorithm)

In SCAN, the disk arm moves towards one end, servicing all requests until it reaches the end, then reverses direction.

Initial head position: 100 Request queue: 55, 58, 39, 18, 90, 160, 150, 38, 184 Direction: Assume the initial direction is towards 0 (decreasing order).

Sort the queue: 18, 38, 39, 55, 58, 90, 150, 160, 184

Seek operations:

1. From 100 to 90: |100−90|=10|100 - 90| = 10|100−90|=10

2. From 90 to 58: ∣90−58∣=32|90 - 58| = 32∣90−58∣=32
3. From 58 to 55: ∣58−55∣=3|58 - 55| = 3∣58−55∣=3
4. From 55 to 39: ∣55−39∣=16|55 - 39| = 16∣55−39∣=16
5. From 39 to 38: ∣39−38∣=1|39 - 38| = 1∣39−38∣=1
6. From 38 to 18: ∣38−18∣=20|38 - 18| = 20∣38−18∣=20
7. Change direction (no seek time incurred).
8. From 18 to 150: ∣18−150∣=132|18 - 150| = 132∣18−150∣=132
9. From 150 to 160: ∣150−160∣=10|150 - 160| = 10∣150−160∣=10
10. From 160 to 184: ∣160−184∣=24|160 - 184| = 24∣160−184∣=24

Total seek time: 10+32+3+16+1+20+132+10+24=24810 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = 24810+32+3+16+1+20+132+10+24=248

Average seek time for SCAN: 2489≈27.56\frac{248}{9} \approx 27.569248≈27.56

3. SSTF (Shortest Seek Time First)

SSTF selects the request with the minimum seek time from the current head position.

Initial head position: 100 Request queue: 55, 58, 39, 18, 90, 160, 150, 38, 184

Seek operations:

1. From 100 to 90: ∣100−90∣=10|100 - 90| = 10∣100−90∣=10
2. From 90 to 55: ∣90−55∣=35|90 - 55| = 35∣90−55∣=35
3. From 55 to 58: ∣55−58∣=3|55 - 58| = 3∣55−58∣=3
4. From 58 to 39: ∣58−39∣=19|58 - 39| = 19∣58−39∣=19
5. From 39 to 38: ∣39−38∣=1|39 - 38| = 1∣39−38∣=1
6. From 38 to 18: ∣38−18∣=20|38 - 18| = 20∣38−18∣=20
7. From 18 to 150: ∣18−150∣=132|18 - 150| = 132∣18−150∣=132
8. From 150 to 160: ∣150−160∣=10|150 - 160| = 10∣150−160∣=10
9. From 160 to 184: ∣160−184∣=24|160 - 184| = 24∣160−184∣=24

Total seek time: 10+35+3+19+1+20+132+10+24=25410 + 35 + 3 + 19 + 1 + 20 + 132 + 10 + 24 = 25410+35+3+19+1+20+132+10+24=254

Average seek time for SSTF: 2549≈28.22\frac{254}{9} \approx 28.229254≈28.22

4.

1. Allocation matrix: Represents the current allocation of resources to each process.
2. Max matrix: Maximum demand of resources for each process.
3. Available vector: Represents currently available resources.
4. Need matrix: Calculated as `Need = Max - Allocation`.

(i) Is the system in a safe state? If yes, mention the safe sequence.

Steps:

1. Calculate the Need matrix.
   Use the formula:
   Need[i][j]=Max[i][j]−Allocation[i][j]Need[i][j] = Max[i][j] - Allocation[i][j]Need[i][j]=Max[i][j]−Allocation[i][j]
2. Check for a safe sequence.
   Follow these steps:
   - Start with the Available vector.
   - Check if the Need of any process can be satisfied with the Available resources.
   - If yes, mark the process as completed and add its Allocation to the Available vector.
   - Repeat until all processes are marked as completed or no process can proceed.

(ii) Can the request `(0, 4, 2, 0)` from process P1 be granted immediately?

Steps:

1. Verify the request against the Need matrix:
   Request[i][j]≤Need[i][j]∀jRequest[i][j] \leq Need[i][j] \quad \forall jRequest[i][j]≤Need[i][j]∀j
2. Verify the request against the Available vector:
   Request[i][j]≤Available[j]∀jRequest[i][j] \leq Available[j] \quad \forall jRequest[i][j]≤Available[j]∀j
3. If both conditions are satisfied, simulate granting the request:
   - Temporarily update the `Available`, `Allocation`, and `Need` matrices.
   - Check if the system remains in a safe state after granting the request.

Let me calculate this for you.

Solution:

(i) Is the system in a safe state?

Yes, the system is in a safe state.
Safe sequence: P0 → P2 → P1 → P3 → P4.

(ii) Can the request (0, 4, 2, 0) from process P1 be granted immediately?

Yes, the request can be granted immediately. After granting the request, the system will remain in a safe state.

4o

You said:

need step by step calculation of finding safe sequence

ChatGPT said:

ChatGPT

Let’s calculate the safe sequence step by step using the Banker’s algorithm. Here's the detailed process:

Step 1: Input Matrices

Allocation Matrix

Allocation=[00121000135406320014]\text{Allocation} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{bmatrix}Allocation=0110000360 1053120424

Max Matrix

Max=[00121750235606520656]\text{Max} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 7 & 5 & 0 \\ 2 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \\ 0 & 6 & 5 & 6 \end{bmatrix}Max=01200073661555520626

Available Vector

Available=[1,5,2,0]\text{Available} = [1, 5, 2, 0]Available=[1,5,2,0]

Need Matrix

We calculate Need as:

Need=Max−Allocation\text{Need} = \text{Max} - \text{Allocation}Need=Max−Allocation Need=[00000750100200200642]\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix}Need=00100070060502400202

Step 2: Iterative Safe Sequence Check

We follow these steps:

1. Start with the Available vector.
2. Find a process whose Need is ≤ Available.
3. Allocate its resources (add its Allocation to Available), mark it as "completed," and move to the next process.

Initial State:

Available=[1,5,2,0]\text{Available} = [1, 5, 2, 0]Available=[1,5,2,0]

No processes are completed yet.

Iteration 1: Find the first process

- Check if any process can be completed:
    - For P0, Need[P0]=[0,0,0,0]≤Available=[1,5,2,0]\text{Need}[P0] = [0, 0, 0, 0] \leq \text{Available} = [1, 5, 2, 0]Need[P0]=[0,0,0,0]≤Available=[1,5,2,0].
      P0 can execute.
- Allocate P0's resources:
  Available=Available+Allocation[P0]=[1,5,2,0]+[0,0,1,2]=[1,5,3,2]\text{Available} = \text{Available} + \text{Allocation}[P0] = [1, 5, 2, 0] + [0, 0, 1, 2] = [1, 5, 3, 2]Available=Available+Allocation[P0]=[1,5,2,0]+[0,0,1,2]=[1,5,3,2]
- Mark P0 as completed.
  Safe sequence so far: P0.

Iteration 2: Find the next process

- Check remaining processes:
    - For P2, Need[P2]=[1,0,0,2]≤Available=[1,5,3,2]\text{Need}[P2] = [1, 0, 0, 2] \leq \text{Available} = [1, 5, 3, 2]Need[P2]=[1,0,0,2]≤Available=[1,5,3,2].
      P2 can execute.
- Allocate P2's resources:
  Available=Available+Allocation[P2]=[1,5,3,2]+[1,3,5,4]=[2,8,8,6]\text{Available} = \text{Available} + \text{Allocation}[P2] = [1, 5, 3, 2] + [1, 3, 5, 4] = [2, 8, 8, 6]Available=Available+Allocation[P2]=[1,5,3,2]+[1,3,5,4]=[2,8,8,6]
- Mark P2 as completed.
  Safe sequence so far: P0 → P2.

Iteration 3: Find the next process

- Check remaining processes:
  - For P1, Need[P1]=[0,7,5,0]≤Available=[2,8,8,6]\text{Need}[P1] = [0, 7, 5, 0] \leq \text{Available} = [2, 8, 8, 6]Need[P1]=[0,7,5,0]≤Available=[2,8,8,6].
    P1 can execute.
- Allocate P1's resources:
  Available=Available+Allocation[P1]=[2,8,8,6]+[1,0,0,0]=[3,8,8,6]\text{Available} = \text{Available} + \text{Allocation}[P1] = [2, 8, 8, 6] + [1, 0, 0, 0] = [3, 8, 8, 6]Available=Available+Allocation[P1]=[2,8,8,6]+[1,0,0,0]=[3,8,8,6]
- Mark P1 as completed.
  Safe sequence so far: P0 → P2 → P1.

Iteration 4: Find the next process

- Check remaining processes:
  - For P3, Need[P3]=[0,0,2,0]≤Available=[3,8,8,6]\text{Need}[P3] = [0, 0, 2, 0] \leq \text{Available} = [3, 8, 8, 6]Need[P3]=[0,0,2,0]≤Available=[3,8,8,6].
    P3 can execute.
- Allocate P3's resources:
  Available=Available+Allocation[P3]=[3,8,8,6]+[0,6,3,2]=[3,14,11,8]\text{Available} = \text{Available} + \text{Allocation}[P3] = [3, 8, 8, 6] + [0, 6, 3, 2] = [3, 14, 11, 8]Available=Available+Allocation[P3]=[3,8,8,6]+[0,6,3,2]=[3,14,11,8]
- Mark P3 as completed.
  Safe sequence so far: P0 → P2 → P1 → P3.

Iteration 5: Find the next process

- Check remaining processes:
  - For P4, Need[P4]=[0,6,4,2]≤Available=[3,14,11,8]\text{Need}[P4] = [0, 6, 4, 2] \leq \text{Available} = [3, 14, 11, 8]Need[P4]=[0,6,4,2]≤Available=[3,14,11,8].
    P4 can execute.
- Allocate P4's resources:
  Available=Available+Allocation[P4]=[3,14,11,8]+[0,0,1,4]=[3,14,12,12]\text{Available} = \text{Available} + \text{Allocation}[P4] = [3, 14, 11, 8] + [0, 0, 1, 4] = [3, 14, 12, 12]Available=Available+Allocation[P4]=[3,14,11,8]+[0,0,1,4]=[3,14,12,12]
- Mark P4 as completed.
  Safe sequence so far: P0 → P2 → P1 → P3 → P4.

Final Safe Sequence:

Safe sequence: P0→P2→P1→P3→P4\text{Safe sequence: } P0 \to P2 \to P1 \to P3 \to P4Safe sequence: P0→P2→P1→P3→P4

5.

Paging is a memory management technique in which the process's virtual address space is divided into fixed-size blocks called pages. Similarly, the physical memory is divided into fixed-size blocks called frames. Each page of the process is mapped to a frame in physical memory.

Paging Works:

1. **Logical Address**: The CPU generates a logical (virtual) address, which consists of a page number and an offset.
2. **Page Table Lookup**: The page number is used to index into the page table to find the frame number.
3. **Physical Address**: The frame number is combined with the offset to form the physical address.
4. **Memory Access**: The physical address is used to access the actual memory location.

Advantages of Paging:

- **Eliminates External Fragmentation**: As both pages and frames are of fixed size, paging eliminates the problem of fitting varying sized memory chunks.
- **Efficient Memory Utilization**: Allows non-contiguous allocation of memory, making more efficient use of physical memory.

Disadvantages of Paging:

- **Internal Fragmentation**: Fixed-size pages can lead to some memory being wasted if the process does not fully use the last page.
- **Overhead of Page Table Management**: Maintaining page tables can add overhead, especially for large address spaces.

Example:

Consider a system with a page size of 4 KB. If a process requires 10 KB of memory, it will be divided into 3 pages (2 full pages and 1 partial page). These pages are mapped to any available frames in the physical memory.

Segmentation

egmentation is a memory management technique in which the process's address space is divided into variable-size segments. Each segment represents a logical unit of the process, such as a function, an array, or a data structure. Each segment has a base address and a length.

Key Concepts:

- **Segment**: A variable-size block of the process's address space, representing a logical unit.
- **Segment Table**: A data structure that keeps track of the segments. Each entry contains the base address and the limit (length) of the segment.
- **Segment Table Entry (STE)**: An entry in the segment table that contains information about a segment, including its base address and length.

egmentation Works:

1. **Logical Address**: The CPU generates a logical address, which consists of a segment number and an offset.
2. **Segment Table Lookup**: The segment number is used to index into the segment table to find the base address and the limit of the segment.
3. **Bounds Checking**: The offset is checked against the limit to ensure it is within the segment's bounds.
4. **Physical Address**: The base address is combined with the offset to form the physical address.
5. **Memory Access**: The physical address is used to access the actual memory location.

Advantages of Segmentation:

- **Logical View**: Segmentation provides a logical view of memory, making it easier to manage and protect logical units such as functions or data structures.
- **Protection and Sharing**: Segments can be protected and shared independently, allowing for finer-grained access control.

Comparison of Paging and Segmentation:

Feature	Paging	Segmentation
Memory Division	Fixed-size pages	Variable-size segments
Address Structure	Page number + offset	Segment number + offset

External Fragmentation	No	Yes
Internal Fragmentation	Yes	No
Logical Division	No logical division	Logical units (functions, arrays, etc.)
Ease of Implementation	Easier to implement	More complex
Protection and Sharing	Coarse-grained	Fine-grained

6.

Single-Level Directory Structure

Definition: In a single-level directory structure, all files are contained within one directory. This means every user shares the same directory space, and all files are stored in a single directory.

Characteristics:

- **Simplicity:** The structure is straightforward and easy to understand and implement.
- **Naming Conflicts:** Since all files are in one directory, each file must have a unique name, leading to potential naming conflicts.
- **Security and Privacy:** With all files in one directory, it is challenging to manage security and privacy, as every user has access to the same directory.

Advantages:

- Simple to implement and navigate.
- Easy to back up and restore.

Disadvantages:

- Scalability issues as the number of files increases.
- Naming conflicts as each file must have a unique name.
- Lack of organization and categorization.

Two-Level Directory Structure

Definition: In a two-level directory structure, there is a separate directory for each user. Each user has their own directory, and within that directory, they can create and manage their own files.

Characteristics:

- **User Isolation:** Each user has a separate directory, providing better isolation and security.
- **Unique User Files:** Users can create files with the same name in their directories without conflicts.
- **Improved Organization:** Users can organize their files in their own directory without affecting others.

Advantages:

- Better organization and categorization of files.
- Improved security and privacy as users have their own directories.
- Easier to manage and search for files within a user's directory.

Disadvantages:

- More complex than a single-level directory.
- Requires more storage space to maintain separate directories for each user.

Various File Operations with Examples

File operations are actions that can be performed on files in a file system. Here are some common file operations with examples:

1. **Create:**
    - **Description:** Creating a new file in the file system.

Example:
python
Copy code

```
with open('example.txt', 'w') as file:

    file.write('Hello, World!')
```

   - This code creates a new file named `example.txt` and writes "Hello, World!" to it.
2. **Open:**
    - **Description:** Opening an existing file for reading or writing.

Example:
python

Copy code

```
file = open('example.txt', 'r')
```

   - This code opens the file `example.txt` for reading.

3. **Read:**
   - **Description:** Reading data from an open file.

Example:
python
Copy code

```
with open('example.txt', 'r') as file:

    content = file.read()

    print(content)
```

   - This code reads the contents of `example.txt` and prints it.

4. **Write:**
   - **Description:** Writing data to an open file.

Example:
python
Copy code

```
with open('example.txt', 'w') as file:

    file.write('New content')
```

   - This code writes "New content" to `example.txt`, overwriting any existing content.

5. **Append:**
   - **Description:** Adding data to the end of an existing file.

Example:
python
Copy code

```
with open('example.txt', 'a') as file:

    file.write('Additional content')
```

   - This code appends "Additional content" to the end of `example.txt`.

6. **Close:**
   - **Description:** Closing an open file to free up system resources.

Example:
python

Copy code

```
file = open('example.txt', 'r')

content = file.read()

file.close()
```

  - This code reads the contents of `example.txt` and then closes the file.

7. **Delete:**
   - **Description:** Removing a file from the file system.

Example:
python
Copy code

```
import os

os.remove('example.txt')
```

  - This code deletes the file `example.txt`.

8. **Rename:**
   - **Description:** Changing the name of a file.

Example:
python
Copy code

```
import os

os.rename('example.txt', 'new_example.txt')
```

  - This code renames `example.txt` to `new_example.txt`.

9. **Copy:**
   - **Description:** Creating a duplicate of a file.

Example:
python
Copy code

```
import shutil

shutil.copy('example.txt', 'copy_of_example.txt')
```

  - This code copies the contents of `example.txt` to `copy_of_example.txt`