QUESTION # 01

Let's analyze this step by step for each scheduling algorithm and calculate the required metrics. I'll explain each concept thoroughly so that you'll understand the process and computations.

Key Terms

1. **Burst Time (BT):** Time required by a process for execution on the CPU.
2. **Turnaround Time (TAT):** The total time a process spends in the system, from arrival to completion.
   TAT=Completion Time (CT)−Arrival Time (AT)\text{TAT} = \text{Completion Time (CT)} - \text{Arrival Time (AT)}
3. **Waiting Time (WT):** Time spent by a process waiting in the ready queue.
   WT=TAT−Burst Time (BT)\text{WT} = \text{TAT} - \text{Burst Time (BT)}
4. **Average Waiting Time (AWT):** The average of waiting times for all processes.

---

1. First-Come, First-Served (FCFS)

Processes are executed in the order of arrival. Since all processes arrive at t=0t = 0, the order of execution is P1→P2→P3→P4→P5P1 \rightarrow P2 \rightarrow P3 \rightarrow P4 \rightarrow P5.

Gantt Chart

```
| P1 | P2 | P3     | P4   | P5    |
0    2   3      11    15     20
```

Calculations

Process	**Burst Time (BT)**	**Completion Time (CT)**	**Turnaround Time (TAT)**	**Waiting Time (WT)**
P1	2	2	2 - 0 = 2	2 - 2 = 0
P2	1	3	3 - 0 = 3	3 - 1 = 2
P3	8	11	11 - 0 = 11	11 - 8 = 3
P4	4	15	15 - 0 = 15	15 - 4 = 11
P5	5	20	20 - 0 = 20	20 - 5 = 15

Average Waiting Time (AWT):

AWT=0+2+3+11+155=6.2 ms\text{AWT} = \frac{0 + 2 + 3 + 11 + 15}{5} = 6.2 \, \text{ms}

---

2. Shortest Job First (SJF)

Processes are executed in ascending order of burst time.

Order of Execution

P2→P1→P4→P5→P3P2 \rightarrow P1 \rightarrow P4 \rightarrow P5 \rightarrow P3

Gantt Chart

```
| P2 | P1 | P4   | P5    | P3       |
0    1    3    7      12       20
```

Calculations

Process	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	2	3	3 - 0 = 3	3 - 2 = 1
P2	1	1	1 - 0 = 1	1 - 1 = 0
P3	8	20	20 - 0 = 20	20 - 8 = 12
P4	4	7	7 - 0 = 7	7 - 4 = 3
P5	5	12	12 - 0 = 12	12 - 5 = 7

Average Waiting Time (AWT):

AWT=1+0+12+3+75=4.6 ms\text{AWT} = \frac{1 + 0 + 12 + 3 + 7}{5} = 4.6 \, \text{ms}

---

3. Non-Preemptive Priority

Processes are executed in ascending order of priority (higher priority = lower priority number).

Order of Execution

P2→P1→P4→P5→P3P2 \rightarrow P1 \rightarrow P4 \rightarrow P5 \rightarrow P3

Gantt Chart

```
| P2 | P1 | P4   | P5    | P3       |
0    1    3    7      12       20
```

Calculations

Process	Burst Time (BT)	Priority	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	2	2	3	3 - 0 = 3	3 - 2 = 1

Process	Burst Time (BT)	Priority	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P2	1	1	1	1 - 0 = 1	1 - 1 = 0
P3	8	4	20	20 - 0 = 20	20 - 8 = 12
P4	4	2	7	7 - 0 = 7	7 - 4 = 3
P5	5	3	12	12 - 0 = 12	12 - 5 = 7

Average Waiting Time (AWT):

AWT=1+0+12+3+75=4.6 ms\text{AWT} = \frac{1 + 0 + 12 + 3 + 7}{5} = 4.6 \, \text{ms}

---

4. Round Robin (RR)

Time quantum = 2 ms. Each process gets a maximum of 2 ms per turn.

Execution Steps

1. P1P1: Runs for 2 ms. Remaining BT = 0.
2. P2P2: Runs for 1 ms. Remaining BT = 0.
3. P3P3: Runs for 2 ms. Remaining BT = 6.
4. P4P4: Runs for 2 ms. Remaining BT = 2.
5. P5P5: Runs for 2 ms. Remaining BT = 3.
6. Repeat until all processes are complete.

Gantt Chart

```
| P1 | P2 | P3 | P4 | P5 | P3 | P4 | P5 | P3 |
0   2   3   5   7   9   11   13   15   20
```

Calculations

Process	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	2	2	2 - 0 = 2	2 - 2 = 0
P2	1	3	3 - 0 = 3	3 - 1 = 2
P3	8	20	20 - 0 = 20	20 - 8 = 12
P4	4	11	11 - 0 = 11	11 - 4 = 7

Process	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P5	5	15	15 - 0 = 15	15 - 5 = 10

Average Waiting Time (AWT):

AWT=0+2+12+7+105=6.2 ms\text{AWT} = \frac{0 + 2 + 12 + 7 + 10}{5} = 6.2 \, \text{ms}

Summary

Scheduling Algorithm	Average Waiting Time (AWT)
First-Come, First-Served (FCFS)	6.2 ms
Shortest Job First (SJF)	4.6 ms
Non-Preemptive Priority	4.6 ms
Round Robin (Quantum = 2)	6.2 ms

Let me know if you'd like a deeper explanation of any algorithm! 😊

QUESTION # 02

Peterson's Solution for Mutual Exclusion

Peterson's solution is a software-based protocol for achieving mutual exclusion between two processes that need to access a critical section. It uses two shared variables:

1. **flag[]**: An array of two Boolean variables to indicate if a process wants to enter the critical section.
2. **turn**: An integer variable to indicate which process gets preference.

---

a) Pseudocode Implementation

```
// Shared variables

boolean flag[2] = {false, false}; // Initially, neither process wants to enter

int turn;                 // Indicates whose turn it is
```

```
// Code for Process P0
do {
   // Entry Section
   flag[0] = true;            // Indicate P0 wants to enter
   turn = 1;                  // Give preference to P1
   while (flag[1] && turn == 1); // Wait if P1 wants to enter and it's P1's turn

   // Critical Section
   // (Code that must be executed by only one process at a time)

   // Exit Section
   flag[0] = false;           // Indicate P0 is leaving the critical section

   // Remainder Section
   // (Code that can be executed without restriction)
} while (true);

// Code for Process P1
do {
   // Entry Section
   flag[1] = true;            // Indicate P1 wants to enter
   turn = 0;                  // Give preference to P0
   while (flag[0] && turn == 0); // Wait if P0 wants to enter and it's P0's turn

   // Critical Section
   // (Code that must be executed by only one process at a time)

   // Exit Section
   flag[1] = false;           // Indicate P1 is leaving the critical section
```

```
    // Remainder Section
    // (Code that can be executed without restriction)
} while (true);
```

---

b) How Peterson's Solution Ensures Mutual Exclusion

Peterson's solution ensures mutual exclusion through the following properties:

1. **Mutual Exclusion**:
    - Each process sets its flag to true before entering the critical section, signaling its intent to enter.
    - A process only enters the critical section if:
        - The other process does not want to enter (flag[other_process] == false), or
        - It is the current process's turn (turn == self).
    - This guarantees that both processes cannot be in the critical section simultaneously.
2. **Progress**:
    - If no process is in the critical section, and one wants to enter, it can proceed without unnecessary delay.
3. **Bounded Waiting**:
    - No process is indefinitely delayed because the turn variable alternates priority between the processes.

---

c) Potential Limitations or Drawbacks of Peterson's Solution

1. **Limited to Two Processes**:
    - The solution is designed for exactly two processes. Extending it to more processes is possible but complex and inefficient.
2. **Busy Waiting**:
    - The while loop causes busy waiting, where a process continuously checks the condition until it can enter the critical section. This wastes CPU cycles.
3. **Relies on Atomic Operations**:

- The solution assumes that operations like setting flag and turn are atomic. In modern multi-core systems, such assumptions may not hold due to caching and memory reordering.

4. **Not Suitable for Modern Architectures**:
    - On modern systems, Peterson's solution may fail due to optimizations like out-of-order execution and weak memory consistency models unless memory barriers or special instructions are used.

If you'd like a detailed walkthrough of how processes alternate using the flag and turn variables, let me know! 😊

QUESTION # 03

To determine whether each state is safe or unsafe using the **Banker's Algorithm**, we follow these steps:

1. **Calculate the Need Matrix**: Need=Max Need−Allocation\text{Need} = \text{Max Need} - \text{Allocation}.
2. **Check Feasibility**: Compare each thread's **Need** to the available resources. A thread can execute if its **Need** is less than or equal to the current available resources.
3. **Simulate Execution**: If a thread executes, its **Allocation** is added back to the available resources. Continue until all threads execute, or no progress can be made.

Step 1: Need Matrix

	Allocation	Max Need	Need
T0	1 2 0 2	4 3 1 6	3 1 1 4
T1	0 1 1 2	2 4 2 4	2 3 1 2
T2	1 2 4 0	3 6 5 1	2 4 1 1
T3	1 2 0 1	2 6 2 3	1 4 2 2
T4	1 0 0 1	3 1 1 2	2 1 1 1

Step 2: Check Each Case

a. Available = (2, 2, 2, 3)

Simulation Steps:

1. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4) with Available=(2,2,2,3)\text{Available} = (2, 2, 2, 3): Need[T0]>Available\text{Need}[T0] > \text{Available}. T0 cannot execute.
2. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]>Available\text{Need}[T1] > \text{Available}. T1 cannot execute.
3. Compare Need[T2]=(2,4,1,1)\text{Need}[T2] = (2, 4, 1, 1): Need[T2]≤Available\text{Need}[T2] \leq \text{Available}. T2 executes.

New Available = (2+1,2+2,2+4,3+0)=(3,4,6,3)(2 + 1, 2 + 2, 2 + 4, 3 + 0) = (3, 4, 6, 3).

4. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4): Need[T0]≤Available\text{Need}[T0] \leq \text{Available}. T0 executes.

New Available = (3+1,4+2,6+0,3+2)=(4,6,6,5)(3 + 1, 4 + 2, 6 + 0, 3 + 2) = (4, 6, 6, 5).

5. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]≤Available\text{Need}[T1] \leq \text{Available}. T1 executes.

New Available = (4+0,6+1,6+1,5+2)=(4,7,7,7)(4 + 0, 6 + 1, 6 + 1, 5 + 2) = (4, 7, 7, 7).

6. Compare Need[T3]=(1,4,2,2)\text{Need}[T3] = (1, 4, 2, 2): Need[T3]≤Available\text{Need}[T3] \leq \text{Available}. T3 executes.

New Available = (4+1,7+2,7+0,7+1)=(5,9,7,8)(4 + 1, 7 + 2, 7 + 0, 7 + 1) = (5, 9, 7, 8).

7. Compare Need[T4]=(2,1,1,1)\text{Need}[T4] = (2, 1, 1, 1): Need[T4]≤Available\text{Need}[T4] \leq \text{Available}. T4 executes.

Safe Sequence: T2→T0→T1→T3→T4T2 \rightarrow T0 \rightarrow T1 \rightarrow T3 \rightarrow T4
State: **Safe**

---

b. Available = (4, 4, 1, 1)

Simulation Steps:

1. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4) with Available=(4,4,1,1)\text{Available} = (4, 4, 1, 1): Need[T0]>Available\text{Need}[T0] > \text{Available}. T0 cannot execute.
2. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]>Available\text{Need}[T1] > \text{Available}. T1 cannot execute.

3. Compare Need[T2]=(2,4,1,1)\text{Need}[T2] = (2, 4, 1, 1): Need[T2]≤Available\text{Need}[T2] \leq \text{Available}. T2 executes.

New Available = (4+1,4+2,1+4,1+0)=(5,6,5,1)(4 + 1, 4 + 2, 1 + 4, 1 + 0) = (5, 6, 5, 1).

4. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4): Need[T0]≤Available\text{Need}[T0] \leq \text{Available}. T0 executes.

New Available = (5+1,6+2,5+0,1+2)=(6,8,5,3)(5 + 1, 6 + 2, 5 + 0, 1 + 2) = (6, 8, 5, 3).

5. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]≤Available\text{Need}[T1] \leq \text{Available}. T1 executes.

New Available = (6+0,8+1,5+1,3+2)=(6,9,6,5)(6 + 0, 8 + 1, 5 + 1, 3 + 2) = (6, 9, 6, 5).

6. Compare Need[T3]=(1,4,2,2)\text{Need}[T3] = (1, 4, 2, 2): Need[T3]≤Available\text{Need}[T3] \leq \text{Available}. T3 executes.

New Available = (6+1,9+2,6+0,5+1)=(7,11,6,6)(6 + 1, 9 + 2, 6 + 0, 5 + 1) = (7, 11, 6, 6).

7. Compare Need[T4]=(2,1,1,1)\text{Need}[T4] = (2, 1, 1, 1): Need[T4]≤Available\text{Need}[T4] \leq \text{Available}. T4 executes.

Safe Sequence: T2→T0→T1→T3→T4T2 \rightarrow T0 \rightarrow T1 \rightarrow T3 \rightarrow T4
State: **Safe**

---

c. Available = (3, 0, 1, 4)

Simulation Steps:

1. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4) with Available=(3,0,1,4)\text{Available} = (3, 0, 1, 4): Need[T0]>Available\text{Need}[T0] > \text{Available}. T0 cannot execute.

2. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]>Available\text{Need}[T1] > \text{Available}. T1 cannot execute.

3. Compare Need[T2]=(2,4,1,1)\text{Need}[T2] = (2, 4, 1, 1): Need[T2]>Available\text{Need}[T2] > \text{Available}. T2 cannot execute.

4. Compare Need[T3]=(1,4,2,2)\text{Need}[T3] = (1, 4, 2, 2): Need[T3]>Available\text{Need}[T3] > \text{Available}. T3 cannot execute.

5. Compare Need[T4]=(2,1,1,1)\text{Need}[T4] = (2, 1, 1, 1): Need[T4]>Available\text{Need}[T4] > \text{Available}. T4 cannot execute.

State: **Unsafe** (No progress is possible)

---

d. Available = (1, 5, 2, 2)

Simulation Steps:

1. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4) with Available=(1,5,2,2)\text{Available} = (1, 5, 2, 2): Need[T0]>Available\text{Need}[T0] > \text{Available}. T0 cannot execute.
2. Compare Need[T1]=(2,3,1,2)\text{Need}[T1] = (2, 3, 1, 2): Need[T1]≤Available\text{Need}[T1] \leq \text{Available}. T1 executes.

New Available = (1+0,5+1,2+1,2+2)=(1,6,3,4)(1 + 0, 5 + 1, 2 + 1, 2 + 2) = (1, 6, 3, 4).

3. Compare Need[T0]=(3,1,1,4)\text{Need}[T0] = (3, 1, 1, 4): Need[T0]≤Available\text{Need}[T0] \leq \text{Available}. T0 executes.

New Available = (1+1,6+2,3+0,4+2)=(2,8,3,6)(1 + 1, 6 + 2, 3 + 0, 4 + 2) = (2, 8, 3, 6).

4. Compare Need[T3]=(1,4,2,2)\text{Need}[T3] = (1, 4, 2, 2):

QUESTION # 04

a) Logical and Physical Address Bit Calculation

Logical Address

- **Number of Pages (Logical Address Space)**: 256256
  To represent 256 pages, log⁡2(256)=8\log_2(256) = 8 bits are required for the **page number**.
- **Page Size**: 4 KB=212 bytes4 \text{ KB} = 2^{12} \text{ bytes}
  To represent an offset within a page, log⁡2(4096)=12\log_2(4096) = 12 bits are required for the **page offset**.

Thus, the **logical address** consists of:

1. 88 bits for the page number
2. 1212 bits for the page offset

Total Logical Address Bits: 8+12=208 + 12 = 20 bits.

Physical Address

- **Number of Frames**: 6464
  To represent 64 frames, log⁡2(64)=6\log_2(64) = 6 bits are required for the **frame number**.
- **Page Size**: 4 KB=212 bytes4 \text{ KB} = 2^{12} \text{ bytes}
  As the physical frame size equals the logical page size, 1212 bits are required for the **frame offset**.

Thus, the **physical address** consists of:

1. 66 bits for the frame number
2. 1212 bits for the frame offset

Total Physical Address Bits: 6+12=186 + 12 = 18 bits.

---

b) Cause of Thrashing and How It is Detected and Eliminated

Cause of Thrashing

Thrashing occurs when a system spends more time **paging** than executing processes. This happens when:

1. **Insufficient Memory**: Processes require more memory than available physical memory, leading to frequent page replacements.
2. **High Degree of Multiprogramming**: Too many processes in memory increase contention for frames.
3. **Poor Locality of Reference**: Processes access pages scattered across the memory instead of accessing pages within a localized area.

Detection of Thrashing

The system detects thrashing by monitoring:

1. **High Page-Fault Rate**: A high frequency of page faults indicates that processes are not able to keep their working sets in memory.
2. **CPU Utilization Drops**: As the page-fault rate increases, CPU utilization drops because processes spend more time waiting for pages to be loaded.

Elimination of Thrashing

1. **Working Set Model**:
   - Maintain a set of pages actively used by a process (its **working set**) in memory.
   - Suspend processes whose working sets cannot fit into the available memory.
2. **Page-Fault Frequency (PFF) Control**:
   - Monitor the page-fault rate of each process.
   - If the rate is too high, allocate more frames to the process. If it is too low, consider removing some frames.

3. **Adjust Degree of Multiprogramming**:
    - Reduce the number of active processes in memory to free up frames for processes with large working sets.
4. **Swapping**:
    - Swap out entire processes to disk when their working sets exceed available memory, allowing other processes to execute efficiently.

Would you like an example or deeper explanation of any of these points?

QUESTION # 05

Purpose of Directory Structures

Directories are used in file systems to organize and manage files efficiently. Below are the purposes of different directory structures along with their diagrams:

---

a) Tree-Structured Directories

Purpose:

- Organize files hierarchically with a root directory at the top.
- Allow directories to contain subdirectories and files.
- Simplify file management through a clear, parent-child relationship.

Features:

1. Each file or directory has a single parent (except the root).
2. Supports operations like searching, creation, and deletion efficiently.
3. Prevents cycles, ensuring straightforward navigation.

Diagram:

```
Root
 |
 +-- Home
 |   +-- User1
 |   |   +-- file1.txt
 |   |   +-- file2.doc

 |   +-- User2

 |

 +-- etc

 |   +-- config.conf

 |

 +-- var

     +-- logs
```

---

b) Acyclic-Graph Directories

Purpose:

- Extend tree-structured directories by allowing shared files or directories.
- Enable efficient use of storage by allowing multiple paths to the same file.
- Useful in collaborative or shared environments where files are used by multiple users or programs.

Features:

1. A directory or file can have multiple parents.
2. No cycles are allowed, ensuring there are no infinite loops during traversal.
3. Typically uses reference counts to track how many directories reference a file.

Diagram:

```
Root

|

+-- Home

|   +-- User1

|   |   +-- file1.txt

|   |

|   +-- Shared

|

+-- Work
```

```
  +-- Shared --> (points to the same directory as Home/Shared)
```

Here, the Shared directory is accessible from both Home and Work.

---

c) General Graph Directories

Purpose:

- Allow full flexibility by enabling cycles in the directory structure.
- Support complex relationships and multiple paths to a file or directory.
- Can be challenging to manage due to potential infinite loops during traversal.

Features:

1. Directories can point to each other, creating cycles.
2. Traversal requires mechanisms to detect and handle loops.
3. Often used in advanced or specialized file systems.

Diagram:

```
Root
|
+-- Home
|   +-- User1
|   |   +-- file1.txt
|   |       ^
|   |       |
|   +----------+
|
+-- Work
    +-- Projects
        +-- file1.txt --> (points back to Home/User1/file1.txt)
```

In this structure, file1.txt is accessible from multiple paths, but cycles must be carefully managed.

Would you like detailed examples or real-world applications for any of these directory structures?