# SOME IMPORTANT QUESTION FOR LP-1

FCFS, SJF, Priority, and Round Robin are CPU scheduling algorithms used in operating systems to manage how processes are assigned to the CPU. Each has a different approach to deciding the order in which processes are executed. Here's an overview of each:

**FCFS (First-Come, First-Served):**

Type: **Non-preemptive scheduling algorithm.**

**Description**: Processes are scheduled in the order they arrive. The first process to arrive is the first one to be executed until completion, followed by the next.

**Advantages**: Simple to implement.

**Disadvantages**: Can lead to the convoy effect, where short processes wait for a long process to finish, increasing average waiting time.

**SJF (Shortest Job First):**

**Type**: Can be preemptive or non-preemptive.

**Description**: The process with the shortest burst time (execution time) is executed next. If preemptive (also known as Shortest Remaining Time First), a new process with a shorter burst time can interrupt a currently running process.

**Advantages**: Minimizes average waiting time; efficient for batch systems.

**Disadvantages**: Requires knowledge of each process's burst time, which is not always feasible. Can lead to starvation of longer processes if shorter processes keep arriving.

**Priority Scheduling:**

**Type:** Can be preemptive or non-preemptive.

**Description:** Each process is assigned a priority, and the process with the highest priority (typically, lower numbers indicate higher priority) is executed first. In preemptive priority scheduling, a newly arrived process with a higher priority can interrupt the current process.

**Advantages:** Provides control over process execution order, which can be useful in time-sensitive applications.

**Disadvantages:** Can lead to starvation of lower-priority processes, though aging can be used to prevent this by gradually increasing the priority of waiting processes.

**Round Robin (RR):**

**Type:** Preemptive scheduling algorithm.

**Description:** Each process is assigned a fixed time slice (or time quantum) and processes are executed in a cyclic order. After a process uses up its time slice, it is placed at the end of the queue, allowing the next process to execute.

**Advantages**: Fairly distributes CPU time among processes and is well-suited for time-sharing systems.

**Disadvantages:** Choosing an appropriate time quantum is crucial. If it's too short, there may be excessive context switching; if it's too long, it behaves like FCFS.

**Summary Table**

| Algorithm | Type | Key Characteristics |
|---|---|---|
| FCFS | Non-preemptive | Processes served in arrival order |
| SJF | Preemptive / Non-preemptive | Shortest burst time first |
| Priority | Preemptive / Non-preemptive | Highest priority first |
| Round Robin | Preemptive | Equal time slices in cyclic order |

These algorithms balance response time, throughput, and waiting time depending on the requirements of the operating system and the nature of the processes being managed.

**Preemptive Scheduling:** A scheduling approach where a running process can be interrupted and moved to a waiting state to allow another, often higher-priority, process to execute. This allows the system to respond quickly to high-priority tasks but requires frequent context switching.

**Non-Preemptive Scheduling**: A scheduling approach where, once a process starts executing on the CPU, it runs to completion or until it voluntarily releases the CPU (e.g., for I/O). This approach is simpler and involves less context switching but can lead to delays for other tasks.

FIFO, LRU, and Optimal are page replacement algorithms used in operating systems to manage how pages are stored in memory when physical memory is limited. These algorithms help decide which page to remove from memory to make space for a new page when memory is full. Here's a quick overview:

**FIFO (First-In, First-Out):**

**Description:** The page that entered memory first is the first to be replaced. Pages are removed in the order they were loaded, regardless of usage.

**Advantages:** Simple to implement.

**Disadvantages:** May remove frequently used pages, leading to poor performance in some cases.

**LRU (Least Recently Used):**

**Description:** The page that has not been used for the longest period is replaced. It assumes that pages used recently will likely be needed again soon.

**Advantages:** Reduces the likelihood of removing frequently accessed pages.

**Disadvantages:** Requires tracking the usage history of pages, which can add complexity and overhead.

**Optimal Page Replacement:**

**Description:** Replaces the page that will not be used for the longest period in the future. This is the theoretical best approach, as it results in the fewest page faults.

**Advantages:** Minimizes page faults.

**Disadvantages**: Requires future knowledge of page references, so it is mainly used for theoretical comparison rather than practical implementation.

**Memory Allocation:**

Memory allocation is the process by which an operating system assigns specific blocks of memory to various programs, processes, or data structures to enable efficient execution. It is a fundamental aspect of computer system operation, ensuring that each process has the necessary memory space to run and store its data.

Memory allocation can be broadly classified into two types:

**Static Allocation:**

Memory is allocated at compile time and remains fixed for the duration of the program's execution.

Typically used for global variables and static data that do not change in size.

**Dynamic Allocation:**

Memory is allocated at runtime as needed, allowing flexibility for programs to request and release memory during execution.

Examples include allocating memory for dynamically-sized data structures like arrays, lists, and queues.

**pass 2 assembler::::**

Here are some potential oral questions with their corresponding answers regarding the Pass-Two Assembler practical:

1. What is the role of Pass-Two in the assembler process?

**Answer:** Pass-Two's main purpose is to generate the object code from the intermediate code produced in Pass-One. In Pass-One, we resolve labels and symbols to memory addresses. In Pass-Two, these

symbols are replaced by their respective memory addresses, and the actual machine-readable code (object code) is generated based on this information.

2. Can you explain how the symbol table is used in this assembler?

**Answer:** The symbol table stores symbols (such as variable names or labels) along with their corresponding memory addresses. During Pass-Two, the assembler looks up these symbols to resolve the operands in the intermediate code to their correct addresses. If a symbol is found in the table, its address is used; if not, an error occurs.

3. What is the purpose of the literal table, and how is it different from the symbol table?

**Answer:** The literal table stores constants (literal values) that appear in the code, such as =5 or =A. These are not labels but specific values that need to be resolved to memory addresses. The symbol table, on the other hand, stores variable names or labels. During Pass-Two, literals are resolved just like symbols, but they are handled separately to distinguish them from user-defined labels.

4. How does the assembler handle opcodes during Pass-Two?

**Answer:** The OpCodeTable maps the mnemonics (like LOAD, ADD, STORE, etc.) to their corresponding numeric opcodes (e.g., 01 for LOAD, 02 for STORE). During Pass-Two, the assembler looks up each opcode from the intermediate code and retrieves the corresponding opcode in numeric form, which is then included in the object code.

**pass 1 assembler:::**

Here are oral questions along with their corresponding answers for the provided Pass-One Assembler code:

1. What is the purpose of the PassOneAssembler class?

**Answer:** The PassOneAssembler class simulates the first pass of an assembler. It processes assembly code to resolve labels, mnemonics, and operands, generating the symbol table, literal table, and intermediate code. In this pass, the assembler doesn't generate the object code but sets up the necessary tables for the second pass.

2. What is the role of the location_counter in this assembler?

**Answer:** The location_counter keeps track of the current memory address where the next instruction will be stored. It starts at 0 (or the value specified by the START directive) and is incremented as each instruction is processed. This helps in assigning addresses to the symbols during the first pass.

3. What is the symbol table and how does it work?

**Answer:** The symbol table is a data structure that stores labels (such as variable names) along with their corresponding memory addresses. During Pass-One, whenever a label is encountered in the assembly code, it is added to the symbol table with its current memory address. The table helps resolve labels to actual memory locations in the second pass.

4. How does the assembler handle duplicate labels?

**Answer:** The assembler checks for duplicate labels when a label is encountered. If the label already exists in the symbol table, an error is raised, ensuring that there are no multiple definitions of the same label. This prevents ambiguity when generating the object code in the second pass.

5. What is the purpose of the opcode table?

**Answer:** The opcode table stores the mapping between mnemonics (assembly language instructions like LOAD, ADD, STORE) and their corresponding machine opcodes (numeric representations). This table is used in Pass-One to verify that the given opcode is valid, and its corresponding machine code can be used in the second pass

In memory management, First Fit, Best Fit, Worst Fit, and Next Fit are common memory allocation strategies used to place processes or data blocks in memory. Here's how each one works:

**First Fit:**

The memory manager scans from the beginning and places the incoming process or data in the first available memory block that is large enough to accommodate it.

This is generally faster since it stops searching as soon as a suitable block is found.

**Example:** If you have blocks of sizes [10, 20, 30, 40] and need to allocate a process of size 25, it will be placed in the 30-sized block.

**Best Fit:**

The memory manager searches the entire list of available memory blocks and allocates the smallest block that is large enough to hold the process or data.

This helps to reduce wasted space (internal fragmentation), as it uses the smallest adequate block.

**Example:** If the blocks are [10, 20, 30, 40] and the process size is 25, it will go into the 30-sized block because it's the closest fit.

**Worst Fit:**

The memory manager places the process in the largest available block of memory.

This approach aims to leave bigger free spaces available for future processes, reducing the number of unusable small fragments.

**Example:** With blocks [10, 20, 30, 40] and a process of size 25, the process would go into the 40-sized block, as it's the largest available space.

**Next Fit:**

Similar to First Fit, but instead of starting the search from the beginning each time, the search begins from the point where the last allocation occurred.

This can improve performance slightly in some cases by reducing repetitive scanning of memory blocks at the beginning.

**Example:** After allocating a process to a block, the next search begins just after that block, continuing cyclically if it reaches the end of the list.

Each strategy has its pros and cons, and the choice depends on the specific needs of the system in terms of memory utilization and efficiency.