B. TECH. (CSE)
(FIFTH SEMES'TER) END SEMESTER
EXAMINATION, Jan.; 2022
OPERATING SYSTEM

- **Difference Between Batch Processing System and Real-Time Processing System**

Batch processing systems and real-time processing systems are two distinct approaches to data processing, each with its own features and use cases. Here’s a detailed comparison between them:

Feature	Batch Processing System	Real-Time Processing System
Definition	Batch processing refers to executing a series of jobs or tasks without user intervention, in "batches," at scheduled times.	Real-time processing involves continuous data processing where the system responds to inputs or events immediately.
Processing Time	Tasks are processed in groups at predefined intervals, meaning processing happens after a delay.	Tasks are processed instantly as soon as the input is received, with minimal latency.
Examples	Payroll processing, billing systems, bank reconciliation, and end-of-day reports.	Air traffic control, online transaction systems, stock trading, autonomous vehicles, and live video streaming.

Feature	Batch Processing System	Real-Time Processing System
User Interaction	There is minimal or no user interaction during processing, and the user often receives the output after the processing is complete.	Users can interact with the system in real-time, and the system provides immediate feedback or response to inputs.
Data Volume	Batch systems typically handle large volumes of data and process it at scheduled times, often during off-peak hours.	Real-time systems handle smaller chunks of data that require immediate processing as events occur.
Delay	There is often a delay in processing the data due to the nature of batch jobs.	There is little to no delay; the processing is continuous and immediate.
Complexity	Generally, batch systems are less complex in terms of processing logic but may require large data storage and management.	Real-time systems are more complex due to the need for continuous monitoring, event handling, and instant responses.
Reliability	Batch systems are more reliable for non-time-critical tasks since they operate in controlled, predictable environments.	Real-time systems need to be highly reliable as failures can have significant, immediate impacts (e.g., in medical systems).

Feature	Batch Processing System	Real-Time Processing System
Resource Usage	Batch systems usually require significant resources for large-scale data processing but use them intermittently.	Real-time systems require consistent and continuous resource utilization to ensure responsiveness at all times.
System Design	Designed to optimize for throughput (i.e., processing large volumes of data).	Designed to optimize for latency, ensuring immediate response to incoming data or events.
Use Case Suitability	Best suited for tasks where immediacy is not critical, and where the data can be collected, processed, and stored in batches.	Best suited for systems that require immediate action or response, especially in dynamic and time-sensitive environments.

- **2. Inconveniences of Using a Computer Without an Operating System**

1. **No User Interface**
   Without an operating system (OS), there is no graphical or command-line interface, making it extremely difficult to interact with the computer. Users cannot easily open files, run programs, or perform basic tasks.
2. **No Resource Management**
   The OS manages system resources like CPU, memory, and storage. Without it, the system cannot allocate these resources efficiently, leading to performance issues, system crashes, and an inability to multitask.
3. **No File Management**
   The OS organizes files into directories, allowing users to store, retrieve,

and manage data. Without an OS, files are unstructured, making data storage chaotic and access cumbersome.

4. Inability to Run Applications
   Applications require an OS to provide necessary services and hardware interaction. Without an OS, running software becomes impossible, as there's no environment to execute programs or provide essential libraries.
5. No Peripheral Support and Security
   The OS manages peripherals (keyboard, printer, etc.) and provides security features like user authentication and virus protection. Without an OS, users face difficulties in connecting and using devices, and the system becomes vulnerable to security threats.

Conclusion:

A computer without an operating system lacks essential functionality, making it inefficient and impractical for regular use. The OS is crucial for managing resources, running applications, and ensuring system security and usability.

- 3. Purpose of Command Interpreter and Its Separation from the Kernel

The command interpreter, also known as the shell, is a program that enables users to interact with the operating system by processing and executing commands. Its main purposes are:

1. Command Execution: It interprets user commands and translates them into actions that the operating system can execute, such as running programs or managing files.
2. User Interface: It provides a way for users to interact with the OS, either through a command-line interface (CLI) or a graphical shell.
3. Scripting and Automation: It allows users to create scripts to automate repetitive tasks, making system management more efficient.

Why It's Separate from the Kernel:

The command interpreter is typically separate from the kernel for the following reasons:

- Separation of Concerns: The kernel handles core system functions like memory management, process control, and hardware interaction. The

command interpreter is focused on user interaction and command processing.

- **Flexibility: By being separate, users can choose from different shells without modifying the kernel, allowing for a customizable user experience.**
- **Security: Separating the shell from the kernel reduces the risk of compromising system stability. If the shell encounters an issue, it does not affect the core system operations.**

Conclusion:

The command interpreter provides an essential interface for user commands and system interaction, while its separation from the kernel ensures flexibility, security, and easier system maintenance

Q In a process, each thread operates with its own execution flow but shares many resources. However, there are certain things that are not shared by each thread inside a process:

1. **Thread Stack: Each thread has its own stack, which stores local variables, function call information, and return addresses. The stack is unique to each thread, as the function calls and local variables may differ between threads.**
2. **Thread Registers: Each thread has its own set of registers. This includes the program counter (PC) and other registers specific to the thread's execution. Since threads can run independently, their register states (like program counter, instruction pointer, etc.) are not shared.**
3. **Thread-Specific Data (TSD): Some data structures can be thread-specific, such as thread-local storage (TLS). This allows each thread to store its own data, distinct from other threads within the same process.**
4. **Execution Context: Each thread has its own execution context, meaning the state in which the thread is running, such as its CPU state, which is independent of other threads.**
5. **Signal Mask: Threads in a process may have their own signal mask. While signals can be delivered to a process as a whole, the mask (which specifies which signals the thread is willing to handle) can be different for each thread.**

These elements are exclusive to each thread because they are critical for the thread's independent execution flow. Threads within a process typically share resources like memory, file descriptors, and other global states.

Q Suppose a web server is running a process of 10 MB which listens to each incoming client requests and then creates a separate threads to service each client requests. Each thread takes 100 KB memory space. Consider web server has a total 1 GB of main memory and server memory stores only user processes, threads. Find the number of maximum client connections supported by this server.

A-To determine the maximum number of client connections the web server can support, we need to calculate how many threads can be created without exceeding the available memory.

Given data:

- **Memory allocated to the server process: 10 MB**
- **Memory per thread: 100 KB**
- **Total system memory available: 1 GB (1024 MB)**
- **Memory used by server process: 10 MB**
- **Memory available for threads: 1024 MB - 10 MB = 1014 MB**

Calculation steps:

1. **Convert memory available for threads from MB to KB:**
   - **1 MB = 1024 KB**
   - **So, 1014 MB = 1014 × 1024 KB = 1032192 KB**
2. **Calculate the number of threads that can be created:**
   - **Each thread takes 100 KB, so the maximum number of threads is:**

Number of threads=Memory available for threadsMemory per thread=1032192 KB100 KB=10321.92\text{Number of threads} = \frac{\text{Memory available for threads}}{\text{Memory per thread}} = \frac{1032192 \, \text{KB}}{100 \, \text{KB}} = 10321.92

Since the number of threads must be an integer, we round it down to 10321 threads.

Final Answer:

The maximum number of client connections (threads) the server can support is 10,321.

Q- What is critical section problem and what are the requirements that need to be satisfied by any solution to critical section problem? Give a solution to a 2-process critical section problem.

A- Here's a short and concise explanation along with the code for Peterson's solution to the 2-process critical section problem:

Critical Section Problem

The Critical Section Problem occurs when multiple processes access shared resources. The objective is to ensure that only one process enters the critical section at a time, preventing data corruption.

Requirements for Solution

1. **Mutual Exclusion: Only one process should access the critical section at a time.**
2. **Progress: If no process is in the critical section, one of the waiting processes must proceed.**
3. **Bounded Waiting: A process should not wait indefinitely to enter the critical section.**

Peterson's Solution (for Two Processes)

Variables:

- **flag[2]: An array indicating if a process wants to enter the critical section.**
- **turn: A variable indicating whose turn it is.**

Code:

```
// Shared variables

bool flag[2] = {false, false};  // flag[0] for Process 0, flag[1] for Process 1

int turn;  // To give priority to the process


// Process 0 (P0)
```

```
void process0() {
    while (true) {
        flag[0] = true;   // P0 wants to enter critical section
        turn = 1;         // Give priority to Process 1
        while (flag[1] == true && turn == 1);  // Busy-wait if P1 wants to enter and it's P1's turn

        // Critical section for P0
        // Access shared resources here

        flag[0] = false;  // P0 is done with the critical section
    }
}

// Process 1 (P1)
void process1() {
    while (true) {
        flag[1] = true;   // P1 wants to enter critical section
        turn = 0;         // Give priority to Process 0
        while (flag[0] == true && turn == 0);  // Busy-wait if P0 wants to enter and it's P0's turn

        // Critical section for P1
        // Access shared resources here

        flag[1] = false;  // P1 is done with the critical section
    }
}
```

Explanation:

- **Mutual Exclusion: flag ensures that only one process enters the critical section. The turn variable guarantees that processes alternate access.**
- **Progress: If no process is in the critical section, one of the waiting processes is allowed to enter based on the turn variable.**
- **Bounded Waiting: A process won't be blocked indefinitely because it alternates turns with the other process.**

This solution ensures safe concurrent access to shared resources for two processes.

Q- A uni-processor computer system has three processes, which alternate 20 ms CPU bursts with 80 ms I/O bursts. All the processes were created at nearly the same time. The 1/O of all the processes can proceed in parallel. What will be the CPU utilization (over a long period of time) using FCFS and Round Robin (time quantum 10 ms) for this system? Show all calculations.

A- CPU Utilization Calculation for FCFS and Round Robin (Time Quantum = 10 ms)

Given:

- **Each process has a 20 ms CPU burst and an 80 ms I/O burst.**
- **There are 3 processes, all of which start at nearly the same time.**
- **I/O bursts of all processes happen in parallel.**
- **The CPU is idle when all processes are in I/O.**
- **The time quantum for Round Robin is 10 ms.**

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

- **Execution Process:**
  - **Process 1 starts at 0 ms, uses the CPU for 20 ms, and then goes into I/O.**
  - **Process 2 starts at 20 ms, uses the CPU for 20 ms, and then goes into I/O.**
  - **Process 3 starts at 40 ms, uses the CPU for 20 ms, and then goes into I/O.**
- **CPU Utilization:**
  - **The CPU is active from 0 ms to 60 ms (total 60 ms of CPU time).**

    - **After that, all processes are in I/O for 80 ms.**
    - **Total CPU time: 60 ms out of 140 ms (60 ms CPU, 80 ms I/O).**
    - **CPU Utilization = 60140×100=42.86%\frac{60}{140} \times 100 = 42.86\%14060×100=42.86%.**

2. Round Robin (Time Quantum = 10 ms)

- **Execution Process:**
    - **Each process gets 10 ms of CPU time in a round-robin fashion.**
    - **Processes are scheduled in this order until they complete their 20 ms CPU burst (each process gets 2 rounds of 10 ms).**
    - **All processes complete their 20 ms CPU burst by 60 ms.**
    - **After that, all processes enter their I/O burst (80 ms).**
- **CPU Utilization:**
    - **The CPU is active from 0 ms to 60 ms (total 60 ms of CPU time).**
    - **After that, the CPU is idle while all processes are in I/O for 80 ms.**
    - **Total CPU time: 60 ms out of 140 ms.**
    - **CPU Utilization = 60140×100=42.86%\frac{60}{140} \times 100 = 42.86\%14060×100=42.86%.**

Conclusion:

Both FCFS and Round Robin (with a time quantum of 10 ms) yield the same CPU utilization of 42.86%. This is because the CPU is actively used only during the 20 ms CPU bursts, and all processes are in parallel I/O for 80 ms after their bursts, leaving the CPU idle during that time.

Q- Given a system using SJF algorithm for short-term scheduling exponential averaging with a and 0.7, what would be the next expected burst time for a process with burst times of 5, 9, 3 and 5 and an initial value for T1 of 20?

A- To calculate the next expected burst time using Shortest Job First (SJF) with Exponential Averaging (also known as Exponential Smoothing), we use the formula:

Tn+1=α·Tn+(1−α)·tnT_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot t_n

Where:

- **Tn+1T_{n+1} is the next predicted burst time.**
- **TnT_n is the predicted burst time for the current process.**
- **tnt_n is the actual burst time for the current process.**
- **α\alpha is the smoothing factor (given as 0.7 in this case).**

Given:

- **The smoothing factor α=0.7\alpha = 0.7.**
- **The initial predicted burst time T1=20T_1 = 20.**
- **The burst times are t1=5t_1 = 5, t2=9t_2 = 9, t3=3t_3 = 3, and t4=5t_4 = 5.**

We will now use the formula iteratively to calculate the next predicted burst time:

Step 1: Calculate T2T_2

For the second burst time t1=5t_1 = 5:

T2=α·T1+(1−α)·t1T_2 = \alpha \cdot T_1 + (1 - \alpha) \cdot t_1
T2=0.7·20+(1−0.7)·5T_2 = 0.7 \cdot 20 + (1 - 0.7) \cdot 5 T2=14+1.5=15.5T_2 = 14 + 1.5 = 15.5

Step 2: Calculate T3T_3

For the third burst time t2=9t_2 = 9:

T3=α·T2+(1−α)·t2T_3 = \alpha \cdot T_2 + (1 - \alpha) \cdot t_2
T3=0.7·15.5+(1−0.7)·9T_3 = 0.7 \cdot 15.5 + (1 - 0.7) \cdot 9
T3=10.85+2.7=13.55T_3 = 10.85 + 2.7 = 13.55

Step 3: Calculate T4T_4

For the fourth burst time t3=3t_3 = 3:

T4=α·T3+(1−α)·t3T_4 = \alpha \cdot T_3 + (1 - \alpha) \cdot t_3
T4=0.7·13.55+(1−0.7)·3T_4 = 0.7 \cdot 13.55 + (1 - 0.7) \cdot 3
T4=9.485+0.9=10.385T_4 = 9.485 + 0.9 = 10.385

Step 4: Calculate T5T_5 (Next predicted burst time)

For the fifth burst time t4=5t_4 = 5:

T5=α·T4+(1−α)·t4T_5 = \alpha \cdot T_4 + (1 - \alpha) \cdot t_4
T5=0.7·10.385+(1−0.7)·5T_5 = 0.7 \cdot 10.385 + (1 - 0.7) \cdot 5
T5=7.2695+1.5=8.7695T_5 = 7.2695 + 1.5 = 8.7695

Final Answer:

The next expected burst time T5T_5 is 8.77 (rounded to two decimal places).

Q- Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate ofx; when it is running, its priority changes at a rate ß. All processes are given a priority of 0 when they enter the ready queue. The parameters ox and ß can be set to give many different scheduling algorithms. (i) What is the algorithm that results from beta > alpha > 0 ? (ii) What is the algorithm that results from alpha < beta < 0 ?

A- Preemptive Priority Scheduling with Dynamic Priorities

In a preemptive priority scheduling algorithm, processes are scheduled based on their priority. If a process with a higher priority enters the ready queue, it can preempt the currently running process. In the case of dynamically changing priorities, the priorities of processes can increase or decrease depending on whether they are in the ready queue or running.

We have two rates that influence the priority change:

- α (alpha): The rate at which a process's priority increases when it is in the ready queue (waiting for the CPU).
- β (beta): The rate at which a process's priority increases when it is running (actively executing on the CPU).

Let's now analyze the two cases provided:

(i) Algorithm when β > α > 0

Explanation:

- α > 0: When the process is in the ready queue, its priority increases at a rate of α. This means the longer a process stays in the ready queue, the higher its priority becomes.
- β > α: When the process is running, its priority increases at a higher rate β than when it is in the ready queue. This means that processes that

are running gain priority even more rapidly than when they are waiting.

Resulting Algorithm:

- **This type of behavior favors short-term running processes. Since β > α, a process that is running gains priority faster than one in the ready queue. This results in a "starvation" or "preemption-prone" system, where the longer a process runs, the higher its priority becomes, making it less likely to be preempted.**
- **Short jobs or processes that are already running will likely continue running and prevent long-waiting processes from gaining enough priority to preempt them.**
- **This scheduling approach can be seen as a "shortest remaining time" strategy, but with dynamic priority adjustment, prioritizing processes that are already running.**

(ii) Algorithm when α < β < 0

Explanation:

- **α < 0: When a process is in the ready queue, its priority decreases at a rate of α. This means the longer the process waits in the ready queue, the lower its priority becomes.**
- **β < 0: When the process is running, its priority decreases at a rate of β. Since β is smaller than α, the priority decay is faster when the process is running compared to when it is waiting.**

Resulting Algorithm:

- **This type of behavior is more likely to favor processes that are waiting in the ready queue, as their priority decreases less quickly than the processes that are currently running.**
- **The processes that are actively running lose priority rapidly due to β < 0, making them more likely to be preempted by higher priority processes from the ready queue.**
- **This results in a system where waiting processes eventually gain enough priority to preempt the running process. The algorithm can be seen as a form of aging where processes that have been waiting for a long time get promoted to avoid starvation.**

Summary of Algorithms:

1. **When β > α > 0:**
   - **Favoring the running process. This leads to a situation where running processes gain higher priority faster than waiting processes. It can be seen as a variant of shortest remaining time or a preemption-prone strategy.**
2. **When α < β < 0:**
   - **Favoring the waiting process. This creates a aging strategy where long-waiting processes gradually gain higher priority to prevent starvation. Running processes lose priority quickly and are preempted in favor of waiting processes.**

Q- List the two differences between the two mechanisms/models of Inter-process communication in tabular form.

A- Here is a table listing the two common models of Inter-Process Communication (IPC): Message Passing and Shared Memory, along with their differences:

Aspect	**Message Passing**	**Shared Memory**
Communication Method	**Communication occurs by sending messages between processes.**	**Processes communicate by reading/writing to a common shared memory space.**
Synchronization	**Synchronization is handled by the IPC system (e.g., message queues, semaphores).**	**Synchronization must be explicitly managed by the processes (e.g., using locks or semaphores).**

These are two of the main differences between Message Passing and Shared Memory IPC mechanisms.

Q- Name a technique which is used to implement virtual memory concept. What is the motivation behind using that technique ?

A- Technique Used to Implement Virtual Memory: Paging

Motivation Behind Using Paging:

Paging is a technique used to implement virtual memory, where the memory is divided into fixed-size blocks called pages. The physical memory (RAM) is

divided into blocks of the same size called frames. The operating system keeps track of all these pages and frames, mapping virtual addresses to physical addresses using a page table.

Motivation for Paging:

1. **Efficient Memory Use: Paging allows processes to be loaded into non-contiguous memory locations, making better use of the available physical memory. This helps in reducing memory fragmentation.**
2. **Virtual Address Space: It provides a way to give processes the illusion of a large, continuous block of memory (virtual memory) while using smaller, non-contiguous physical memory. This enables efficient and safe allocation of memory.**
3. **Isolation and Security: It ensures that processes are isolated from each other. One process cannot directly access another process's memory, preventing accidental or malicious interference.**

Summary:

Paging improves memory management by enabling efficient use of memory, providing virtual address space, and ensuring process isolation. This technique is a key part of modern operating systems' ability to implement virtual memory.

Q-Consider a system having below segmentation memory management technique implemented. The virtual address is divided as follows: (CO3)

Segment Number (8 bits)

Offset (12 bits)

Then what can be the maximum size of process in this system?

A- Given:

- **Segment Number: 8 bits**
- **Offset: 12 bits**

To calculate the maximum size of a process:

Step 1: Segment Number (8 bits)

- **The segment number is used to identify the segment. Since it is 8 bits, the system can address 2^8 = 256 segments.**

Step 2: Offset (12 bits)

- **The offset refers to the address within a segment. Since the offset is 12 bits, the maximum size of each segment is 2^12 = 4096 bytes (4 KB).**

Step 3: Maximum Process Size

- **Each process can have 256 segments (from the segment number), and each segment can have a size of 4096 bytes (from the offset).**

Therefore, the maximum size of a process is:

Maximum Process Size=Number of Segments×Size of Each Segment\text{Maximum Process Size} = \text{Number of Segments} \times \text{Size of Each Segment} Maximum Process Size=256×4096 bytes\text{Maximum Process Size} = 256 \times 4096 \text{ bytes}
Maximum Process Size=1,048,576 bytes=1 MB\text{Maximum Process Size} = 1,048,576 \text{ bytes} = 1 \text{ MB}

Final Answer:

The maximum size of a process in this system is 1 MB.

Q- Why are page sizes always in powers of 2?

A- Page sizes are always in powers of 2 because it simplifies the calculation of memory addresses. When the page size is a power of 2, the offset and the page number can be efficiently extracted using simple bitwise operations. This improves memory management, address translation, and hardware design.

Q- WHAT to u think about thrashing list the one sol

A- Thrashing occurs when a system spends the majority of its time swapping data between main memory and disk, instead of executing processes. This typically happens when there is insufficient physical memory, causing excessive paging and significant performance degradation.

Solution (1 Mark):

One solution to thrashing is increasing the amount of physical memory (RAM). By providing more memory, the system can handle more processes without excessive paging, thus improving performance and reducing the likelihood of thrashing.

Q-Consider a single level paging scheme. The virtual address space is 4 GB and page size is 128 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page ?

A-To calculate the maximum page table entry size such that the entire page table fits in one page, we can break down the problem as follows:

Step 1: Determine the total number of pages in the virtual address space

The virtual address space is 4 GB, and the page size is 128 KB. We need to determine how many pages the virtual address space is divided into.

Number of pages=Virtual Address SpacePage Size\text{Number of pages} = \frac{\text{Virtual Address Space}}{\text{Page Size}}
Number of pages=4 GB128 KB=4×1024×1024128=32,768 pages\text{Number of pages} = \frac{4 \, \text{GB}}{128 \, \text{KB}} = \frac{4 \times 1024 \times 1024}{128} = 32,768 \, \text{pages}

Step 2: Determine the size of a page

The page size is given as 128 KB.

Page Size=128 KB=128×1024=131,072 bytes\text{Page Size} = 128 \, \text{KB} = 128 \times 1024 = 131,072 \, \text{bytes}

Step 3: Calculate the maximum number of entries in the page table

We want the entire page table to fit in one page. Since the page table is stored in memory, and we want it to fit into a single page, the total size of the page table must be less than or equal to the size of a single page, which is 128 KB.

The total size of the page table can be expressed as:

Page Table Size=(Number of Pages)×(Page Table Entry Size)\text{Page Table Size} = (\text{Number of Pages}) \times (\text{Page Table Entry Size})

Since we want the page table to fit into one page, the page table size must be equal to or less than the page size:

Page Table Size≤131,072 bytes\text{Page Table Size} \leq 131,072 \, \text{bytes} 32,768×Page Table Entry Size≤131,07232,768 \times \text{Page Table Entry Size} \leq 131,072

Step 4: Solve for the maximum page table entry size

Now, solve for the page table entry size:

Page Table Entry Size≤131,07232,768=4 bytes\text{Page Table Entry Size} \leq \frac{131,072}{32,768} = 4 \, \text{bytes}

Conclusion

The maximum page table entry size that ensures the entire page table fits in one page is 4 bytes.

Q-

Name any four techniques used for managing the free space of secondary storage blocks ?

A-Four common techniques used for managing the free space of secondary storage blocks are:

1. **Bitmaps (or Bit Vector):**
   - **A bitmap is a simple array of bits where each bit corresponds to a block on the disk. If a block is free, its corresponding bit is 0; if the block is allocated, the bit is 1. This allows efficient allocation and deallocation of blocks, as well as quick scanning of free blocks.**
2. **Free List:**
   - **A free list is a linked list that maintains a list of all free blocks. Each free block points to the next free block. This method allows efficient allocation by simply removing a block from the front of the list when it is allocated. Deallocation is performed by adding the block back to the list.**
3. **Group Allocation:**
   - **In this method, blocks are allocated in groups. A free space manager keeps track of groups of blocks that are free or allocated. This can reduce fragmentation and improve performance by allocating and deallocating blocks in contiguous groups rather than individually.**
4. **Buddy System:**
   - **The buddy system divides the storage into blocks of varying sizes, with each block being a power of 2. When a block is freed, the system attempts to merge it with its "buddy" block if that block is also free. This helps to minimize fragmentation and allows the system to**

allocate blocks in a flexible way while maintaining a structure that is efficient for allocation and deallocation.

These techniques help improve the efficiency of managing free space on secondary storage, reducing fragmentation and ensuring quick allocation and deallocation of blocks.

Q-Name the four different types of directory structures w.r.t. secondary storage.

(iii) Find the number of hard disk blocks needed to store a 100 KB file, if each hard disk block size is 16 KB.

(iv) Define cylinder, track and sector in context of disk.

(v) What is shell and shell script w.r.t. operating system?

A-(i) Four Types of Directory Structures with Respect to Secondary Storage:

1. Single-Level Directory:
   - In this structure, all files are stored in a single directory. It's simple but inefficient for systems with many users or files, as it leads to potential naming conflicts and difficulty in managing a large number of files.
2. Two-Level Directory:
   - This structure consists of a root directory and separate directories for each user. Each user has their own subdirectory, which helps in organizing files more efficiently. However, it still has some limitations when it comes to sharing files between users.
3. Hierarchical (Tree) Directory:
   - In this structure, directories are organized in a tree-like fashion, where each directory can contain subdirectories and files. This is the most commonly used structure in

modern file systems, allowing for a well-organized and scalable way to store files and manage directories.

4. Acyclic-Graph Directory:
   - This structure allows directories to have links to other directories. Unlike a tree, it permits a file or directory to be part of multiple directories (using hard links), which can help in sharing files across different directories while avoiding duplication.

(ii) Number of Hard Disk Blocks Needed to Store a 100 KB File, If Each Hard Disk Block Size Is 16 KB:

To calculate the number of blocks required:

Number of blocks=File SizeBlock Size\text{Number of blocks} = \frac{\text{File Size}}{\text{Block Size}}

Given:

- File size = 100 KB
- Block size = 16 KB

Number of blocks=100 KB16 KB=6.25\text{Number of blocks} = \frac{100 \, \text{KB}}{16 \, \text{KB}} = 6.25

Since the number of blocks must be an integer, we round up to the next whole number. Therefore, the file would require 7 blocks.

(iii) Define Cylinder, Track, and Sector in the Context of Disk:

1. Cylinder:
   - A cylinder is a set of tracks located at the same position on different platters of a hard disk. It consists of all the tracks that are vertically aligned across all the disk

platters. When the read/write head moves from one track to another without changing the platter, it moves within the same cylinder.

2. Track:
   - A track is a circular path on the surface of a disk platter where data is magnetically recorded. Each platter on the disk is divided into several tracks, and data is written and read from these tracks.
3. Sector:
   - A sector is the smallest unit of data storage on a disk and typically holds 512 bytes (though modern systems may use larger sector sizes, such as 4 KB). A sector is a subdivision of a track. Multiple sectors form a complete track on a disk.

(iv) Shell and Shell Script in the Context of Operating System:

1. Shell:
   - The shell is a command-line interface (CLI) that allows users to interact with the operating system by entering commands. It acts as an intermediary between the user and the kernel of the operating system. The shell interprets the commands entered by the user and executes them, providing access to system services.
2. Shell Script:
   - A shell script is a text file containing a series of shell commands that are executed in sequence. Shell scripts are used to automate tasks, such as system administration, file management, and program execution. They can include loops, conditionals, and

functions, making them powerful tools for automating complex operations in Unix-like operating systems.

Q-Suppose that the disk drive has 200 cylinders, numbered 0 to 199. The drive is currently serving a request at cylinder 53. The queue of pending request, in FIFO order is 98, 183, 37, 122, 14, 124, 14, 124, 65, 67. Starting from the current head position, what is the total distance (in cylinder) that the disk arm moves to satisfy the entire pending request for each of the following disk scheduling algorithms? (i) FCFS (ii) SSTF (iii) SCAN (iv) C-SCAN

A-We are given a disk drive with 200 cylinders (numbered 0 to 199) and the current position of the disk head at cylinder 53. The pending requests in the queue are:

98, 183, 37, 122, 14, 124, 14, 124, 65, 67

Let's calculate the total distance the disk arm moves for each disk scheduling algorithm:

(i) FCFS (First-Come, First-Served)

FCFS processes requests in the order they arrive.

- Current head position = 53
- Requests to serve in FIFO order = 98, 183, 37, 122, 14, 124, 14, 124, 65, 67

To calculate the total distance moved, we sum the absolute differences between the current head position and each request, processed one after another:

1. Move from 53 to 98: |98 - 53| = 45
2. Move from 98 to 183: |183 - 98| = 85
3. Move from 183 to 37: |37 - 183| = 146
4. Move from 37 to 122: |122 - 37| = 85

5. Move from 122 to 14: |14 - 122| = 108
6. Move from 14 to 124: |124 - 14| = 110
7. Move from 124 to 14: |14 - 124| = 110
8. Move from 14 to 124: |124 - 14| = 110
9. Move from 124 to 65: |65 - 124| = 59
10. Move from 65 to 67: |67 - 65| = 2

Total distance for FCFS = 45+85+146+85+108+110+110+110+59+2=780 cylinders45 + 85 + 146 + 85 + 108 + 110 + 110 + 110 + 59 + 2 = 780 \, \text{cylinders}

(ii) SSTF (Shortest Seek Time First)

SSTF serves the request closest to the current head position.

- Current head position = 53
- Requests: 98, 183, 37, 122, 14, 124, 14, 124, 65, 67

We select the closest request and move the head to that position, repeating the process for the remaining requests.

1. Move from 53 to 65 (closest): |65 - 53| = 12
2. Move from 65 to 67 (closest): |67 - 65| = 2
3. Move from 67 to 37 (closest): |37 - 67| = 30
4. Move from 37 to 14 (closest): |14 - 37| = 23
5. Move from 14 to 14 (same request): 0
6. Move from 14 to 98 (closest): |98 - 14| = 84
7. Move from 98 to 122 (closest): |122 - 98| = 24
8. Move from 122 to 124 (closest): |124 - 122| = 2
9. Move from 124 to 124 (same request): 0

10. Move from 124 to 183 (last request): |183 - 124| = 59

Total distance for SSTF = 12+2+30+23+0+84+24+2+0+59=236 cylinders12 + 2 + 30 + 23 + 0 + 84 + 24 + 2 + 0 + 59 = 236 \, \text{cylinders}

(iii) SCAN (Elevator Algorithm)

In SCAN, the disk arm moves in one direction, serving all requests in that direction, and then reverses direction when it reaches the end of the disk.

- **Current head position = 53**
- **Requests: 98, 183, 37, 122, 14, 124, 14, 124, 65, 67**

The head will move towards the highest cylinder first (increasing order), serving all requests in that direction, then reverse to serve requests in the opposite direction.

1. **Move from 53 to 67: Move right (next request in increasing order). |67 - 53| = 14**
2. **Move from 67 to 98: |98 - 67| = 31**
3. **Move from 98 to 122: |122 - 98| = 24**
4. **Move from 122 to 124: |124 - 122| = 2**
5. **Move from 124 to 124: 0 (same request)**
6. **Move from 124 to 183: |183 - 124| = 59**

At this point, the head is at cylinder 183, and it will reverse direction and move towards the lowest cylinder (cylinder 0).

7. **Move from 183 to 14: |183 - 14| = 169**
8. **Move from 14 to 14: 0 (same request)**
9. **Move from 14 to 37: |37 - 14| = 23**

10. Move from 37 to 65: |65 - 37| = 28

Total distance for SCAN = 14+31+24+2+0+59+169+0+23+28=350 cylinders14 + 31 + 24 + 2 + 0 + 59 + 169 + 0 + 23 + 28 = 350 \, \text{cylinders}

(iv) C-SCAN (Circular SCAN)

In C-SCAN, the disk arm moves in one direction, serving requests in that direction, but when it reaches the end, it jumps to the beginning of the disk and continues servicing the requests in that direction.

- **Current head position = 53**
- **Requests: 98, 183, 37, 122, 14, 124, 14, 124, 65, 67**

The head will move towards the highest cylinder first (increasing order), serving all requests in that direction, and then jump to the beginning (cylinder 0) to serve the remaining requests.

1. **Move from 53 to 67: |67 - 53| = 14**
2. **Move from 67 to 98: |98 - 67| = 31**
3. **Move from 98 to 122: |122 - 98| = 24**
4. **Move from 122 to 124: |124 - 122| = 2**
5. **Move from 124 to 124: 0 (same request)**
6. **Move from 124 to 183: |183 - 124| = 59**

At this point, the head is at cylinder 183, and it jumps to cylinder 0.

7. **Jump to cylinder 0: No distance**
8. **Move from 0 to 14: |14 - 0| = 14**
9. **Move from 14 to 14: 0 (same request)**
10. **Move from 14 to 37: |37 - 14| = 23**

11. Move from 37 to 65: |65 - 37| = 28

Total distance for C-SCAN = 14+31+24+2+0+59+0+14+0+23+28=195 cylinders14 + 31 + 24 + 2 + 0 + 59 + 0 + 14 + 0 + 23 + 28 = 195 \, \text{cylinders}

Final Results:

1. FCFS: 780 cylinders
2. SSTF: 236 cylinders
3. SCAN: 350 cylinders
4. C-SCAN: 195 cylinders

Q-Consider a 500 GB hard disk having 32 KB block sizes. The size of the block pointer is 2 bytes. File allocation table (FAT) based file system is used to keep track of allocated blocks on hard disk. Then:

(i) What will be the total no. of entries in the FAT?

(ii) Find the size of the file allocation table (FAT) in Megabytes (MB).

(iii) How many HDD blocks are needed to store FAT?

(iv) How much percentage of HDD is used to store FAT?

(v) Find the maximum size of a file that can be stored on this disk as per above FAT

A-We are given the following details about the hard disk:

- Disk size = 500 GB
- Block size = 32 KB
- Block pointer size = 2 bytes
- File Allocation Table (FAT) is used to keep track of allocated blocks

(i) What will be the total number of entries in the FAT?

The FAT stores one entry for each block on the disk. To find the total number of entries in the FAT, we need to calculate the total number of blocks on the disk.

1. **Convert the disk size into KB:**

Disk size=500 GB=500×1024 MB=500×1024×1024 KB=524288000 KB \text{Disk size} = 500 \, \text{GB} = 500 \times 1024 \, \text{MB} = 500 \times 1024 \times 1024 \, \text{KB} = 524288000 \, \text{KB}

2. **Find the total number of blocks:**

Number of blocks=Disk size in KBBlock size in KB=524288000 KB32 KB=16384000 blocks\text{Number of blocks} = \frac{\text{Disk size in KB}}{\text{Block size in KB}} = \frac{524288000 \, \text{KB}}{32 \, \text{KB}} = 16384000 \, \text{blocks}

Thus, the total number of entries in the FAT is 16,384,000.

(ii) Find the size of the File Allocation Table (FAT) in Megabytes (MB).

The size of the FAT is determined by the number of entries and the size of each entry (block pointer). The number of entries is 16,384,000, and the size of each entry is 2 bytes.

1. **Size of the FAT in bytes:**

Size of FAT in bytes=Number of entries×Size of each entry=16384000×2 bytes=32768000 bytes\text{Size of FAT in bytes} = \text{Number of entries} \times \text{Size of each entry} = 16384000 \times 2 \, \text{bytes} = 32768000 \, \text{bytes}

2. **Convert the size of FAT into Megabytes (MB):**

Size of FAT in MB=32768000 bytes1024×1024=31.25 MB\text{Size of FAT in MB} = \frac{32768000 \, \text{bytes}}{1024 \times 1024} = 31.25 \, \text{MB}

Thus, the size of the FAT is 31.25 MB.

(iii) How many HDD blocks are needed to store FAT?

To calculate how many blocks are needed to store the FAT, we divide the total size of the FAT by the block size.

1. Number of blocks needed to store the FAT:

Number of blocks for FAT=Size of FAT in bytesBlock size in bytes=32768000 bytes32×1024 bytes=3276800032768=1000 blocks\text{Number of blocks for FAT} = \frac{\text{Size of FAT in bytes}}{\text{Block size in bytes}} = \frac{32768000 \, \text{bytes}}{32 \times 1024 \, \text{bytes}} = \frac{32768000}{32768} = 1000 \, \text{blocks}

Thus, 1000 HDD blocks are needed to store the FAT.

(iv) How much percentage of HDD is used to store FAT?

To calculate the percentage of the hard disk used to store the FAT, we divide the size of the FAT by the total disk size and then multiply by 100.

1. Percentage of disk used by the FAT:

Percentage used by FAT=Size of FATDisk size×100=31.25 MB500000 MB×100=0.00625%\text{Percentage used by FAT} = \frac{\text{Size of FAT}}{\text{Disk size}} \times 100 = \frac{31.25 \, \text{MB}}{500000 \, \text{MB}} \times 100 = 0.00625\%

Thus, 0.00625% of the HDD is used to store the FAT.

(v) Find the maximum size of a file that can be stored on this disk as per above FAT.

The maximum size of a file is determined by the number of blocks that can be allocated to a file, and each block is of size 32 KB.

1. **Maximum number of blocks that can be allocated to a file:**

The maximum number of blocks that can be allocated to a file is equal to the total number of blocks on the disk, which is 16,384,000 blocks.

2. **Maximum file size:**

Maximum file size=Number of blocks×Block size=16384000×32 KB=524288000 KB\text{Maximum file size} = \text{Number of blocks} \times \text{Block size} = 16384000 \times 32 \, \text{KB} = 524288000 \, \text{KB}

Convert this to GB:

Maximum file size in GB=524288000 KB1024×1024=500 GB\text{Maximum file size in GB} = \frac{524288000 \, \text{KB}}{1024 \times 1024} = 500 \, \text{GB}

Thus, the maximum size of a file that can be stored on this disk is 500 GB.

Summary of Answers:

1. **Total number of entries in the FAT: 16,384,000**
2. **Size of the FAT in MB: 31.25 MB**
3. **Number of HDD blocks needed to store FAT: 1000 blocks**
4. **Percentage of HDD used to store FAT: 0.00625%**
5. **Maximum size of a file: 500 GB**

Q-(a)

(i) What is translation lookaside buffer? What is stored inside it?

(ii) What is page fauit? What actions are taken after page fault?

(iii) On what factors does the allocation of minimum and maximum number of page frames to a process in a system depend?

(iv) Draw a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

(v) Consider a computer system with 40- bit virtual addressing and page size of 16 KB. If the computer system has a one-level page table per process and each page table entry requires 48 bits, then find the size of the per-process page table in megabytes (MB),

A-(a)

(i) What is Translation Lookaside Buffer (TLB)? What is stored inside it?

A Translation Lookaside Buffer (TLB) is a small, fast memory cache that stores recent translations of virtual memory addresses to physical memory addresses. When a process accesses a memory location, the TLB is checked to see if the translation for the virtual address is already stored. If it is, the physical address can be retrieved quickly (a TLB hit). If the translation is not found (a TLB miss), the system must reference the page table to obtain the physical address.

What is stored inside the TLB? The TLB typically stores a small number of entries containing the following information:

- **Virtual page number (VPN): The virtual address of the page.**
- **Physical frame number (PFN): The corresponding physical memory address (frame) for the virtual page.**
- **Additional control bits: For example, valid bit, access permissions (read/write), etc.**

TLBs significantly improve the speed of address translation by reducing the need to access the full page table for each memory access.

(ii) What is a page fault? What actions are taken after a page fault?

A page fault occurs when a program attempts to access a page of memory that is not currently in physical memory (RAM). This can happen if the page is not loaded into memory or has been swapped out to disk.

Actions taken after a page fault:

1. **Check for valid memory access: The operating system checks if the access is valid, i.e., if the page is part of the process's address space and the access rights are valid (read/write/execute).**
2. **Load the page into memory: If the page is valid and not already in memory, it must be loaded from disk (swap space or disk file) into RAM. This may involve choosing a page to evict if there is no free space.**
3. **Update page table: The page table is updated to reflect the new physical memory location of the page.**
4. **Restart the instruction: After the page is loaded, the instruction that caused the page fault is restarted, and the program continues execution.**

If the access is invalid (for example, accessing memory outside the process's allocated space), a segmentation fault or access violation is raised.

(iii) On what factors does the allocation of minimum and maximum number of page frames to a process in a system depend?

The allocation of page frames to a process depends on several factors:

1. **Total available physical memory: The amount of physical memory in the system affects how many frames can be allocated across all processes.**
2. **Size of the process: Larger processes that require more memory will need more page frames to hold their data and code in memory.**
3. **System load: If the system is running many processes simultaneously, the operating system may allocate fewer frames to each process to ensure that all processes have some memory. Conversely, if fewer processes are running, the operating system may allocate more frames to each process.**
4. **Process priority: High-priority processes or real-time processes may be allocated more memory resources.**
5. **Page replacement policy: Depending on the page replacement algorithm (e.g., LRU, FIFO, etc.), the system may adjust the number of frames allocated to a process dynamically based on the number of page faults and the behavior of the process.**
6. **Threshold for minimum and maximum frames: The system may set a minimum and maximum threshold for the number of page frames allocated to a process. This ensures that each process has a reasonable amount of memory to avoid excessive paging while also not consuming all the system's resources.**

(iv) Draw a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

To enforce memory protection and prevent programs from modifying each other's memory, modern systems use a Memory Management Unit (MMU), which uses techniques such as base and limit registers or page tables to provide memory isolation.

Here is a simplified mechanism for memory protection:

1. Base and Limit Registers:
    - Each process has a base register that points to the start of the process's allocated memory.
    - The limit register specifies the size of the allocated memory region for the process.
    - When a program accesses memory, the MMU checks if the address is within the range of the base and limit registers. If the address is outside this range, a segmentation fault or memory protection error is triggered.
2. Page Tables with Protection Bits:
    - Each page of memory has an entry in the page table with protection bits (e.g., read, write, execute).
    - The MMU checks the protection bits before allowing a memory access. If the access violates the protection rules (e.g., trying to write to a read-only page), an error is raised.

Diagram:

```
+---------------------------------+
|        User Program          |  <--- This program can access only its allocated memory
|      (Process Address Space)     |
|                          |
```

+---------------------------------+

| |

| Kernel Space | <--- Kernel is protected from user programs

| |

+---------------------------------+

| Memory Management Unit | <--- MMU checks if access is within limits and protection rules

+---------------------------------+

In this way, memory protection ensures that processes can only access their allocated memory and cannot overwrite other processes' memory, preserving system stability and security.

(v) Consider a computer system with 40-bit virtual addressing and a page size of 16 KB. If the computer system has a one-level page table per process and each page table entry requires 48 bits, then find the size of the per-process page table in megabytes (MB).

Let's break this down:

1. Virtual Address Space:
   With 40-bit virtual addressing, the total addressable memory is:

240 bytes=1 TB(1 Terabyte)2^{40} \, \text{bytes} = 1 \, \text{TB} \quad (\text{1 Terabyte})

2. Page Size:
   Each page is 16 KB (which is 2142^{14} bytes).
3. Number of Pages: The number of pages required to cover the entire 1 TB address space is:

240 bytes16 KB (214 bytes)=240214=226 pages\frac{2^{40} \, \text{bytes}}{16 \, \text{KB} \, (\text{2}^{14} \, \text{bytes})} = \frac{2^{40}}{2^{14}} = 2^{26} \, \text{pages}

So, we need 2262^{26} entries in the page table.

4. **Page Table Entry Size: Each page table entry requires 48 bits, or 6 bytes (since 1 byte = 8 bits).**
5. **Size of the Page Table: The total size of the page table is:**

Page Table Size=Number of Entries×Size of each entry\text{Page Table Size} = \text{Number of Entries} \times \text{Size of each entry} Page Table Size=226×6 bytes\text{Page Table Size} = 2^{26} \times 6 \, \text{bytes}
Page Table Size=6×226 bytes=228 bytes=256 MB\text{Page Table Size} = 6 \times 2^{26} \, \text{bytes} = 2^{28} \, \text{bytes} = 256 \, \text{MB}

Thus, the size of the per-process page table is 256 MB.

Summary of Answers:

1. **TLB: A cache for storing recent virtual-to-physical address translations.**
2. **Page Fault: Occurs when a program accesses a page not in memory, leading to loading it from disk and updating the page table.**
3. **Page Frame Allocation: Depends on available physical memory, process size, system load, priority, and page replacement policy.**
4. **Memory Protection: Enforced via base/limit registers or page table with protection bits, preventing programs from accessing other programs' memory.**

5. **Page Table Size Calculation: The size of the per-process page table is 256 MB.**

Q-Consider a demand-paging system wir a paging disk that has an average access and transfer time of 20 ms. Addresses are translated through a page table in main memory, with an access time of 1 u per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory. Assume that 80% of the time the data or instructions are found in the associative memory, 10% of the remaining time, data or instruction access cause page faults and for the rest of the time; it is found in main memory. What is the effective memory acess time

A-To compute the Effective Memory Access Time (EMAT) for a demand-paging system, we consider all possible cases for accessing a memory reference. Given that the system uses an associative memory (TLB), the calculation takes into account the hit rate in the TLB, the page fault rate, and the average disk access time when a page fault occurs.

Given Information:

- **Associative Memory (TLB) Hit Rate: 80% (0.80)**
- **Page Fault Rate (after TLB miss): 10% of the remaining accesses (0.10)**
- **Main Memory Access Time: 1 µs per access**
- **Disk Access Time (when a page fault occurs): 20 ms = 20,000 µs**
- **Page Table Accesses: 2 memory accesses (because it takes 1 memory access to fetch the page table entry, and another to fetch the data)**

Breakdown of the Memory Access Scenarios:

1. **TLB Hit (80% of the time):**
   - **If the page table entry is found in the TLB, we only need 1 memory access to retrieve the data (since TLB is directly in hardware).**

Access time for TLB hit = 1 memory reference = 1 µs.

2. **TLB Miss, but Data in Main Memory (10% of the remaining 20% time):**
   - **If the data is not found in the TLB, but it is in main memory, it requires 2 memory accesses (one for the page table lookup and one for accessing the data).**

Access time for TLB miss, data in main memory = 2 memory accesses = 2 µs.

3. **Page Fault (10% of the remaining 20% time):**
   - **If there is a page fault, the system must fetch the page from disk, which takes 20 ms (20,000 µs), plus the usual 2 memory accesses for page table lookup and data access.**

Access time for page fault = 20,000 µs (disk access) + 2 µs (memory accesses) = 20,002 µs.

Calculation of Effective Memory Access Time:

We can calculate the Effective Memory Access Time (EMAT) by considering the weighted average of these different cases:

EMAT=(0.80×1 µs)+(0.20×0.10×2 µs)+(0.20×0.10×20,002 µs)\text{EMAT} = (0.80 \times 1 \, \mu\text{s}) + (0.20 \times 0.10 \times 2 \, \mu\text{s}) + (0.20 \times 0.10 \times 20,002 \, \mu\text{s})

Let's break this down:

- For the TLB hit: 0.80×1 µs=0.80 µs0.80 \times 1 \, \mu\text{s} = 0.80 \, \mu\text{s}
- For the TLB miss, but data in memory: 0.20×0.10×2 µs=0.04 µs0.20 \times 0.10 \times 2 \, \mu\text{s} = 0.04 \, \mu\text{s}
- For the page fault: 0.20×0.10×20,002 µs=400.04 µs0.20 \times 0.10 \times 20,002 \, \mu\text{s} = 400.04 \, \mu\text{s}

Now, summing these terms:

EMAT=0.80 µs+0.04 µs+400.04 µs=400.88 µs\text{EMAT} = 0.80 \, \mu\text{s} + 0.04 \, \mu\text{s} + 400.04 \, \mu\text{s} = 400.88 \, \mu\text{s}

Thus, the Effective Memory Access Time (EMAT) is 400.88 µs.

Q-(i) Write a shell script to enter 2 strings from the user. Print whether the entered strings are the same or not. Also check whether the length of each string is equal to zero or not and print appropriate messages.

(ii) Explain in brief five different types of shell environment variables.

A-(i) Shell Script to Enter Two Strings and Check Conditions

Below is a shell script that prompts the user to enter two strings, compares them, checks if their lengths are zero, and prints appropriate messages:

```
#!/bin/bash


# Prompt the user to enter the first string

echo "Enter the first string: "

read string1
```

```
# Prompt the user to enter the second string
echo "Enter the second string: "
read string2

# Compare the two strings
if [ "$string1" == "$string2" ]; then
  echo "The strings are the same."
else
  echo "The strings are not the same."
fi

# Check if the length of the first string is zero
if [ -z "$string1" ]; then
  echo "The first string is empty."
else
  echo "The first string is not empty."
fi

# Check if the length of the second string is zero
if [ -z "$string2" ]; then
  echo "The second string is empty."
else
  echo "The second string is not empty."
```

fi

Explanation:

1. **Input Strings: The script uses the read command to get input from the user for both string1 and string2.**
2. **String Comparison: The if condition compares the two strings using ==. If the strings are equal, it prints "The strings are the same", otherwise, it prints "The strings are not the same".**
3. **Check Empty Strings: The -z option checks if the string length is zero. If -z returns true, it prints that the string is empty. Otherwise, it prints that the string is not empty.**

Example Output:

```
Enter the first string: hello

Enter the second string: world

The strings are not the same.

The first string is not empty.

The second string is not empty.
```

(ii) Explanation of Five Different Types of Shell Environment Variables

In a shell environment, variables can be classified into different types. Here are five common types of shell environment variables:

1. **User-defined Variables:**
    - **These are variables defined by the user in the shell script or command line.**
    - **Example:**
    - **NAME="John"**

- echo $NAME # Output: John
- The user can define any variable to store values during script execution.

2. System-defined Variables:
   - These are variables that are set by the system or shell environment automatically.
   - Example: $HOME, $PATH, $USER
     - $HOME stores the current user's home directory.
     - $PATH defines the directories the shell should search for executable files.
     - $USER stores the current user's login name.
3. Shell Special Variables:
   - These variables have a specific meaning in shell scripting and can be used for special purposes.
   - Example:
     - $?: Stores the exit status of the last executed command.
     - $#: Stores the number of arguments passed to a script.
     - $0: Stores the name of the shell script itself.
4. Environment Variables:
   - These variables are inherited by child processes and can be accessed globally by any program that is run from the shell.
   - Example:

        - export PATH="$PATH:/new/path": Adds a new directory to the system's search path for executables.
        - export HOME=/home/username: Sets the user's home directory globally.

5. Read-only Variables:
    - These variables cannot be changed after their initial assignment. Once set, their value is constant.
    - Example:
    - readonly PI=3.14159
    - PI=3 # This will result in an error, as PI is a read-only variable
    - readonly is used to define such variables to prevent accidental modification.

Summary of Types:

- User-defined variables: Created by users, can store arbitrary data.
- System-defined variables: Automatically set by the system, like $USER, $PATH.
- Shell special variables: Special meaning in shell scripts, like $?, $0, etc.
- Environment variables: Global variables inherited by child processes, like $HOME, $PATH.
- Read-only variables: Variables that cannot be changed after initialization, like readonly PI.