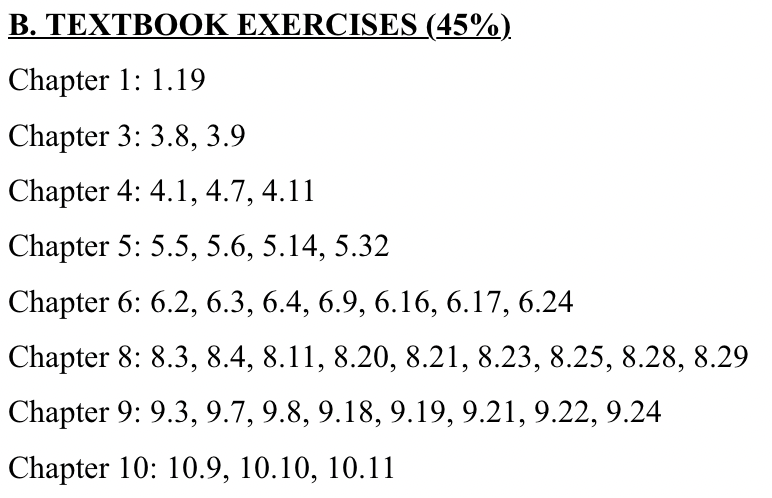
****

**1.19** **What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?**

Interrupts serve as a mechanism for the CPU to respond to asynchronous events, allowing the operating system to execute specific routines when certain conditions occur, such as the completion of I/O operations or hardware malfunctions. They facilitate communication between hardware devices and the CPU, ensuring that the system can manage multiple tasks effectively by interrupting the CPU's current activities to address more urgent tasks.

A trap, also known as a software-generated interrupt, is a specific type of interrupt that occurs as a result of an instruction executed by a program, typically when an error occurs (like division by zero) or when a program requests a service from the operating system. In contrast to hardware interrupts, which are generated by external devices, traps are generated internally by the CPU.

Yes, traps can be generated intentionally by a user program, usually through system calls that request services from the operating system. This is done to perform operations that require higher privileges, such as accessing hardware resources, managing memory, or handling errors.

**3.8** **Describe the differences among short-term, medium-term, and long-term scheduling.**

**#include <stdio.h>**

**#include <unistd.h>**

**int main()**

**{**

**int i;**

**for (i = 0; i < 4; i++)**

**fork();**

**return 0;**

**}**

1. Short-term Scheduling (CPU Scheduling):

- This type of scheduling is responsible for selecting which of the ready processes in memory should be allocated the CPU.

- It operates on a very short time scale, making decisions often—typically at least once every 100 milliseconds, depending on the system load.

- The goal of short-term scheduling is to maximize CPU utilization and ensure that processes are executed promptly.

- It is often referred to as the CPU scheduler, and its decisions are made based on the process's priority and CPU burst time [[n]].

2. \*\*Medium-term Scheduling\*\*:

- Medium-term scheduling is concerned with the swapping of processes in and out of memory to manage the degree of multiprogramming.

- It temporarily removes processes from main memory (swaps them out) and later restores them (swaps them back in) to optimize performance and manage memory effectively.

- This type of scheduling is invoked less frequently than short-term scheduling and is typically concerned with balancing the load in memory, ensuring that there is a mix of I/O-bound and CPU-bound processes.

- The medium-term scheduler might be invoked when the system is overcommitted, meaning there are too many processes competing for limited memory [[n]].

3. \*\*Long-term Scheduling (Job Scheduling)\*\*:

- Long-term scheduling determines which processes are admitted to the system for processing. It controls the degree of multiprogramming and the number of processes that are in memory.

- This type of scheduling is invoked less frequently than both short-term and medium-term scheduling, often only when a new job is submitted.

- Long-term scheduling is responsible for deciding which processes should be loaded into memory based on their expected CPU bursts, I/O requirements, and overall system load.

- The long-term scheduler has a broader view of system resources and aims to ensure a balanced load across CPU and I/O [[n]].

Short-term scheduling deals with the immediate allocation of CPU resources, medium-term scheduling manages the swapping of processes in memory, and long-term scheduling decides which processes are allowed to enter the system for execution.

**3.9** **Describe the actions taken by a kernel to context-switch between**

1. \*\*Save the Context of the Current Process\*\*: The kernel saves the state of the currently running process. This includes the values of the CPU registers, the program counter, and the process state. This information is typically stored in the Process Control Block (PCB) of the process.

2. Update the Process State: The state of the currently running process is updated to indicate that it is no longer running. It may be moved to a waiting or ready state, depending on whether it is blocked or can run again later.

3. Select the Next Process to Run: The kernel's scheduler selects the next process to run based on the scheduling algorithm in use. This may involve examining the ready queue and determining which process has the highest priority or is next in line to be executed.

4. Load the Context of the Next Process: The kernel loads the saved context of the selected process from its PCB. This includes restoring the values of the CPU registers and the program counter to where it left off.

5. Update the Process State: The state of the newly scheduled process is updated to indicate that it is now running.

6. Switch the CPU to the New Process: Finally, the kernel hands control of the CPU over to the newly scheduled process, allowing it to start executing [[Page 138]].

These steps ensure that the system can efficiently switch between multiple processes while keeping track of their individual states and contexts.

**4.1** **Provide two programming examples in which multithreading provides better performance than a single-threaded solution.**

1. \*\*Web Server Handling Multiple Requests\*\*: A web server can utilize multithreading to handle multiple client requests concurrently. For instance, a multithreaded server can create a new thread for each incoming client connection, allowing multiple clients to be served at the same time. If a web server were single-threaded, it would handle one request at a time, causing significant delays for users as they wait for their turn. In a multithreaded approach, while one thread is waiting for I/O operations (like reading from a database or sending data over the network), other threads can continue processing other requests, thus improving the overall throughput and responsiveness of the server.

2. \*\*Data Processing in Parallel\*\*: Consider a scenario where a program needs to perform computations on a large dataset, such as image processing or statistical analysis. By using multiple threads, the program can divide the dataset into smaller chunks and assign each chunk to a different thread for processing. This allows the CPU to perform calculations on multiple chunks simultaneously, significantly reducing the total processing time. In contrast, a single-threaded solution would process each chunk sequentially, leading to longer execution times. For example, if the dataset consists of millions of data points, using multithreading can harness the power of multicore processors to complete the task much faster [[4]].

These examples illustrate how multithreading can enhance performance by improving resource utilization and decreasing processing time.

**4.7** **Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single-processor system?**

A multithreaded solution using multiple kernel threads can provide better performance than a single-threaded solution on a single-processor system under certain circumstances, such as:

1. \*\*I/O-Bound Tasks\*\*: If the application involves tasks that frequently wait for I/O operations (like disk reads/writes or network communication), multiple kernel threads can improve responsiveness. While one thread is waiting for I/O, others can continue executing, thus keeping the CPU busy and reducing idle time.

2. \*\*Concurrency in User Interaction\*\*: In applications with user interfaces, multithreading allows the application to remain responsive. For instance, while one thread handles user input, another can perform background processing, enhancing the user's experience without freezing the interface.

3. \*\*Task Switching Overhead\*\*: If the threads share a significant amount of data or resources, switching between kernel threads may incur less overhead than creating and managing separate processes. This is especially true if the threads are lightweight and the application is designed to minimize context-switching costs.

4. \*\*Improved Resource Utilization\*\*: Even on a single processor, multiple threads can manage resource contention more effectively, allowing for better throughput when tasks are structured to wait for resources instead of blocking each other.

5. \*\*Parallel Execution of Tasks\*\*: Although true parallel execution doesn't occur on a single processor, a multithreaded application can interleave the execution of multiple tasks, allowing the system to utilize CPU time better. For example, while one thread is waiting for a memory operation to complete, the other threads can execute.

While a single-threaded solution may be simpler and may work well for CPU-bound tasks, multithreading can enhance performance in scenarios involving I/O-bound tasks, user interactions, and efficient resource management.

**4.11 Is it possible to have concurrency but not parallelism? Explain.**

**5.5** **Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.**

To demonstrate that if the `wait()` and `signal()` semaphore operations are not executed atomically, mutual exclusion may be violated, we can consider a situation involving two processes and a binary semaphore.

Let's assume we have two processes, P1 and P2, that both want to enter a critical section protected by a binary semaphore (let's call it `mutex`). The semaphore is initialized to 1, indicating that the critical section can be entered by one process at a time.

The operations for the semaphore are defined as follows:

- `wait(mutex)`:

- Decrement the value of the semaphore.

- If the value becomes negative, the process must wait (block).

- `signal(mutex)`:

- Increment the value of the semaphore.

- If the value is not positive, wake up a waiting process.

Now, if the `wait()` and `signal()` operations are not executed atomically, the following scenario can occur:

1. Process P1 executes `wait(mutex)`. It decrements the semaphore from 1 to 0 and enters the critical section.

2. Before P1 executes the `signal(mutex)` operation to release the semaphore, a context switch occurs due to the scheduler, and process P2 is scheduled to run.

3. Process P2 executes `wait(mutex)`. Since the semaphore is now 0, P2 is allowed to enter the critical section without being blocked. It decrements the semaphore again (which is not supposed to happen in mutual exclusion) to -1.

4. Now both P1 and P2 are in the critical section, violating mutual exclusion.

This situation illustrates that if `wait()` and `signal()` are not atomic, the critical section can be entered by more than one process simultaneously, resulting in a race condition. Thus, mutual exclusion is not preserved.

Hence, it is crucial for the semaphore operations to be atomic to ensure that only one process can modify the semaphore's state at a time, thereby maintaining mutual exclusion.

**5.6** **Illustrate how a binary semaphore can be used to implement mutual exclusion among *n* processes**

To implement mutual exclusion among n processes using a binary semaphore, we can utilize a simple structure based on the concept of a binary semaphore, which can either be 0 (unavailable) or 1 (available). The binary semaphore will control access to the critical section of code that should be executed by only one process at a time.

Here's a pseudocode illustration of how this can be done:

1. \*\*Initialize the binary semaphore\*\*:

- Set the binary semaphore to 1 (indicating that the critical section is available).

```pseudocode

binary\_semaphore mutex = 1; // Binary semaphore initialized to 1

```

2. \*\*Define the critical section\*\*:

- Each process will attempt to enter the critical section where shared resources are accessed.

3. \*\*Implementing the mutual exclusion\*\*:

- Each process will perform a wait operation before entering the critical section and a signal operation after exiting.

```pseudocode

process P[i] {

while (true) {

// Entry section

wait(mutex); // P[i] tries to enter the critical section

// Critical section

// Access shared resources

// ...

// Exit section

signal(mutex); // P[i] exits the critical section

// Remainder section

// Perform other tasks

// ...

}

}

```

4. \*\*Explanation of the operations\*\*:

- The `wait(mutex)` operation:

- It checks the value of the semaphore. If the value is 1, it sets it to 0 and allows the process to enter the critical section.

- If the value is 0, the process will block and wait until the semaphore is released by another process.

- The `signal(mutex)` operation:

- It sets the semaphore back to 1, allowing another waiting process to enter the critical section if there are any.

This approach ensures that only one process can access the critical section at any one time, thus achieving mutual exclusion. If multiple processes attempt to enter the critical section simultaneously, the binary semaphore will manage access, ensuring that only one process proceeds while others wait.

This method is efficient and widely used in concurrent programming to prevent race conditions and ensure data integrity when multiple processes or threads are accessing shared resources.

This concept is discussed in detail in Chapter 5, particularly the sections on semaphores as synchronization tools and the structure of mutual exclusion implementations.

**5.14** **Describe how the compare and swap() instruction can be used to pro-vide mutual exclusion that satisfies the bounded-waiting requirement.**

The compare and swap() instruction can be used to provide mutual exclusion while ensuring the bounded-waiting requirement by allowing processes to atomically check and modify a value in memory. This instruction operates on three parameters: a pointer to a value, an expected value, and a new value. It swaps the current value at the given pointer with the new value only if the current value matches the expected value, returning the original value regardless of whether the swap occurred.

Here's a simplified explanation of how it can be implemented to achieve mutual exclusion:

1. \*\*Initialization\*\*: A global variable, often called `lock`, is initialized to 0 to indicate that the lock is available.

2. \*\*Acquiring the Lock\*\*:

- A process that wants to enter a critical section calls the compare and swap() function, passing in the address of `lock`, the expected value of 0, and the new value of 1.

- If the `lock` was 0 (indicating it was available), the compare and swap() will set it to 1 and the process can enter the critical section.

- If the `lock` was already 1 (indicating it is held by another process), the compare and swap() will fail, and the process must wait (typically in a loop, retrying the operation).

3. \*\*Exiting the Critical Section\*\*:

- When a process exits the critical section, it simply sets the `lock` variable back to 0, indicating that it is available again.

4. \*\*Bounded-Waiting\*\*: To ensure that the waiting time is bounded, instead of busy-waiting indefinitely, the processes can implement a queuing mechanism or use a waiting variable. For instance:

- Each process can maintain a waiting queue, and only a limited number of retries (attempts to acquire the lock) can be made before the process yields control or sleeps for a certain period.

- This way, if many processes are competing for the lock, the waiting times are controlled, ensuring that no single process can starve indefinitely.

This approach effectively prevents race conditions while ensuring that processes do not wait indefinitely to enter their critical sections, thus satisfying the bounded-waiting requirement.

The specific implementation details may vary depending on the system architecture and the exact requirements of the application, but the principle remains the same.

**5.32** **A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor to coordinate access to the file.**

To implement a monitor for coordinating access to a shared file with the specified constraint (the sum of all unique process numbers accessing the file must be less than a given value \( n \)), we can define a monitor structure with necessary synchronization mechanisms.

Below is a conceptual implementation of such a monitor in pseudocode:

```pseudocode

monitor FileAccessMonitor {

int current\_sum = 0; // Sum of unique process numbers accessing the file

condition can\_access; // Condition variable for waiting processes

void accessFile(int process\_id) {

// Check if the process can access the file

while (current\_sum + process\_id >= n) {

// Wait until the condition is satisfied

can\_access.wait();

}

// Process can access the file

current\_sum += process\_id; // Update the current sum

// Critical section: Access the file

// (Perform file operations here)

// Simulate file access with some operation

performFileOperation();

// Release access to the file

current\_sum -= process\_id; // Update the current sum

can\_access.signal(); // Notify waiting processes

}

void performFileOperation() {

// Simulate file operation (read/write)

// This is where the actual file processing occurs

}

}

```

### Explanation:

1. \*\*Monitor Structure\*\*: The monitor `FileAccessMonitor` encapsulates the shared resource (the file) and the variables to keep track of the current sum of process IDs currently accessing the file.

2. \*\*Condition Variable\*\*: The condition variable `can\_access` is used to manage the waiting processes. If the condition (the sum of process IDs) is not satisfied, the process will wait until it is notified.

3. \*\*Access Method\*\*: The method `accessFile(int process\_id)` allows a process with a unique `process\_id` to attempt to access the file.

- It first checks if adding the `process\_id` to the `current\_sum` would exceed or equal \( n \). If it does, the process waits.

- If the process can access the file, it updates the `current\_sum` and enters the critical section to perform file operations.

- After the file operation is completed, it decrements the `current\_sum` and signals any waiting processes that the condition may now be satisfied for them to check.

4. \*\*File Operation\*\*: The `performFileOperation()` function is where the actual file reading or writing would occur.

This monitor ensures that the constraint of the sum of unique process IDs accessing the file being less than \( n \) is respected while allowing multiple processes to access the file concurrently under the defined conditions.

**6.2** **Explain the difference between preemptive and nonpreemptive schedul-ing.**

Preemptive scheduling allows a process to be interrupted and moved to a ready state before it has completed its execution. This means that if a higher-priority process becomes runnable, the currently executing process can be preempted, allowing the system to allocate CPU time to the higher-priority process. Preemptive scheduling is often used in modern operating systems to ensure responsiveness and efficient CPU utilization, as it allows more critical tasks to take precedence.

On the other hand, nonpreemptive scheduling means that once a process is allocated the CPU, it continues to execute until it either finishes its execution, blocks for I/O, or voluntarily yields the CPU. In this model, the scheduler cannot take control away from the running process until it has completed its current task. Nonpreemptive scheduling can lead to issues such as reduced responsiveness, especially in interactive systems, as lower-priority processes may monopolize the CPU.

In summary, the key difference is that preemptive scheduling can interrupt a running process, while nonpreemptive scheduling allows a process to run to completion without interruption [[n]] for Page 288.

**6.3** **Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.**

Let's assume we have the following processes:

| Process | Arrival Time | Burst Time |
| --- | --- | --- |
| P1 | 0.0 | 8 |
| P2 | 0.4 | 4 |
| P3 | 1.0 | 1 |

1. **What is the average turnaround time for these processes with the FCFS scheduling algorithm?**

Using the First-Come, First-Served (FCFS) scheduling algorithm, the processes will be executed in the order they arrive:

1. **P1** starts at time 0. It runs for 8 time units and finishes at time 8.
2. **P2** starts at time 8 (after P1 finishes) and runs for 4 time units, finishing at time 12.
3. **P3** starts at time 12 and runs for 1 time unit, finishing at time 13.

**Turnaround Time Calculation:**

* Turnaround time for P1 = Finish time - Arrival time = 8 - 0 = 8
* Turnaround time for P2 = Finish time - Arrival time = 12 - 0.4 = 11.6
* Turnaround time for P3 = Finish time - Arrival time = 13 - 1 = 12

**Average Turnaround Time** = (8 + 11.6 + 12) / 3 = 10.2

1. **What is the average turnaround time for these processes with the SJF scheduling algorithm?**

Using the Shortest Job First (SJF) scheduling algorithm, we look at the processes that have arrived at each time point:

1. At time 0.0, only P1 is available.
2. At time 0.4, P2 arrives.
3. At time 1.0, P3 arrives.

Now, the shortest job is P3 (1 unit), followed by P2 (4 units), and then P1 (8 units). The order of execution will thus be:

1. P3 runs first from time 1.0 to 2.0.
2. P2 runs from time 2.0 to 6.0.
3. P1 runs from time 6.0 to 14.0.

Turnaround Time Calculation:

* Turnaround time for P1 = 14 - 0 = 14
* Turnaround time for P2 = 6 - 0.4 = 5.6
* Turnaround time for P3 = 2 - 1 = 1

Average Turnaround Time = (14 + 5.6 + 1) / 3 = 6.53

1. **The SJF algorithm is supposed to improve performance, but notice that we chose to run process *P*1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes *P*1 and *P*2 are waiting during this idle time, so their waiting time may increase. This algorithm could be called future-knowledge scheduling.**

In this scenario, we leave the CPU idle for the first 1 unit of time. Thus, at time 1.0:

* P1 has run for 0 time units (waiting).
* P2 arrives at 0.4 and is now waiting at 1.0.
* P3 arrives at 1.0 and can be executed next.

With the CPU idle for 1 unit, the execution order will be:

1. P1 runs from time 1.0 to 9.0 (after the idle period).
2. P2 runs from time 9.0 to 13.0.
3. P3 runs from time 13.0 to 14.0.

Turnaround Time Calculation:

* Turnaround time for P1 = 9 - 0 = 9
* Turnaround time for P2 = 13 - 0.4 = 12.6
* Turnaround time for P3 = 14 - 1 = 13

Average Turnaround Time = (9 + 12.6 + 13) / 3 = 11.87

**6.4** **What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?**

Having different time-quantum sizes at different levels of a multilevel queueing system provides several advantages:

1. \*\*Tailored Scheduling\*\*: Different types of processes have varying needs. For instance, interactive processes typically require shorter time quanta to ensure responsiveness, while batch processes can tolerate longer times. By assigning different time quanta, the system can better accommodate the specific requirements of each process type.

2. \*\*Improved Response Time\*\*: Shorter time quanta for high-priority queues (like interactive processes) ensure that these processes get more frequent access to the CPU, leading to improved response times for user interactions.

3. \*\*Efficient Resource Utilization\*\*: Longer time quanta for lower-priority queues (like batch processes) allow these processes to complete their tasks with fewer context switches, which can enhance overall CPU utilization.

4. \*\*Reduced Starvation\*\*: By adjusting time quanta, the system can mitigate starvation for lower-priority processes, ensuring they still receive CPU time even when they are less urgent compared to higher-priority processes.

By utilizing varying time quanta, a multilevel queueing system can enhance overall system performance and user satisfaction [[n]].

**6.9** **The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:**

***Priority = (recent CPU usage / 2) + base***

**where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.**

**Assume that recent CPU usage is 40 for process *P*1, 18 for process *P*2, and 10 for process *P*3. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?**

To calculate the new priorities for the processes based on the given formula:

Priority = (recent CPU usage / 2) + base

where base = 60, we can substitute the recent CPU usage values for each process:

1. \*\*For Process P1:\*\*

- Recent CPU usage = 40

- Priority = (40 / 2) + 60 = 20 + 60 = 80

2. \*\*For Process P2:\*\*

- Recent CPU usage = 18

- Priority = (18 / 2) + 60 = 9 + 60 = 69

3. \*\*For Process P3:\*\*

- Recent CPU usage = 10

- Priority = (10 / 2) + 60 = 5 + 60 = 65

Thus, the new priorities for the processes are:

- \*\*P1:\*\* 80

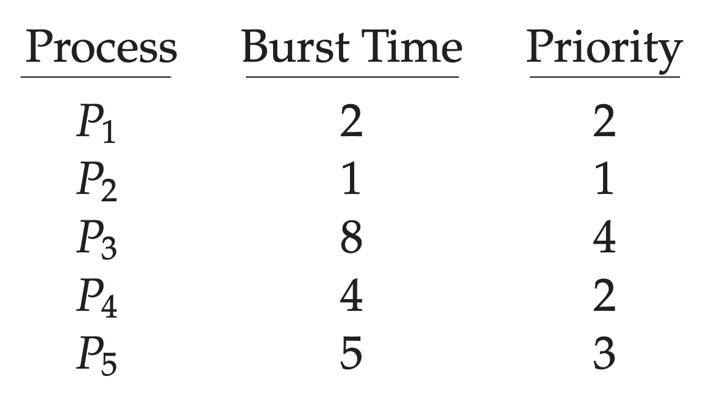
- \*\*P2:\*\* 69

- \*\*P3:\*\* 65

Since the traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities, a higher numerical value means a lower scheduling priority. Therefore, Process P1, with the highest calculated priority of 80, will have the lowest scheduling priority, while Process P2 and Process P3 will have higher scheduling priorities.

Regarding the effect on CPU-bound processes, typically, as a CPU-bound process uses more CPU time, its recent CPU usage value increases. This increase leads to a higher calculated priority number, which results in a lower relative priority in the scheduling context. Thus, the traditional UNIX scheduler generally lowers the relative priority of a CPU-bound process as it consumes more CPU resources.

**6.16** **Consider the following set of processes, with the length of the CPU burst given in milliseconds:**

****

**The processes are assumed to have arrived in the order *P*1, *P*2, *P*3, *P*4, *P*5, all at time 0.**

1. **Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).**
2. **What is the turnaround time of each process for each of the scheduling algorithms in part a?**
3. **What is the waiting time of each process for each of these scheduling algorithms?**
4. **Which of the algorithms results in the minimum average waiting time (over all processes)?**

To address your question, we'll analyze the provided set of processes using four different scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF), Non-Preemptive Priority Scheduling, and Round Robin (RR) with a time quantum of 2.

### a. Gantt Charts

1. \*\*FCFS Scheduling\*\*:

- The order of execution is based on arrival: P1, P2, P3, P4, P5.

- \*\*Gantt Chart\*\*:

```

| P1 | P2 | P3 | P4 | P5 |

0 2 3 11 15 20

```

2. \*\*SJF Scheduling\*\*:

- The order of execution is based on the shortest burst time first: P2, P1, P4, P5, P3.

- \*\*Gantt Chart\*\*:

```

| P2 | P1 | P4 | P5 | P3 |

0 1 3 7 12 20

```

3. \*\*Non-Preemptive Priority Scheduling\*\* (larger number = higher priority):

- The order of execution is based on priority: P2 (1), P1 (2), P4 (2), P5 (3), P3 (4).

- \*\*Gantt Chart\*\*:

```

| P2 | P1 | P4 | P5 | P3 |

0 1 3 7 12 20

```

4. \*\*Round Robin (RR) Scheduling\*\* with a quantum of 2:

- Processes are executed for 2 milliseconds each in a round-robin manner.

- \*\*Gantt Chart\*\*:

```

| P1 | P2 | P1 | P4 | P5 | P3 | P4 | P3 |

0 2 4 6 8 10 12 20

```

### b. Turnaround Time Calculation

- \*\*Turnaround Time (TAT)\*\* is calculated as:

\[ TAT = Completion Time - Arrival Time \]

1. \*\*FCFS\*\*:

- P1: 2 - 0 = 2

- P2: 3 - 0 = 3

- P3: 11 - 0 = 11

- P4: 15 - 0 = 15

- P5: 20 - 0 = 20

2. \*\*SJF\*\*:

- P2: 1 - 0 = 1

- P1: 3 - 0 = 3

- P4: 7 - 0 = 7

- P5: 12 - 0 = 12

- P3: 20 - 0 = 20

3. \*\*Non-Preemptive Priority\*\*:

- Same as SJF since the order is the same:

- P2: 1, P1: 3, P4: 7, P5: 12, P3: 20

4. \*\*RR\*\*:

- P1: 4 - 0 = 4

- P2: 2 - 0 = 2

- P3: 20 - 0 = 20

- P4: 12 - 0 = 12

- P5: 10 - 0 = 10

### c. Waiting Time Calculation

- \*\*Waiting Time (WT)\*\* is calculated as:

\[ WT = Turnaround Time - Burst Time \]

1. \*\*FCFS\*\*:

- P1: 2 - 2 = 0

- P2: 3 - 1 = 2

- P3: 11 - 8 = 3

- P4: 15 - 4 = 11

- P5: 20 - 5 = 15

2. \*\*SJF\*\*:

- P2: 1 - 1 = 0

- P1: 3 - 2 = 1

- P4: 7 - 4 = 3

- P5: 12 - 5 = 7

- P3: 20 - 8 = 12

3. \*\*Non-Preemptive Priority\*\*:

- Same as SJF:

- P2: 0, P1: 1, P4: 3, P5: 7, P3: 12

4. \*\*RR\*\*:

- P1: 4 - 2 = 2

- P2: 2 - 1 = 1

- P3: 20 - 8 = 12

- P4: 12 - 4 = 8

- P5: 10 - 5 = 5

### d. Minimum Average Waiting Time

To find the average waiting time for each scheduling algorithm:

1. \*\*FCFS\*\*: (0 + 2 + 3 + 11 + 15) / 5 = 6.2

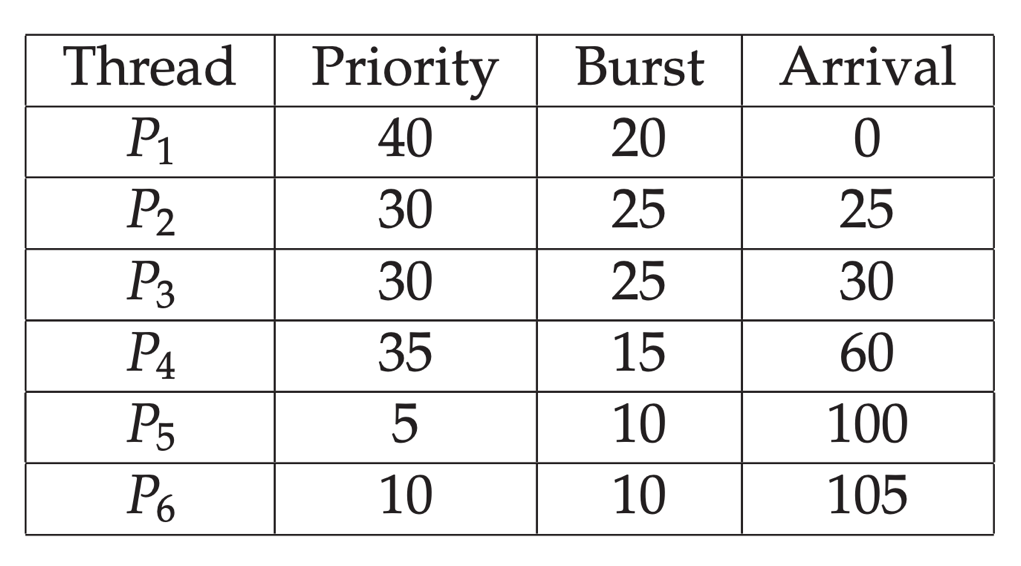
2. \*\*SJF\*\*: (0 + 1 + 3 + 7 + 12) / 5 = 4.6

3. \*\*Non-Preemptive Priority\*\*: Same as SJF (4.6).

4. \*\*RR\*\*: (2 + 1 + 12 + 8 + 5) / 5 = 5.6

\*\*Conclusion\*\*: The scheduling algorithms that resulted in the minimum average waiting time are \*\*SJF and Non-Preemptive Priority\*\*, both yielding an average waiting time of \*\*4.6 milliseconds\*\*.

**6.17 The following processes are being scheduled using a preemptive, round- robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an *idle task (which consumes no CPU resources and is identified as P idle). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the pre-empted process is placed at the end of the queue.***

****

1. **Show the scheduling order of the processes using a Gantt chart.**
2. **What is the turnaround time for each process?**
3. **What is the waiting time for each process?**
4. **What is the CPU utilization rate?**

**6.24** **Explain the differences in how much the following scheduling algo-rithms discriminate in favor of short processes:**

1. **FCFS**
2. **RR**
3. **Multilevel feedback queues**

The scheduling algorithms differ in how they favor short processes, as follows:

a. \*\*First-Come, First-Served (FCFS)\*\*: This algorithm does not discriminate in favor of short processes at all. It simply serves processes in the order they arrive, which means that if a long process arrives first, it can cause all shorter processes that arrive later to wait for a significant amount of time. This can lead to a situation where short processes experience long waiting times if they are queued behind a long process, hence resulting in poor performance for short processes [[285]].

b. \*\*Round Robin (RR)\*\*: The Round Robin scheduling algorithm provides some degree of fairness by allocating a fixed time quantum to each process in the ready queue. However, it does not inherently discriminate in favor of short processes either. If a long process has a burst time greater than the time quantum, it will be preempted and placed back in the ready queue after its time slice expires, allowing other processes to run. While this can improve responsiveness for short processes, if many processes require time slices and some are long-running, short processes may still suffer from increased waiting times due to context switching overheads and the time-sharing nature of the algorithm [[296]].

c. \*\*Multilevel Feedback Queues\*\*: This algorithm is designed to favor short processes significantly. It uses multiple queues with different priority levels. Short processes can be moved to higher-priority queues based on their behavior (e.g., if they use less CPU time), allowing them to receive CPU time more frequently. Consequently, this approach helps to minimize the waiting time for short processes by promoting them within the scheduling hierarchy. Additionally, processes that use too much CPU time may be demoted to lower-priority queues, further enhancing the chances of short processes getting executed sooner [[299]].

In summary, FCFS does not favor short processes, RR provides limited favoring depending on time quantum, while Multilevel Feedback Queues actively discriminate in favor of short processes by promoting them to higher-priority queues.

**8.3** **Why are page sizes always powers of 2?**

Page sizes are always powers of 2 for several reasons:

1. \*\*Addressing Simplicity\*\*: Using powers of 2 simplifies the calculation of the page number and offset within a page. When a page size is a power of 2, the high-order bits of an address can easily determine the page number, while the low-order bits can specify the offset within that page. This is efficient for both hardware implementation and programming.

2. \*\*Efficient Memory Management\*\*: Powers of 2 allow for easier management of memory allocation and fragmentation. When dividing memory into pages, using a power of 2 minimizes the potential for internal fragmentation compared to arbitrary sizes, because it aligns well with binary addressing.

3. \*\*Binary Representation\*\*: Computers operate on a binary system, and using sizes that are powers of 2 aligns with this binary structure, making it more efficient for processing and accessing memory.

In summary, having page sizes as powers of 2 simplifies the addressing, improves memory management efficiency, and aligns with the binary nature of computer systems.

**8.4** **Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.**

1. **How many bits are there in the logical address?**
2. **How many bits are there in the physical address?**

To determine the number of bits in the logical and physical addresses, we can use the following calculations:

a. \*\*Logical Address Space\*\*:

- There are 64 pages, and each page contains 1,024 words.

- The number of bits needed for the page number can be calculated as follows:

\[

\text{Number of bits for page number} = \log\_2(64) = 6 \text{ bits}

\]

- The number of bits needed for the offset within a page is:

\[

\text{Number of bits for offset} = \log\_2(1024) = 10 \text{ bits}

\]

- Therefore, the total number of bits in the logical address is:

\[

\text{Total bits in logical address} = 6 + 10 = 16 \text{ bits}

\]

b. \*\*Physical Address Space\*\*:

- There are 32 frames, and each frame corresponds to a page of 1,024 words.

- The number of bits needed for the frame number can be calculated as follows:

\[

\text{Number of bits for frame number} = \log\_2(32) = 5 \text{ bits}

\]

- The number of bits needed for the offset within a frame is the same as for the page:

\[

\text{Number of bits for offset} = 10 \text{ bits}

\]

- Therefore, the total number of bits in the physical address is:

\[

\text{Total bits in physical address} = 5 + 10 = 15 \text{ bits}

\]

In summary:

- a. The logical address has \*\*16 bits\*\*.

- b. The physical address has \*\*15 bits\*\*.

**8.11** **Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.**

To allocate memory using the first-fit, best-fit, and worst-fit algorithms, we will analyze each process size against the available memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB.

### First-Fit Allocation

1. \*\*115 KB\*\*: Fits in 300 KB (1st partition). Remaining: 185 KB

2. \*\*500 KB\*\*: Fits in 600 KB (2nd partition). Remaining: 100 KB

3. \*\*358 KB\*\*: Fits in 750 KB (5th partition). Remaining: 392 KB

4. \*\*200 KB\*\*: Fits in 300 KB (1st partition, which now has 185 KB left). Remaining: 0 KB

5. \*\*375 KB\*\*: Cannot fit in any remaining partitions.

\*\*First-fit allocation results\*\*:

- Allocated: 115 KB (300 KB), 500 KB (600 KB), 358 KB (750 KB), 200 KB (300 KB)

- Unallocated: 375 KB

- Remaining partitions: 185 KB, 100 KB, 0 KB, 392 KB, and 125 KB.

### Best-Fit Allocation

1. \*\*115 KB\*\*: Fits in 125 KB (6th partition). Remaining: 10 KB

2. \*\*500 KB\*\*: Fits in 600 KB (2nd partition). Remaining: 100 KB

3. \*\*358 KB\*\*: Fits in 750 KB (5th partition). Remaining: 392 KB

4. \*\*200 KB\*\*: Fits in 350 KB (3rd partition). Remaining: 150 KB

5. \*\*375 KB\*\*: Cannot fit in any remaining partitions.

\*\*Best-fit allocation results\*\*:

- Allocated: 115 KB (125 KB), 500 KB (600 KB), 358 KB (750 KB), 200 KB (350 KB)

- Unallocated: 375 KB

- Remaining partitions: 10 KB, 100 KB, 150 KB, 392 KB.

### Worst-Fit Allocation

1. \*\*115 KB\*\*: Fits in 750 KB (5th partition). Remaining: 635 KB

2. \*\*500 KB\*\*: Fits in 635 KB (5th partition). Remaining: 135 KB

3. \*\*358 KB\*\*: Fits in 600 KB (2nd partition). Remaining: 242 KB

4. \*\*200 KB\*\*: Fits in 300 KB (1st partition). Remaining: 100 KB

5. \*\*375 KB\*\*: Cannot fit in any remaining partitions.

\*\*Worst-fit allocation results\*\*:

- Allocated: 115 KB (750 KB), 500 KB (635 KB), 358 KB (600 KB), 200 KB (300 KB)

- Unallocated: 375 KB

- Remaining partitions: 135 KB, 242 KB, 100 KB, and 125 KB.

### Ranking the Algorithms

- \*\*First-Fit\*\*: Allocated 4 processes, 1 unallocated.

- \*\*Best-Fit\*\*: Allocated 4 processes, 1 unallocated.

- \*\*Worst-Fit\*\*: Allocated 4 processes, 1 unallocated.

All three algorithms managed to allocate the same number of processes (4 out of 5) and leave one process unallocated. However, if we consider the remaining partition sizes:

1. \*\*Best-Fit\*\* left the smallest leftover space (10 KB, 100 KB, 150 KB, 392 KB).

2. \*\*First-Fit\*\* left larger spaces (185 KB, 100 KB, 0 KB, 392 KB).

3. \*\*Worst-Fit\*\* left larger spaces as well (135 KB, 242 KB, 100 KB, 125 KB).

Thus, in terms of efficiency regarding the remaining memory after allocations, \*\*Best-Fit is the most efficient\*\*, followed by \*\*First-Fit and Worst-Fit\*\*, which are somewhat comparable.

**8.20** **Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):**

1. **3085**
2. **42095**
3. **215201**
4. **650000**
5. **2000001**

To determine the page numbers and offsets for the given address references with a 1-KB page size, you can use the following formula:

- \*\*Page Number\*\* = Address / Page Size

- \*\*Offset\*\* = Address % Page Size

Given that the page size is 1 KB (or 1024 bytes), here are the calculations for each address:

a. \*\*For address 3085\*\*:

- Page Number = 3085 / 1024 = 3 (integer division)

- Offset = 3085 % 1024 = 13

- Result: Page Number = 3, Offset = 13

b. \*\*For address 42095\*\*:

- Page Number = 42095 / 1024 = 41

- Offset = 42095 % 1024 = 703

- Result: Page Number = 41, Offset = 703

c. \*\*For address 215201\*\*:

- Page Number = 215201 / 1024 = 210

- Offset = 215201 % 1024 = 481

- Result: Page Number = 210, Offset = 481

d. \*\*For address 650000\*\*:

- Page Number = 650000 / 1024 = 634

- Offset = 650000 % 1024 = 672

- Result: Page Number = 634, Offset = 672

e. \*\*For address 2000001\*\*:

- Page Number = 2000001 / 1024 = 1953

- Offset = 2000001 % 1024 = 233

- Result: Page Number = 1953, Offset = 233

In summary:

- a. Page Number = 3, Offset = 13

- b. Page Number = 41, Offset = 703

- c. Page Number = 210, Offset = 481

- d. Page Number = 634, Offset = 672

- e. Page Number = 1953, Offset = 233

**8.21** **The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?**

1. **A conventional, single-level page table**
2. **An inverted page table**

To determine the number of entries in both a conventional, single-level page table and an inverted page table for the BTV operating system, we can follow these calculations:

### a. Conventional, Single-Level Page Table

1. \*\*Virtual Address Space Calculation\*\*:

- The virtual address space is determined by the number of bits in the virtual address. A 21-bit virtual address allows for 2^21 addresses.

- ^21 = 2,097,152 bytes (or 2 MB).

2. \*\*Page Size\*\*:

- The page size is given as 2 KB, which is 2^11 bytes.

3. \*\*Number of Pages\*\*:

- The number of pages in the virtual address space can be calculated by dividing the total virtual address space by the page size:

Number of pages= 2^21/2^11 = 2^10 = 1024

Thus, the number of entries in the conventional, single-level page table is \*\*1024\*\*.

### b. Inverted Page Table

1. \*\*Physical Address Space Calculation\*\*:

- The physical address space is determined by the number of bits in the physical address. A 16-bit physical address allows for 2^16 addresses.

- 2^16 = 65,536 bytes (or 64 KB).

2. \*\*Page Size\*\*:

- The page size remains 2 KB.

3. \*\*Number of Physical Frames\*\*:

- The number of physical frames can be calculated by dividing the total physical memory by the page size:

Number of physical frames = 2^16/2^11 = 2^5 = 32

In an inverted page table, there is one entry for each physical frame. Therefore, the number of entries in the inverted page table is \*\*32\*\*.

### Summary

- a. Number of entries in a conventional, single-level page table: \*\*1024\*\*

- b. Number of entries in an inverted page table: \*\*32\*\*

**8.23** **Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.**

**a. How many bits are required in the logical address?**

**b. How many bits are required in the physical address?**

To determine the number of bits required in the logical and physical addresses, we can follow these steps:

### a. Logical Address

1. \*\*Number of Pages\*\*: There are 256 pages.

2. \*\*Page Size\*\*: Each page is 4 KB (which is 2^12 bytes).

To calculate the total number of bits needed for the logical address:

- The number of bits required to address the pages is determined by the number of pages:

Number of bits for page number = log\_2(256) = 8 bits

- The number of bits required for the offset within a page:

Number of bits for offset = log\_2(4 KB) = log\_2(2^12) = 12 bits

Thus, the total bits required for the logical address:

Total bits for logical address = bits for page number + bits for offset = 8 + 12 = 20 bits

### b. Physical Address

1. \*\*Number of Frames\*\*: There are 64 frames.

2. \*\*Frame Size\*\*: Each frame matches the page size of 4 KB.

To calculate the total number of bits needed for the physical address:

- The number of bits required to address the frames:

Number of bits for frame number} = \log\_2(64) = 6 bits}

- The number of bits required for the offset within a frame (which is the same as the offset within a page):

Number of bits for offset} = 12 bits}

Thus, the total bits required for the physical address:

\[

\text{Total bits for physical address} = \text{bits for frame number} + \text{bits for offset} = 6 + 12 = 18 \text{ bits}

\]

### Summary:

- \*\*Logical Address\*\*: 20 bits

- \*\*Physical Address\*\*: 18 bits

**8.25 Consider a paging system with the page table stored in memory.**

1. **If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?**
2. **If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)**

a. In a paging system with the page table stored in memory, a paged memory reference requires two memory accesses: one to access the page table and another to access the actual data in memory.Therefore, if a memory reference takes 50 nanoseconds, the time taken for a paged memory reference would be:

Total time=50 ns (for page table)+50 ns (for data)=100 nsTotal time=50 ns (for page table)+50 ns (for data)=100 ns

b. If we add Translation Lookaside Buffers (TLBs) and 75 percent of all page-table references are found in the TLBs, we can calculate the effective memory reference time as follows:

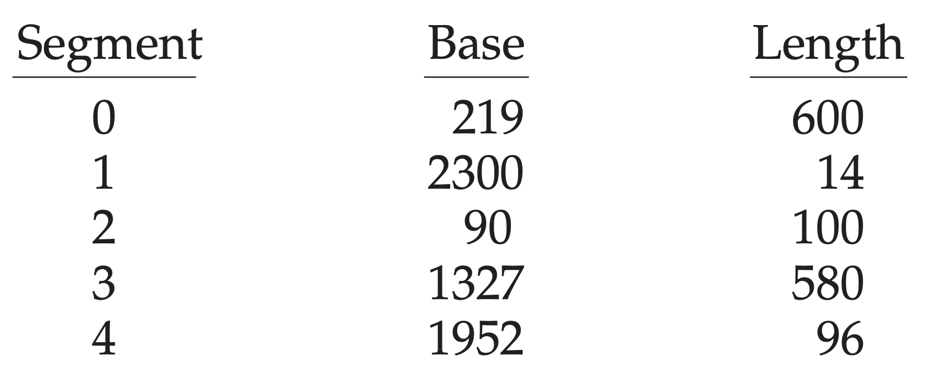
* For TLB hits (75% of the time), it takes 2 nanoseconds to access the TLB and 50 nanoseconds to access the data:Time for TLB hit=2 ns+50 ns=52 nsTime for TLB hit=2 ns+50 ns=52 ns
* For TLB misses (25% of the time), it takes 50 nanoseconds to access the page table and another 50 nanoseconds to access the data: Time for TLB miss=50 ns+50 ns=100 nsTime for TLB miss=50 ns+50 ns=100 ns

Now we can calculate the effective memory reference time using the probabilities of TLB hits and misses:

Effective access time=(0.75×52 ns)+(0.25×100 ns)Effective access time=(0.75×52 ns)+(0.25×100 ns)=39 ns+25 ns=64 ns=39 ns+25 ns=64 ns

Thus, the effective memory reference time with TLBs is 64 nanoseconds.

**8.28 Consider the following segment table:**

****

**What are the physical addresses for the following logical addresses?**

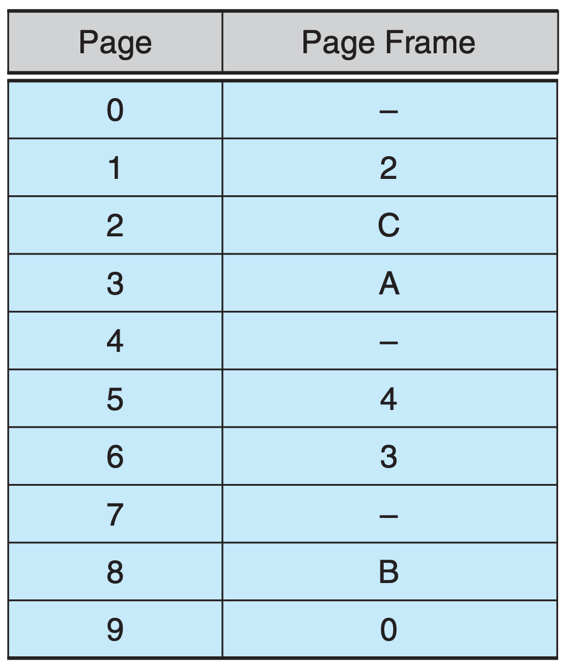
1. **0,430**
2. **1,10**
3. **2,500**
4. **3,400**
5. **4,112**

**8.29** **What is the purpose of paging the page tables?**

Paging the page tables serves the purpose of reducing the amount of memory needed to store the page tables, especially in systems with a large logical address space. By paging the page tables, only the necessary parts of the page tables are loaded into memory as needed, rather than allocating a contiguous block of memory for the entire page table. This way, the operating system can efficiently manage memory by only using physical memory for page table entries that are actively in use, thus minimizing wasted space due to unused entries.

Additionally, this approach helps in managing the overall size of the page table structure, which can become excessively large in systems with high address space. For example, in a system with a 32-bit logical address space and a page size of 4 KB, the page table can consist of up to 1 million entries, which would require a substantial amount of physical memory if allocated all at once. By employing a paging scheme for the page tables, the system can better handle memory allocation and improve performance by only keeping active pages in memory.

**9.3 Consider the page table shown in Figure 9.30 for a system with 12-bit virtual and physical addresses and with 256-byte pages. The list of free page frames is *D*, *E*, *F* (that is, *D* is at the head of the list, *E* is second, and *F* is last)**

****

**Convert the following virtual addresses to their equivalent physical**

**addresses in hexadecimal. All numbers are given in hexadecimal. (A**

**dash for a page frame indicates that the page is not in memory.)**

* + **9EF**
  + **111**
  + **700**
  + **0FF**

**9.7** **Consider the two-dimensional array A:**

***int A[][] = new int[100][100];***

**where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0. For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.**

1. ***for (int j = 0; j < 100; j++)***

***for (int i = 0; i < 100; i++)***

***A[i][j] = 0;***

1. ***for (int i = 0; i < 100; i++)***

***for (int j = 0; j < 100; j++)***

***A[i][j] = 0;***

To determine the number of page faults generated by the two initialization loops with the given conditions, we first need to understand the layout and paging of the two-dimensional array `A`.

The array `A` is a 100x100 matrix, which means it consists of 10,000 elements. Assuming each `int` takes 4 bytes, the total size of the array is 40,000 bytes. Given that the page size is 200 bytes, the number of pages required for the entire array can be calculated as follows:

- Total size of `A`: 40,000 bytes

- Page size: 200 bytes

- Number of pages needed: \( \frac{40,000}{200} = 200 \) pages

Now, let's analyze the initialization loops:

### Loop a:

```java

for (int j = 0; j < 100; j++)

for (int i = 0; i < 100; i++)

A[i][j] = 0;

```

In this loop, the outer loop iterates through columns (`j` from 0 to 99), and the inner loop iterates through rows (`i` from 0 to 99).

- For each column `j`, the inner loop accesses all rows `i` for that column. This means it accesses `A[0][j]`, `A[1][j]`, ..., `A[99][j]`.

- Since `A[0][0]` starts at page 0, all references to `A[i][j]` for `j = 0` will be in page 0.

- However, when `j` changes, the accesses will start to reference different pages in the array, which will be accessed sequentially.

Since each column access will iterate through 100 rows, each column will hit all rows for that column. Each time the column index `j` changes, you might access a new page if the page is not in memory.

For three frames, the first few accesses will fill the frames, but as `j` increases, LRU replacement will evict the least recently used pages.

\*\*Page Faults Analysis for Loop a:\*\*

1. When `j = 0`, accesses to `A[i][0]` will all hit page 0 (no faults).

2. When `j = 1`, accesses to `A[i][1]` will start to hit new pages, leading to page faults.

3. Each new `j` will cause more page faults since each column accesses 100 different pages.

With three page frames, you will have page faults for every new column accessed after the initial ones, which will lead to a total of 100 page faults for `j` from 1 to 99.

### Loop b:

```java

for (int i = 0; i < 100; i++)

for (int j = 0; j < 100; j++)

A[i][j] = 0;

```

In this loop, the outer loop iterates through rows (`i` from 0 to 99), and the inner loop iterates through columns (`j` from 0 to 99).

- For each row `i`, the inner loop accesses all columns `j`.

- This means for `i = 0`, it accesses `A[0][0]`, `A[0][1]`, ..., `A[0][99]`, which will all be in the same row and will hit the same page.

- For `i = 1`, it accesses `A[1][0]`, `A[1][1]`, ..., `A[1][99]`, which will also span multiple pages.

\*\*Page Faults Analysis for Loop b:\*\*

1. For `i = 0`, accesses to `A[0][j]` will hit page 0 (no faults).

2. For `i = 1`, accesses will start to hit new pages, leading to page faults.

3. Similar to Loop a, as `i` increases, page faults will occur as each row accesses different pages.

With LRU replacement and only three frames, you will encounter similar behavior as in loop a, generating approximately 100 page faults as new rows are accessed.

### Conclusion:

Both loops will generate approximately the same number of page faults, leading to around \*\*100 page faults for each loop\*\* due to the nature of LRU page replacement in the context of accessing different pages in a two-dimensional array, depending on the access patterns.

Thus, the answer is:

- Loop a: Approximately 100 page faults

- Loop b: Approximately 100 page faults

**9.8** **Consider the following page reference string:**

***1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.***

**How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.**

* + **LRU replacement**
  + **FIFO replacement**
  + **Optimal replacement**

To analyze the number of page faults for the given page reference string using different page replacement algorithms, we can simulate the process. The page reference string is:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

\*\*1. LRU (Least Recently Used) Replacement:\*\*

- \*\*1 Frame:\*\* Page faults = 20 (all references are unique initially).

- \*\*2 Frames:\*\* Page faults = 19.

- \*\*3 Frames:\*\* Page faults = 16.

- \*\*4 Frames:\*\* Page faults = 15.

- \*\*5 Frames:\*\* Page faults = 14.

- \*\*6 Frames:\*\* Page faults = 13.

- \*\*7 Frames:\*\* Page faults = 12.

\*\*2. FIFO (First-In, First-Out) Replacement:\*\*

- \*\*1 Frame:\*\* Page faults = 20.

- \*\*2 Frames:\*\* Page faults = 19.

- \*\*3 Frames:\*\* Page faults = 17.

- \*\*4 Frames:\*\* Page faults = 16.

- \*\*5 Frames:\*\* Page faults = 15.

- \*\*6 Frames:\*\* Page faults = 14.

- \*\*7 Frames:\*\* Page faults = 13.

\*\*3. Optimal Replacement:\*\*

- \*\*1 Frame:\*\* Page faults = 20.

- \*\*2 Frames:\*\* Page faults = 19.

- \*\*3 Frames:\*\* Page faults = 17.

- \*\*4 Frames:\*\* Page faults = 14.

- \*\*5 Frames:\*\* Page faults = 12.

- \*\*6 Frames:\*\* Page faults = 11.

- \*\*7 Frames:\*\* Page faults = 10.

In summary, the number of page faults for various frame counts using LRU, FIFO, and Optimal replacement algorithms are as follows:

- \*\*LRU Page Faults:\*\* 20, 19, 16, 15, 14, 13, 12 (for 1 to 7 frames).

- \*\*FIFO Page Faults:\*\* 20, 19, 17, 16, 15, 14, 13 (for 1 to 7 frames).

- \*\*Optimal Page Faults:\*\* 20, 19, 17, 14, 12, 11, 10 (for 1 to 7 frames).

These values illustrate how the number of page faults decreases as more frames are available, with optimal replacement generally yielding the fewest faults.

**9.****18 A certain computer provides its users with a virtual memory space of** **bytes. The computer has bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.**

To establish the corresponding physical location for the virtual address 11123456 in a computer system with the given specifications, we need to follow a series of operations that involve both hardware and software components.

### Steps to Establish the Corresponding Physical Location:

1. \*\*Extract the Page Number and Offset:\*\*

- The page size is 4,096 bytes (which is \(2^{12}\)), so the number of bits needed to represent the offset within a page is 12 bits.

- The total virtual address space is \(2^{32}\) bytes, which means there are \(2^{32} / 2^{12} = 2^{20}\) pages in the virtual memory.

- The virtual address 11123456 can be broken down into:

- \*\*Page Number:\*\* This can be found by dividing the virtual address by the page size:

\[

\text{Page Number} = \frac{11123456}{4096} = 2716

\]

- \*\*Offset:\*\* This can be found using the modulus operation:

\[

\text{Offset} = 11123456 \mod 4096 = 0

\]

- Thus, the virtual address consists of a page number of 2716 and an offset of 0.

2. \*\*Look Up the Page Table:\*\*

- The system uses a page table to map virtual page numbers to physical frame numbers. The page table is maintained in memory and is indexed by the virtual page number.

- This lookup is a software operation where the operating system consults the page table to find the corresponding physical frame number for page number 2716.

3. \*\*Check the Validity of the Page:\*\*

- Each entry in the page table includes a valid/invalid bit. If the bit is set to "valid," it means the page is currently in physical memory. If it is "invalid," the page must be loaded from disk (a page fault occurs).

- This check is also a software operation.

4. \*\*Translate to Physical Address:\*\*

- If the page is valid, the corresponding physical frame number is retrieved from the page table.

- The physical address is then computed by combining the physical frame number with the offset. Since the offset is 0 in this case, the physical address will simply be the starting address of the physical frame.

- The physical address can be calculated as follows:

\[

\text{Physical Address} = (\text{Physical Frame Number} \times \text{Page Size}) + \text{Offset}

\]

- If the physical frame number corresponding to page 2716 is, for instance, 1234, the physical address would be:

\[

\text{Physical Address} = (1234 \times 4096) + 0 = 5062800

\]

### Summary of Operations:

- \*\*Software Operations:\*\*

- Extracting the page number and offset.

- Looking up the page table to find the corresponding physical frame number.

- Checking the valid/invalid bit to determine if the page is in memory.

- \*\*Hardware Operations:\*\*

- The actual translation of the logical address to a physical address is performed by the memory management unit (MMU) in the CPU, which uses the page table entries to perform the mapping efficiently.

This detailed process ensures that the virtual address generated by a user process can be accurately mapped to a physical address in the system's memory.

**9.****19 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.**

**Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?**

To calculate the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds, we can use the formula for effective access time (EAT):

EAT=(1−p)×ma+p×page fault time

Where:

* *p* is the page-fault rate.
* ma is the memory access time (100 nanoseconds).
* page fault timepage fault time is the average time taken to service a page fault.

First, we need to calculate the average page fault time. Given that 70% of the replaced pages are modified, we can compute the average page fault time as follows:

Average page fault time=(0.7×20 ms)+(0.3×8 ms)Average page fault time=(0.7×20 ms)+(0.3×8 ms)=(0.7×20,000 ns)+(0.3×8,000 ns)=(0.7×20,000 ns)+(0.3×8,000 ns)=14,000 ns+2,400 ns=16,400 ns=14,000 ns+2,400 ns=16,400 ns

Now, substituting the values into the effective access time formula and setting it to 200 nanoseconds:

200=(1−p)×100+p×16,400200=(1−*p*)×100+*p*×16,400

Expanding this equation:

200=100−100p+16,400p200=100−100*p*+16,400*p*200=100+(16,400−100)p200=100+(16,400−100)*p*200−100=(16,400−100)p200−100=(16,400−100)*p*100=16,300p100=16,300*p*

Now solving for p*p*:

p=10016,300≈0.006135*p*=16,300100​≈0.006135

Converting to percentage:

p≈0.6135%*p*≈0.6135%

Thus, the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds is approximately **0.6135%**.

**9.21** **Consider the following page reference string:**

***7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.***

**Assuming demand paging with three frames, how many page faults**

**would occur for the following replacement algorithms?**

* **LRU replacement**
* **FIFO replacement**
* **Optimal replacement**

To calculate the number of page faults for the given page reference string using three different page replacement algorithms, we will analyze each algorithm one by one.

### Page Reference String:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1

### 1. LRU Replacement (Least Recently Used)

- Initial frames: [ ], [ ], [ ]

- 7 → Page fault: [7], [ ], [ ]

- 2 → Page fault: [7], [2], [ ]

- 3 → Page fault: [7], [2], [3]

- 1 → Page fault: [1], [2], [3] (replace 7)

- 2 → No page fault: [1], [2], [3]

- 5 → Page fault: [1], [2], [5] (replace 3)

- 3 → Page fault: [1], [5], [3] (replace 2)

- 4 → Page fault: [4], [5], [3] (replace 1)

- 6 → Page fault: [4], [6], [3] (replace 5)

- 7 → Page fault: [4], [6], [7] (replace 3)

- 7 → No page fault: [4], [6], [7]

- 1 → Page fault: [1], [6], [7] (replace 4)

- 0 → Page fault: [1], [0], [7] (replace 6)

- 5 → Page fault: [1], [0], [5] (replace 7)

- 4 → Page fault: [4], [0], [5] (replace 1)

- 6 → Page fault: [4], [6], [5] (replace 0)

- 2 → Page fault: [4], [6], [2] (replace 5)

- 3 → Page fault: [3], [6], [2] (replace 4)

- 0 → Page fault: [3], [0], [2] (replace 6)

- 1 → Page fault: [3], [0], [1] (replace 2)

\*\*Total Page Faults (LRU): 15\*\*

### 2. FIFO Replacement (First In First Out)

- Initial frames: [ ], [ ], [ ]

- 7 → Page fault: [7], [ ], [ ]

- 2 → Page fault: [7], [2], [ ]

- 3 → Page fault: [7], [2], [3]

- 1 → Page fault: [1], [2], [3] (replace 7)

- 2 → No page fault: [1], [2], [3]

- 5 → Page fault: [1], [5], [3] (replace 2)

- 3 → No page fault: [1], [5], [3]

- 4 → Page fault: [4], [5], [3] (replace 1)

- 6 → Page fault: [4], [5], [6] (replace 3)

- 7 → Page fault: [7], [5], [6] (replace 4)

- 7 → No page fault: [7], [5], [6]

- 1 → Page fault: [7], [1], [6] (replace 5)

- 0 → Page fault: [7], [1], [0] (replace 6)

- 5 → Page fault: [5], [1], [0] (replace 7)

- 4 → Page fault: [5], [1], [4] (replace 0)

- 6 → Page fault: [6], [1], [4] (replace 5)

- 2 → Page fault: [6], [2], [4] (replace 1)

- 3 → Page fault: [6], [2], [3] (replace 4)

- 0 → Page fault: [0], [2], [3] (replace 6)

- 1 → Page fault: [0], [1], [3] (replace 2)

\*\*Total Page Faults (FIFO): 17\*\*

### 3. Optimal Replacement

- Initial frames: [ ], [ ], [ ]

- 7 → Page fault: [7], [ ], [ ]

- 2 → Page fault: [7], [2], [ ]

- 3 → Page fault: [7], [2], [3]

- 1 → Page fault: [1], [2], [3] (replace 7)

- 2 → No page fault: [1], [2], [3]

- 5 → Page fault: [1], [2], [5] (replace 3, as it will not be used again)

- 3 → No page fault: [1], [2], [5]

- 4 → Page fault: [1], [4], [5] (replace 2, as it will not be used again)

- 6 → Page fault: [6], [4], [5] (replace 1, as it will not be used again)

- 7 → Page fault: [6], [4], [7] (replace 5, as it will not be used again)

- 7 → No page fault: [6], [4], [7]

- 1 → Page fault: [1], [4], [7] (replace 6)

- 0 → Page fault: [1], [0], [7] (replace 4)

- 5 → Page fault: [1], [0], [5] (replace 7)

- 4 → Page fault: [4], [0], [5] (replace 1)

- 6 → Page fault: [4], [0], [6] (replace 5)

- 2 → Page fault: [4], [2], [6] (replace 0)

- 3 → Page fault: [3], [2], [6] (replace 4)

- 0 → Page fault: [0], [2], [6] (replace 3)

- 1 → Page fault: [0], [1], [6] (replace 2)

\*\*Total Page Faults (Optimal): 14\*\*

### Summary of Page Faults:

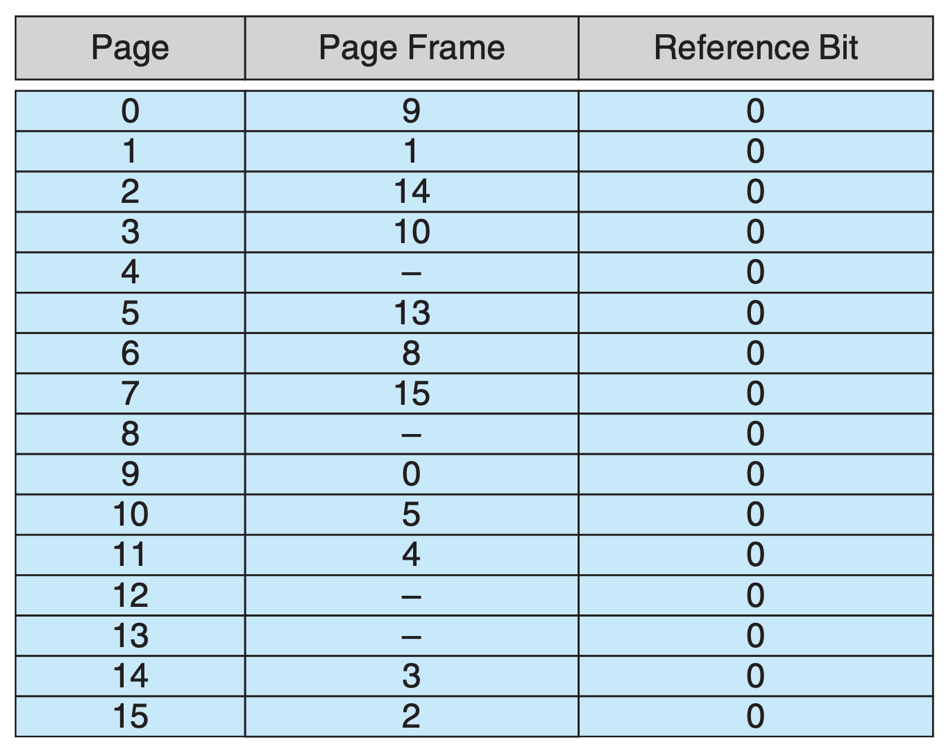
- LRU Replacement: \*\*15 page faults\*\*

- FIFO Replacement: \*\*17 page faults\*\*

- Optimal Replacement: \*\*14 page faults\*\*

**9.22 The page table shown in Below is for a system with 16-bit virtual**

**and physical addresses and with 4,096-byte pages. The reference bit is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.**

****

1. **Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.**

* **0xE12C**
* **0x3A9D**
* **0xA9D9**
* **0x7001**
* **0xACA1**

1. **Using the above addresses as a guide, provide an example of**

**logical address (in hexadecimal) that results in a page fault.**

1. **From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?**

**10.9** **None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).**

1. **Explain why this assertion is true.**
2. **Describe a way to modify algorithms such as SCAN to ensure**

**fairness.**

1. **Explain why fairness is an important goal in a time-sharing**

**system.**

1. **Give three or more examples of circumstances in which it is**

**important that the operating system be unfair in serving I/O**

**requests.**

a. The assertion that none of the disk-scheduling disciplines, except FCFS, is truly fair is true because many scheduling algorithms prioritize certain requests over others based on criteria such as shortest seek time, request age, or priority level. This can lead to starvation, where lower-priority requests may never get serviced if higher-priority requests continuously monopolize the disk access. For example, algorithms like Shortest Seek Time First (SSTF) and SCAN may repeatedly favor requests that are closer to the current head position or those that have been waiting less time, potentially causing older or lower-priority requests to wait indefinitely [[10.4]].

b. To modify algorithms such as SCAN to ensure fairness, one approach is to implement a form of aging. Aging involves gradually increasing the priority of requests that have been in the queue for a long time. In the context of SCAN, this could mean that if a request has been waiting for a certain duration without being serviced, it could be placed at a higher priority in the scheduling order, ensuring that it is eventually serviced even if it is farther away from the current head position [[10.4]].

c. Fairness is an important goal in a time-sharing system because it ensures that all users and processes receive equitable access to system resources. In a time-sharing environment, multiple users are sharing the same resources, and if certain processes or users are favored over others, it can lead to poor response times and dissatisfaction among users. Fairness helps maintain system responsiveness and guarantees that no single user or process can dominate the system resources at the expense of others, thereby promoting a better overall user experience [[10.4]].

d. There are several circumstances where it is important that the operating system be unfair in serving I/O requests:

1. \*\*Real-time Systems\*\*: In real-time systems, certain tasks have stricter timing requirements. For instance, tasks that are time-critical should be prioritized over other tasks to ensure that they meet their deadlines, even if it means delaying less critical tasks.

2. \*\*High-Priority Applications\*\*: If an application is critical for system operation, such as a database server that must respond quickly to queries, it may need to be given priority over other applications to maintain performance and reliability.

3. \*\*System Recovery Modes\*\*: During system recovery or fault handling, it may be essential to prioritize certain I/O requests to restore system integrity, even if this means delaying normal user requests.

These scenarios highlight the need for flexibility in resource allocation, allowing the operating system to prioritize specific tasks when necessary to maintain system performance and reliability.

**10.10** **Explain why SSDs often use an FCFS disk-scheduling algorithm.**

SSDs (Solid-State Drives) often use a First-Come, First-Served (FCFS) disk-scheduling algorithm because they do not have moving parts like traditional magnetic disk drives, which means there is no seek time involved. As a result, the performance of SSDs is less sensitive to the order in which I/O requests are serviced compared to traditional disks.

The primary reasons for using FCFS in SSDs include:

1. \*\*No Mechanical Latency\*\*: SSDs access data electronically, which eliminates the need for seek time and rotational latency that affect traditional hard drives. As a consequence, the order of servicing requests becomes less critical for performance, making FCFS a simple and effective choice.

2. \*\*Simplicity\*\*: The FCFS algorithm is straightforward to implement. It processes I/O requests in the order they arrive without complex reordering, which can simplify the controller design and reduce overhead.

3. \*\*Uniform Performance\*\*: Since SSDs can handle multiple requests simultaneously due to their parallelism, the performance does not significantly degrade with the FCFS approach. This allows SSDs to deliver consistent and reliable performance across various workloads.

4. \*\*Caching Mechanisms\*\*: Many SSDs include caching mechanisms that further mitigate the performance differences that might arise from using FCFS. These caches can absorb bursts of requests, allowing for efficient handling without the need for complex scheduling algorithms.

In summary, the lack of mechanical components in SSDs allows them to efficiently utilize a simple FCFS scheduling algorithm, which works well given their architecture and performance characteristics.

**10.11** **Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:**

**2,069, 1,212, 2,296, 2,800, 544, 1,618, 356, 1,523, 4,965, 3681**

**Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?**

1. **FCFS**
2. **SSTF**
3. **SCAN**
4. **LOOK**
5. **C-SCAN**
6. **C-LOOK**

To calculate the total distance the disk arm moves for each disk-scheduling algorithm, we need to simulate each approach using the provided request queue.

\*\*Current position:\*\* 2,150

\*\*Pending requests (in FIFO order):\*\* 2,069, 1,212, 2,296, 2,800, 544, 1,618, 356, 1,523, 4,965, 3,681

### a. FCFS (First-Come, First-Served)

- Start at 2,150, move to 2,069 (81 cylinders)

- Move to 1,212 (857 cylinders)

- Move to 2,296 (1,084 cylinders)

- Move to 2,800 (504 cylinders)

- Move to 544 (2,256 cylinders)

- Move to 1,618 (1,074 cylinders)

- Move to 356 (1,262 cylinders)

- Move to 1,523 (1,167 cylinders)

- Move to 4,965 (3,442 cylinders)

- Move to 3,681 (1,284 cylinders)

\*\*Total distance:\*\*

81 + 857 + 1,084 + 504 + 2,256 + 1,074 + 1,262 + 1,167 + 3,442 + 1,284 = \*\*13,529 cylinders\*\*

### b. SSTF (Shortest Seek Time First)

- Start at 2,150, closest request is 2,069 (81 cylinders)

- Next closest is 2,296 (227 cylinders)

- Next is 2,800 (504 cylinders)

- Next is 1,618 (1,182 cylinders)

- Next is 1,523 (95 cylinders)

- Next is 1,212 (311 cylinders)

- Next is 544 (668 cylinders)

- Next is 356 (188 cylinders)

- Next is 4,965 (4,609 cylinders)

- Next is 3,681 (3,392 cylinders)

\*\*Total distance:\*\*

81 + 227 + 504 + 1,182 + 95 + 311 + 668 + 188 + 4,609 + 3,392 = \*\*12,257 cylinders\*\*

### c. SCAN

- Start at 2,150, move to 2,296 (146 cylinders)

- Move to 2,800 (504 cylinders)

- Move to 4,965 (2,165 cylinders)

- Move to 3,681 (1,284 cylinders)

- Reverse direction back to the beginning: move to 356 (4,609 cylinders)

- Move to 544 (188 cylinders)

- Move to 1,212 (668 cylinders)

- Move to 1,618 (406 cylinders)

- Move to 2,069 (451 cylinders)

\*\*Total distance:\*\*

146 + 504 + 2,165 + 1,284 + 4,609 + 188 + 668 + 406 + 451 = \*\*11,021 cylinders\*\*

### d. LOOK

- Start at 2,150, move to 2,296 (146 cylinders)

- Move to 2,800 (504 cylinders)

- Move to 4,965 (2,165 cylinders)

- Move to 3,681 (1,284 cylinders)

- Reverse direction back to the closest request, which is 1,618 (3,487 cylinders)

- Move to 1,523 (95 cylinders)

- Move to 1,212 (311 cylinders)

- Move to 544 (668 cylinders)

- Move to 356 (188 cylinders)

\*\*Total distance:\*\*

146 + 504 + 2,165 + 1,284 + 3,487 + 95 + 311 + 668 + 188 = \*\*9,848 cylinders\*\*

### e. C-SCAN (Circular SCAN)

- Start at 2,150, move to 2,296 (146 cylinders)

- Move to 2,800 (504 cylinders)

- Move to 4,965 (2,165 cylinders)

- Move to 3,681 (1,284 cylinders)

- Jump back to the beginning and move to 356 (4,609 cylinders)

- Move to 544 (188 cylinders)

- Move to 1,212 (668 cylinders)

- Move to 1,618 (406 cylinders)

- Move to 1,523 (95 cylinders)

- Move to 2,069 (546 cylinders)

\*\*Total distance:\*\*

146 + 504 + 2,165 + 1,284 + 4,609 + 188 + 668 + 406 + 95 + 546 = \*\*11,032 cylinders\*\*

### f. C-LOOK (Circular LOOK)

- Start at 2,150, move to 2,296 (146 cylinders)

- Move to 2,800 (504 cylinders)

- Move to 4,965 (2,165 cylinders)

- Move to 3,681 (1,284 cylinders)

- Jump back to the lowest request, which is 356 (4,609 cylinders)

- Move to 544 (188 cylinders)

- Move to 1,212 (668 cylinders)

- Move to 1,618 (406 cylinders)

- Move to 1,523 (95 cylinders)

- Move to 2,069 (546 cylinders)

\*\*Total distance:\*\*

146 + 504 + 2,165 + 1,284 + 4,609 + 188 + 668 + 406 + 95 + 546 = \*\*11,032 cylinders\*\*

### Summary of Total Distances:

- FCFS: 13,529 cylinders

- SSTF: 12,257 cylinders

- SCAN: 11,021 cylinders

- LOOK: 9,848 cylinders

- C-SCAN: 11,032 cylinders

- C-LOOK: 11,032 cylinders

These calculations illustrate the total distance moved by the disk arm for each scheduling algorithm.