MODERN OPERATING SYSTEM

# ASSIGNMENT

GURUPRIYAN

20384111

## EXPERIMENT - 01

**Aim:** Write a program to simulate the following non-pre-emptive CPU scheduling algorithms to find turnaround time and waiting time.

(a) FCFS (b) SJF (c) Round Robin (pre-emptive) (d) Priority.

**Algorithms:**

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

1. Order the Processes: List the processes by their arrival time (earliest first).
2. Calculate Waiting and Turnaround Time:
    - First Process: Its waiting time is 0.
    - Next Processes: Add the burst time and waiting time of the previous process to get the waiting time for the current process.
    - Turnaround Time: Add burst time and waiting time for each process.
3. Display Details: Show each process with its ID, burst time, arrival time, waiting time, and turnaround time.

(b) SJF (Shortest Job First)

1. Order the Processes: List the processes by their burst time (shortest first).
2. Calculate Waiting and Turnaround Time: Similar to FCFS.
3. Display Details: Show each process with its ID, burst time, arrival time, waiting time, and turnaround time.

(c) Round Robin (Pre-emptive)

1. Input Processes and Quantum Time.
2. Initialize Times: Set remaining time for each process to its burst time, and waiting and turnaround times to 0.
3. Process Execution: Repeat until all processes are done:
    - Each Process:
        - If its remaining time is $\leq$ quantum and $> 0$:
            - Add remaining time to the current time.
            - Set remaining time to 0.
            - Calculate waiting time as current time minus burst time.
            - Turnaround time is the current time.
            - Increment completed count.
        - If remaining time $>$ quantum:

- Add quantum to current time.
- Subtract quantum from remaining time.

4. Display Details: Show each process with its ID, burst time, arrival time, waiting time, and turnaround time.

(d) Priority

1. Order the Processes: List the processes by their priority (lowest first).
2. Calculate Waiting and Turnaround Time: Similar to FCFS.
3. Display Details: Show each process with its ID, burst time, arrival time, waiting time, and turnaround time.

**Source Code:**

```
class Process: # Defines a class to represent a process with attributes pid, arrival_time, burst_time, and priority if provided.
    def __init__(self, pid, arrival_time, burst_time, priority=None):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.priority = priority

def fcfs(processes): # Implements First Come, First Served scheduling algorithm for given processes.
    processes.sort(key=lambda x: x.arrival_time)  # Sort processes by arrival time
    n = len(processes)
    waiting_time = [0] * n
    turnaround_time = [0] * n
    total_waiting_time = 0
    total_turnaround_time = 0
    for i in range(n):
        if i == 0:
            waiting_time[i] = 0
        else:
            waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time
        total_waiting_time += waiting_time[i]
        total_turnaround_time += turnaround_time[i]
    avg_waiting_time = total_waiting_time / n
    avg_turnaround_time = total_turnaround_time / n
    return avg_waiting_time, avg_turnaround_time

def sjf(processes): # Implements Shortest Job First scheduling algorithm for given processes.
    processes.sort(key=lambda x: x.burst_time)  # Sort processes by burst time
```

```
    return fcfs(processes)

def priority(processes): # Implements Priority scheduling algorithm for given processes.
    processes.sort(key=lambda x: x.priority, reverse=True)  # Sort processes by priority
    return fcfs(processes)

def round_robin(processes, time_slice): # Implements Round Robin scheduling algorithm for given processes with a specified time slice.
    n = len(processes)
    burst_remaining = [p.burst_time for p in processes]
    waiting_time = 0
    turnaround_time = 0
    t = 0
    while True:
        done = True
        for i in range(n):
            if burst_remaining[i] > 0:
                done = False
                if burst_remaining[i] > time_slice:
                    t += time_slice
                    burst_remaining[i] -= time_slice
                else:
                    t += burst_remaining[i]
                    waiting_time += t - processes[i].burst_time
                    turnaround_time += t
                    burst_remaining[i] = 0
        if done:
            break
    avg_waiting_time = waiting_time / n
    avg_turnaround_time = turnaround_time / n
    return avg_waiting_time, avg_turnaround_time


if __name__ == "__main__":
    processes = [
        Process(3, 5, 4, 1),
        Process(4, 1, 8, 4),
        Process(3, 2, 3, 1),
        Process(2, 3, 6, 2)
    ]
    print("FCFS:")
    avg_waiting_time, avg_turnaround_time = fcfs(processes)
    print("Average Waiting Time:", avg_waiting_time)
    print("Average Turnaround Time:", avg_turnaround_time)
    print("\nSJF:")
    avg_waiting_time, avg_turnaround_time = sjf(processes)
    print("Average Waiting Time:", avg_waiting_time)
```

```
    print("Average Turnaround Time:", avg_turnaround_time)
    print("\nPriority:")
    avg_waiting_time, avg_turnaround_time = priority(processes)
    print("Average Waiting Time:", avg_waiting_time)
    print("Average Turnaround Time:", avg_turnaround_time)
    print("\nRound Robin:")
    avg_waiting_time, avg_turnaround_time = round_robin(processes, 2)
    print("Average Waiting Time:", avg_waiting_time)
    print("Average Turnaround Time:", avg_turnaround_time)
```

Output:

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS> & "C:/Users/
g_time.py"
FCFS:
Average Waiting Time: 9.0
Average Turnaround Time: 14.25

SJF:
Average Waiting Time: 9.0
Average Turnaround Time: 14.25

Priority:
Average Waiting Time: 9.0
Average Turnaround Time: 14.25

Round Robin:
Average Waiting Time: 11.25
Average Turnaround Time: 16.5
```

## EXPERIMENT - 02

**Aim:** Write a program to simulate the following file allocation strategies.

(a) Sequential (b) Indexed (c) Linked

**Algorithms:**

1. Sequential File Allocation

- Search for Free Blocks: Start from the beginning of the file allocation table and look for consecutive free blocks.
- Allocate Blocks: If enough free blocks are found, mark them as allocated for the file.
- Update Tables: Store the file name and the corresponding blocks in the file allocation table and file list.

2. Indexed File Allocation

- Create Index Block: Allocate an index block for each file, which will hold the block numbers of the data blocks.
- Enter Block Numbers: For each data block, the user enters a block number.
- Allocate Blocks: Mark the entered block as allocated and store its number in the index block.
- Update Tables: Store the file name and the index block in the file allocation table and file list.

3. Linked File Allocation

- Enter Block Numbers: For each data block, the user enters a block number.
- Allocate Blocks: Mark the entered block as allocated and store the block numbers sequentially, creating a linked list of blocks.
- Update List: Store the file name and the list of block numbers in the file list. The file allocation table is not used in this strategy.

4. Displaying File Allocation Table

- Print Table: Show the file allocation table with block numbers and the corresponding file names.

5. Displaying Files

- Print Files: Show the list of files, including each file name and the blocks allocated to it.

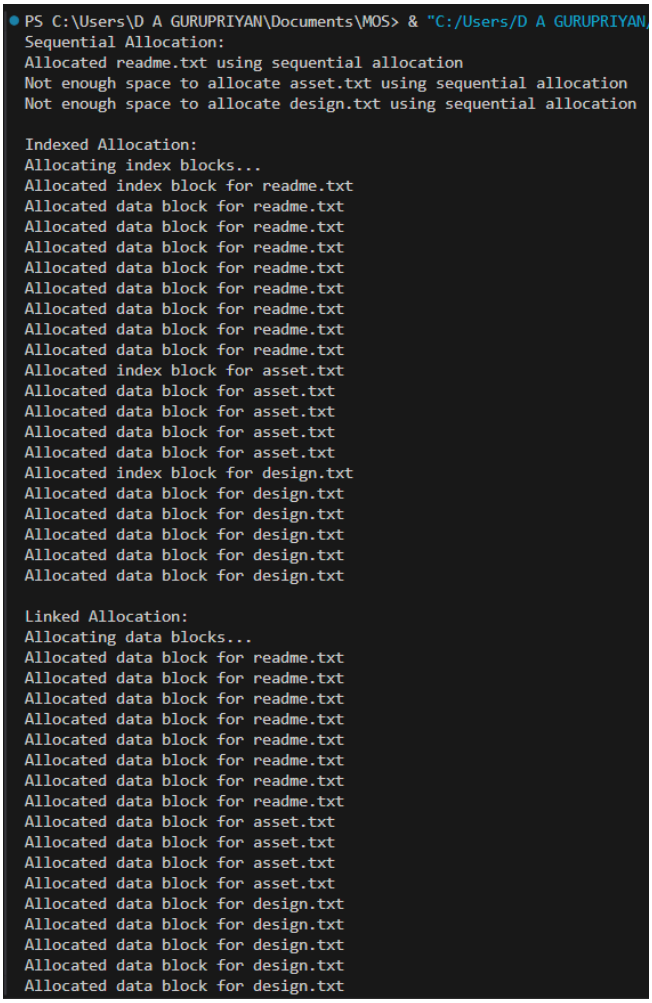
**Source Code:**

```
class File: # Defines a class to represent a file with attributes name and
size.
```

```
    def __init__(self, name, size):
        self.name = name
        self.size = size

class Block: # Defines a class to represent a block with data and a reference
to the next block.
    def __init__(self, data):
        self.data = data
        self.next_block = None

def sequential_allocation(files, disk_blocks):  # Allocates disk space to
files using sequential allocation.
    allocated_space = 0
    for file in files:
        if allocated_space + file.size <= disk_blocks:
            allocated_space += file.size
            print(f"Allocated {file.name} using sequential allocation")
        else:
            print(f"Not enough space to allocate {file.name} using sequential
allocation")

def indexed_allocation(files, disk_blocks): # Allocates disk space to files
using indexed allocation.
    index_blocks = len(files)
    if index_blocks <= disk_blocks:
        print("Allocating index blocks...")
        for file in files:
            print(f"Allocated index block for {file.name}")
            allocated_blocks = 0
            while allocated_blocks < file.size:
                print(f"Allocated data block for {file.name}")
                allocated_blocks += 1
    else:
        print("Not enough space to allocate index blocks for each file")

def linked_allocation(files, disk_blocks): # Allocates disk space to files
using linked allocation.
    print("Allocating data blocks...")
    for file in files:
        if file.size <= disk_blocks:
            allocated_blocks = 0
            current_block = Block(file.name)
            while allocated_blocks < file.size:
                print(f"Allocated data block for {file.name}")
                allocated_blocks += 1
                if allocated_blocks < file.size:
                    current_block.next_block = Block(file.name)
                    current_block = current_block.next_block
```

```
        else:
            print(f"Not enough space to allocate {file.name} using linked
allocation")

if __name__ == "__main__":
    disk_blocks = 10
    files = [
        File("readme.txt", 8),
        File("asset.txt", 4),
        File("design.txt", 5)
    ]
    print("Sequential Allocation:")
    sequential_allocation(files, disk_blocks)
    print("\nIndexed Allocation:")
    indexed_allocation(files, disk_blocks)
    print("\nLinked Allocation:")
    linked_allocation(files, disk_blocks)
```

**Output:**



## EXPERIMENT - 03

**Aim:** Write a program to simulate paging technique of memory management

**Algorithm:**

- Start: Begin the process.
- Input Memory and Page Size: Ask the user to enter the memory size and page size in bytes.
- Calculate Number of Pages: Determine how many pages are needed by dividing the memory size by the page size and rounding up to the nearest whole number.
- Input Logical Address: Ask the user to enter a logical address.
- Calculate Page Offset: Find the page offset by taking the remainder of the logical address divided by the page size.
- Calculate Page Number: Determine the page number by dividing the logical address by the page size.
- Calculate Physical Address: Compute the physical address using the page number and page offset.
- Output Results: Display the number of pages, the page offset, and the physical address.
- Stop: End the process.

**Source Code:**

```
class PageTable: # Manages the page table for address translation.
    def __init__(self, num_frames): # Initializes the page table with a
specified number of frames.
        self.page_table = {}
        self.num_frames = num_frames

    def map_page(self, logical_address, frame_number): # Maps a logical
address to a frame number in the page table.
        self.page_table[logical_address] = frame_number
    def translate_address(self, logical_address): # Translates a logical
address to a physical address using the page table.
        if logical_address in self.page_table:
            return self.page_table[logical_address]
        else:
            return None

class Memory: # Represents the main memory.
    def __init__(self, num_frames): # Initializes the memory with a specified
number of frames.
        self.frames = [None] * num_frames
    def load_page(self, frame_number, page_data): # Loads a page into the
specified frame of memory.
        self.frames[frame_number] = page_data
```

```
    def get_page(self, frame_number): # Retrieves the data stored in a
specific frame of memory.
        return self.frames[frame_number]

class Process:  # Represents a process with its associated pages.
    def __init__(self, process_id, pages): # Initializes the process with a
unique ID and its pages.
        self.process_id = process_id
        self.pages = pages

def simulate_paging(processes, memory_size): # Simulates paging by loading
processes into memory and performing address translation.
    num_frames = memory_size // 4  # Assuming each page is 4 units in size
    memory = Memory(num_frames)
    page_table = PageTable(num_frames)
    for process in processes:
        print(f"Loading process {process.process_id} into memory...")
        for i, page in enumerate(process.pages):
            frame_number = i % num_frames
            memory.load_page(frame_number, page)
            page_table.map_page((process.process_id, i), frame_number)
            print(f"Page {i} of process {process.process_id} loaded into frame
{frame_number}")
    print("\nAddress Translation:")
    for process in processes:
        print(f"Process {process.process_id} pages:")
        for i, _ in enumerate(process.pages):
            logical_address = (process.process_id, i)
            physical_address = page_table.translate_address(logical_address)
            print(f"Logical Address: {logical_address} -> Physical Address:
{physical_address}")

if __name__ == "__main__":
    processes = [
        Process(1, ['A', 'E', 'I', 'O', 'U']),
        Process(2, ['G', 'U', 'R', 'U', 'P']),
    ]
    memory_size = 16  # 16 units of memory
    simulate_paging(processes, memory_size)
```

**Output:**

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS> & "C:/Users/D A GU
py"
Loading process 1 into memory...
Page 0 of process 1 loaded into frame 0
Page 1 of process 1 loaded into frame 1
Page 2 of process 1 loaded into frame 2
Page 3 of process 1 loaded into frame 3
Page 4 of process 1 loaded into frame 0
Loading process 2 into memory...
Page 0 of process 2 loaded into frame 0
Page 1 of process 2 loaded into frame 1
Page 2 of process 2 loaded into frame 2
Page 3 of process 2 loaded into frame 3
Page 4 of process 2 loaded into frame 0

Address Translation:
Process 1 pages:
Logical Address: (1, 0) -> Physical Address: 0
Logical Address: (1, 1) -> Physical Address: 1
Logical Address: (1, 2) -> Physical Address: 2
Logical Address: (1, 3) -> Physical Address: 3
Logical Address: (1, 4) -> Physical Address: 0
Process 2 pages:
Logical Address: (2, 0) -> Physical Address: 0
Logical Address: (2, 1) -> Physical Address: 1
Logical Address: (2, 2) -> Physical Address: 2
Logical Address: (2, 3) -> Physical Address: 3
Logical Address: (2, 4) -> Physical Address: 0
```

## EXPERIMENT - 04

**Aim:** Write a program to simulate the following file organization techniques

(a) Single level directory (b) Two level directory (c) Hierarchical.

**Algorithm:**

(a) Single Level Directory

1. Initialize: Start with an empty list to store file names and their sizes.
2. Operations:
    - create_file(file_name, size):
        - Check if the file name is already in the list. If it is, show an error message.
        - If not, add the file name and size to the list.
    - delete_file(file_name):
        - Check if the file name is in the list. If it is, remove it from the list.
        - If not, show an error message.
    - display_files():
        - Show all file names and their sizes from the list.

(b) Two Level Directory

1. Initialize: Start with an empty dictionary for the directory structure.
2. Operations:
    - create_file(directory_name, file_name, size):
        - Check if the directory exists. If not, create it as a key with an empty dictionary as its value.
        - Check if the file name is already in the directory. If it is, show an error message.
        - If not, add the file name and size to the directory.
    - delete_file(directory_name, file_name):
        - Check if the directory exists. If it does, check if the file name is in the directory. If it is, remove it.
        - If not, show an error message.
    - display_files():
        - Show the directory structure and files in it.

(c) Hierarchical Directory

1. Initialize: Start with a root directory as an empty dictionary.
2. Operations:
    - create_file(path, file_name, size):
        - Split the path into directory names.
        - Traverse the directory structure from the root using the directory names.
        - Create any directories that don't exist.
        - Check if the file name is already in the directory. If it is, show an error message.
        - If not, add the file name and size to the directory.
    - delete_file(path):
        - Split the path into directory names.
        - Traverse the directory structure from the root using the directory names.
        - Check if the file name is in the directory. If it is, remove it.
        - If not, show an error message.
    - display_files():
        - Show the directory structure and files in it.

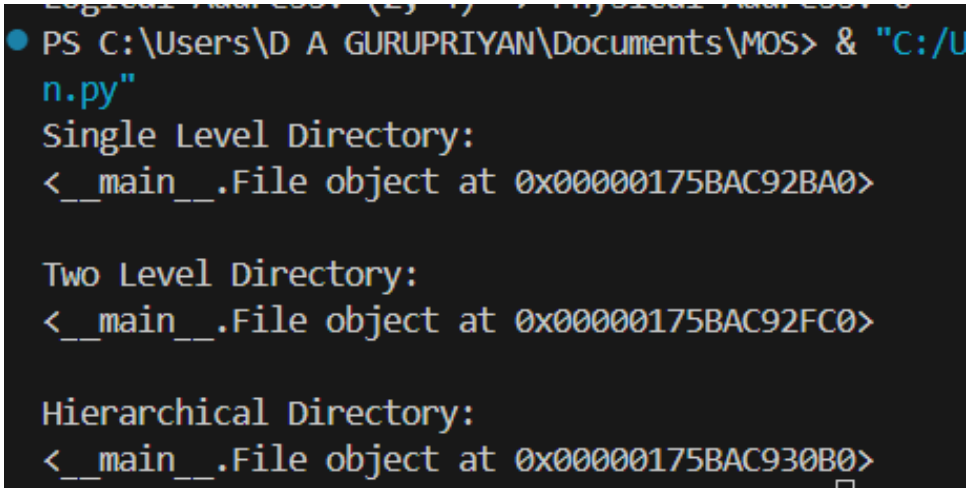
**Source Code:**

```
class File: # Represents a file with attributes name and owner.
    def __init__(self, name, owner):
        self.name = name
        self.owner = owner

class SingleLevelDirectory: # Represents a single-level directory structure.
    def __init__(self): # Initializes an empty single-level directory.
        self.files = {}
    def add_file(self, file): # Adds a file to the single-level directory.
        self.files[file.name] = file
    def search_file(self, file_name): # Searches for a file in the single-
level directory.
        if file_name in self.files:
            return self.files[file_name]
        else:
            return None

class TwoLevelDirectory: # Represents a two-level directory structure.
```

```
    def __init__(self):
        self.master_directory = {}
    def add_user_directory(self, user, directory):
        self.master_directory[user] = directory
    def search_file(self, user, file_name):
        if user in self.master_directory:
            user_directory = self.master_directory[user]
            return user_directory.search_file(file_name)
        else:
            return None

class HierarchicalDirectory: # Represents a hierarchical directory structure.
    def __init__(self, name):
        self.name = name
        self.files = {}
        self.subdirectories = {}
    def add_file(self, file):
        self.files[file.name] = file
    def add_subdirectory(self, directory):
        self.subdirectories[directory.name] = directory
    def search_file(self, file_name):
        if file_name in self.files:
            return self.files[file_name]
        else:
            for subdirectory in self.subdirectories.values():
                result = subdirectory.search_file(file_name)
                if result:
                    return result
            return None

if __name__ == "__main__":
    print("Single Level Directory:")
    single_level_dir = SingleLevelDirectory()
    single_level_dir.add_file(File("single_level1.txt", "user1"))
    single_level_dir.add_file(File("single_level2.txt", "user2"))
    print(single_level_dir.search_file("single_level1.txt"))
    print("\nTwo Level Directory:")
    two_level_dir = TwoLevelDirectory()
    user1_dir = SingleLevelDirectory()
    user1_dir.add_file(File("two_level1.txt", "user1"))
    user2_dir = SingleLevelDirectory()
    user2_dir.add_file(File("two_level2.txt", "user2"))
    two_level_dir.add_user_directory("user1", user1_dir)
    two_level_dir.add_user_directory("user2", user2_dir)
    print(two_level_dir.search_file("user1", "two_level1.txt"))
    print("\nHierarchical Directory:")
    root_dir = HierarchicalDirectory("root")
    user1_dir = HierarchicalDirectory("user1")
```

```
    user1_dir.add_file(File("file1.txt", "user1"))
    user2_dir = HierarchicalDirectory("user2")
    user2_dir.add_file(File("file2.txt", "user2"))
    root_dir.add_subdirectory(user1_dir)
    root_dir.add_subdirectory(user2_dir)
    print(root_dir.search_file("file1.txt"))
```

**Output:**



## EXPERIMENT - 05

**Aim:** Write a program to simulate Banker's algorithm for the purpose of deadlock avoidance.

**Procedure:**

- **Initialize Data Structures**:
    - **available_resources**: A list showing the number of available resources for each type.
    - **max_resources**: A table showing the maximum resources each process might need.
    - **allocated_resources**: A table showing the resources currently assigned to each process.
    - **need**: A table showing the remaining resources each process still needs.
    - **num_processes**: The number of processes.
    - **num_resources**: The number of resource types.
- **Calculate the Need Matrix**:
    - Subtract the **allocated_resources** from **max_resources** to get the **need** matrix.
- **Implement the is_safe_state Function**:
    - Copy **available_resources** to **work**.
    - Create a list called **finish** with all values set to False, indicating processes are not finished.
    - Create an empty list called **safe_sequence**.
    - While there are processes that can still get their required resources:
        - Check each process to see if its resource needs can be met by **work**.
        - If yes, update **work** with resources released by the process, mark it as finished, and add it to **safe_sequence**.
    - If all processes finish and **safe_sequence** includes all processes, return True and the sequence.
    - Otherwise, return False and an empty sequence.
- **Implement the request_resources Function**:
    - Check if the requested resources do not exceed the process's needs.
    - Check if the requested resources are available.

- If both checks pass, temporarily allocate the resources and update the state.
- Use **is_safe_state** to check if the system remains safe.
- If safe, print success and the sequence.
- If not safe, print a message indicating it leads to an unsafe state and undo the changes.

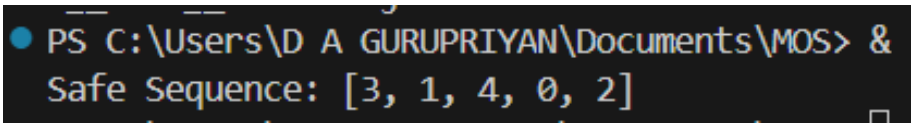
- **Implement the display_state Function**:
  - Print the current state of the system, showing the resources available, allocated, and needed for each process.

**Source Code:**

```
import numpy as np

class BankersAlgorithm: # Implements the Banker's Algorithm for deadlock
avoidance.
    def __init__(self, allocation, max_demand, available):
        self.allocation = np.array(allocation)
        self.max_demand = np.array(max_demand)
        self.available = np.array(available)
        self.num_processes, self.num_resources = self.allocation.shape
    def is_safe_state(self): # Checks if the system is in a safe state.
        work = self.available.copy()
        finish = np.zeros(self.num_processes, dtype=bool)
        # Check if a process can be satisfied with the current resources
        def can_satisfy(process):  # Checks if a process can be satisfied with
the current resources.
            return all(need <= work) and not finish[process]
        safe_sequence = []
        while np.any(~finish):
            found = False
            for process in range(self.num_processes):
                need = self.max_demand[process] - self.allocation[process]
                if can_satisfy(process):
                    work += self.allocation[process]
                    finish[process] = True
                    safe_sequence.append(process)
                    found = True
                    break
            if not found:
                break
        return len(safe_sequence) == self.num_processes, safe_sequence

if __name__ == "__main__":
    allocation = [
        [1, 0, 0],
```

```
        [0, 2, 0],
        [2, 0, 3],
        [2, 3, 1],
        [2, 0, 2]
    ]
    max_demand = [
        [7, 5, 3],
        [3, 2, 2],
        [9, 0, 2],
        [2, 2, 2],
        [4, 3, 3]
    ]
    available = [2, 3, 2]
    banker = BankersAlgorithm(allocation, max_demand, available)
    safe, sequence = banker.is_safe_state()
    if safe:
        print("Safe Sequence:", sequence)
    else:
        print("Unsafe State, deadlock may occur")
```

**Output:**



## EXPERIMENT - 06

**Aim:** Write a program to simulate disk scheduling algorithms

(a) FCFS (b) SCAN (c) C-SCAN.

**Algorithm:**

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

- Start at Initial Position: Begin from the initial head position.
- Process Requests in Order: Handle disk requests in the order they arrive.
- Calculate Movement: For each request, calculate the distance (absolute difference) between the current head position and the request.
- Update Head Position: Move the head to the position of the processed request.
- Repeat: Continue this for all requests.
- Total Movement: The total head movement is the sum of all calculated distances.

2. SCAN (Elevator) Algorithm

- Start at Initial Position: Begin from the initial head position.
- Move in One Direction: Move the head in one direction (towards higher or lower cylinder numbers).
- Process Requests: Handle requests as you encounter them, removing each from the list.
- Change Direction at End: When reaching the end of the disk, reverse the direction of the head.
- Repeat: Continue this process until all requests are handled.
- Total Movement: The total head movement is the total distance the head travels.

3. C-SCAN (Circular SCAN) Algorithm

- Start at Initial Position: Begin from the initial head position.
- Move in One Direction: Move the head in one direction to the end of the disk.
- Process Requests: Handle requests as you encounter them, removing each from the list.
- Jump to Beginning: After reaching the end, move the head back to the start of the disk.
- Continue in Same Direction: Move the head in the same direction again, processing any remaining requests.
- Repeat: Continue this process until all requests are handled.

- Total Movement: The total head movement is the total distance the head travels.

**Source code:**

```
class DiskScheduler: # Represents a disk scheduler for disk I/O operations.
    def __init__(self, requests, head_start):
        self.requests = requests
        self.head_position = head_start

    def fcfs(self): # Implements the First-Come, First-Served (FCFS) disk
scheduling algorithm.
        total_seek_time = 0
        for request in self.requests:
            total_seek_time += abs(request - self.head_position)
            self.head_position = request
        return total_seek_time

    def scan(self): # Implements the SCAN disk scheduling algorithm.
        total_seek_time = 0
        sorted_requests = sorted(self.requests)
        min_request = sorted_requests[0]
        max_request = sorted_requests[-1]
        # Move the head towards the maximum request
        for request in sorted_requests:
            if request >= self.head_position:
                total_seek_time += abs(request - self.head_position)
                self.head_position = request
        # Move the head towards the minimum request
        total_seek_time += abs(max_request - min_request)
        self.head_position = min_request
        return total_seek_time

    def c_scan(self): # Implements the Circular SCAN (C-SCAN) disk scheduling
algorithm.
        total_seek_time = 0
        sorted_requests = sorted(self.requests)
        max_request = sorted_requests[-1]
        # Move the head towards the maximum request
        for request in sorted_requests:
            if request >= self.head_position:
                total_seek_time += abs(request - self.head_position)
                self.head_position = request
        # Move the head to the beginning of the disk
        total_seek_time += abs(max_request)
        self.head_position = 0
        return total_seek_time

if __name__ == "__main__":
    requests = [120, 18, 37, 144, 14, 97, 65, 67]
```

```
    head_start = 53
    scheduler = DiskScheduler(requests, head_start)
    print("FCFS Seek Time:", scheduler.fcfs())
    print("SCAN Seek Time:", scheduler.scan())
    print("C-SCAN Seek Time:", scheduler.c_scan())
```

**Output:**

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS>
y"
FCFS Seek Time: 542
SCAN Seek Time: 207
C-SCAN Seek Time: 274
```

## EXPERIMENT - 07

**Aim:** Write a program to simulate page replacement algorithms

(a) FIFO (b) LRU (c) LFU

**Procedure:**

(a) FIFO (First-In, First-Out)

1. Initialize: Start with an empty page table.
2. Check Page Reference: For each page request, see if the page is in the page table.
3. Page Fault Handling:
    - If the page isn't in the table (page fault):
        - If there's space, add the page to the table.
        - If the table is full, remove the oldest page and add the new one.
4. Count Faults: Keep track of how many page faults occur.

(b) LRU (Least Recently Used)

1. Initialize: Start with an empty page table.
2. Check Page Reference: For each page request, see if the page is in the page table.
3. Page Fault Handling:
    - If the page isn't in the table (page fault):
        - If there's space, add the page to the table.
        - If the table is full, replace the least recently used page with the new one.
4. Update Usage: If the page is already in the table, mark it as the most recently used.
5. Count Faults: Keep track of how many page faults occur.

(c) LFU (Least Frequently Used)

1. Initialize: Start with an empty page table and a frequency table.
2. Check Page Reference: For each page request, see if the page is in the page table.
3. Page Fault Handling:
    - If the page isn't in the table (page fault):
        - If there's space, add the page to the table.
        - If the table is full, replace the least frequently used page with the new one.

4. Update Frequency: If the page is already in the table, increase its frequency count.
5. Count Faults: Keep track of how many page faults occur.

**Source Code:**

```
from collections import deque, Counter

class PageReplacement: # Represents a page replacement algorithm for managing page faults.
    def __init__(self, page_frames):
        self.page_frames = page_frames
        self.page_queue = deque()
        self.page_access = []

    def fifo(self, page_references): # Implements the First-In, First-Out (FIFO) page replacement algorithm.
        page_faults = 0
        for page in page_references:
            if page not in self.page_frames:
                page_faults += 1
                if len(self.page_frames) == len(self.page_queue):
                    if self.page_queue:
                        self.page_frames.remove(self.page_queue.popleft())
                self.page_frames.add(page)
                self.page_queue.append(page)
            elif page in self.page_frames:
                if page in self.page_queue:
                    self.page_queue.remove(page)
                self.page_queue.append(page)
        return page_faults

    def lru(self, page_references): # Implements the Least Recently Used (LRU) page replacement algorithm.
        page_faults = 0
        for page in page_references:
            if page not in self.page_frames:
                page_faults += 1
                if len(self.page_frames) == len(self.page_access):
                    self.page_frames.remove(self.page_access.pop(0))
                self.page_frames.add(page)
                self.page_access.append(page)
            else:
                if page in self.page_access:
                    self.page_access.remove(page)
                self.page_access.append(page)
        return page_faults
```

```
    def lfu(self, page_references): # Implements the Least Frequently Used
(LFU) page replacement algorithm.
        page_faults = 0
        page_counter = Counter()
        for page in page_references:
            if page not in self.page_frames:
                page_faults += 1
                if len(self.page_frames) == len(self.page_access):
                    least_frequent_page = min(self.page_access,
key=page_counter.get)
                    self.page_frames.remove(least_frequent_page)
                    self.page_access.remove(least_frequent_page)
                self.page_frames.add(page)
                self.page_access.append(page)
            page_counter[page] += 1
        return page_faults

if __name__ == "__main__":
    page_references = [2, 2, 3, 4, 5, 1, 2, 4, 9, 7, 8, 7, 8, 1, 7, 8, 9, 4,
4, 5, 4, 2, 3, 2, 2, 2, 3, 6]
    page_frames = set()
    pr = PageReplacement(page_frames)
    print("FIFO Page Faults:", pr.fifo(page_references.copy()))
    pr = PageReplacement(page_frames)
    print("LRU Page Faults:", pr.lru(page_references.copy()))
    pr = PageReplacement(page_frames)
    print("LFU Page Faults:", pr.lfu(page_references.copy()))
```

**Output:**

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS>
algo.py"
FIFO Page Faults: 24
LRU Page Faults: 8
LFU Page Faults: 0
```

## EXPERIMENT - 08

**Aim:** Write a program to simulate producer-consumer problem using semaphores

**Procedure:**

- **Initialization**:
    - Define buffer size and initialize buffer, mutex semaphore, empty semaphore (equal to buffer size), and full semaphore (set to 0).
    - Define the number of items to produce and initialize counters for items produced and consumed.
- **Producer Class**:
    - Create a class called Producer extending threading.Thread.
    - Implement the run() method for the Producer:
        - Loop until desired items are produced.
        - Simulate production time with sleep function.
        - Acquire empty semaphore to wait if buffer is full.
        - Acquire mutex semaphore to access buffer.
        - Append produced item to buffer.
        - Print producer's thread name and the produced item.
        - Update items produced counter.
        - Release mutex semaphore.
        - Release full semaphore to notify consumer an item is available.
- **Consumer Class**:
    - Create a class called Consumer extending threading.Thread.
    - Implement the run() method for the Consumer:
        - Loop until desired items are consumed.
        - Simulate consumption time with sleep function.
        - Acquire full semaphore to wait if buffer is empty.
        - Acquire mutex semaphore to access buffer.
        - Remove consumed item from buffer.
        - Print consumer's thread name and the consumed item.
        - Update items consumed counter.
        - Release mutex semaphore.

- Release empty semaphore to notify producer that space is available.

- **Create Instances**:
  - Create instances of Producer and Consumer classes.
- **Start Threads**:
  - Start producer and consumer threads.
- **Wait for Completion**:
  - Wait for all threads to finish using join() method on each thread.

**Source Code:**

```
import time
import random

BUFFER_SIZE = 5

buffer = []
buffer_lock = False
buffer_full = 0

def producer_consumer(): # Defines the producer-consumer function.
    global buffer, buffer_lock, buffer_full
    while True:
        item = random.randint(1, 100)
        if buffer_full == BUFFER_SIZE:
            print("Buffer is full. Waiting...")
            time.sleep(0.1)  # Wait if buffer is full
        else:
            if buffer_lock:
                print("Buffer is being modified. Waiting...")
                time.sleep(0.1)  # Wait if buffer is being modified
            else:
                buffer_lock = True  # Acquire the lock
                buffer.append(item)  # Add item to the buffer
                buffer_full += 1
                print(f"Produced {item}, Buffer: {buffer}")
                buffer_lock = False  # Release the lock
                if random.random() < 0.5:
                    if buffer_full == 0:
                        print("Buffer is empty. Waiting...")
                        time.sleep(0.1)  # Wait if buffer is empty
                    else:
                        if buffer_lock:
                            print("Buffer is being modified. Waiting...")
```

```
                            time.sleep(0.1)  # Wait if buffer is being
modified
                        else:
                            buffer_lock = True  # Acquire the lock
                            item = buffer.pop(0)  # Remove item from the
buffer
                            buffer_full -= 1
                            print(f"Consumed {item}, Buffer: {buffer}")
                            buffer_lock = False  # Release the lock
        time.sleep(random.uniform(0.1, 1))
# Start the producer-consumer loop
producer_consumer()
```

**Output:**

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS> & "C:/Users/D A G
.py"
Produced 78, Buffer: [78]
Produced 58, Buffer: [78, 58]
Consumed 78, Buffer: [58]
Produced 59, Buffer: [58, 59]
Consumed 58, Buffer: [59]
Produced 100, Buffer: [59, 100]
Produced 46, Buffer: [59, 100, 46]
Consumed 59, Buffer: [100, 46]
Produced 98, Buffer: [100, 46, 98]
Consumed 100, Buffer: [46, 98]
Produced 32, Buffer: [46, 98, 32]
Produced 57, Buffer: [46, 98, 32, 57]
Consumed 46, Buffer: [98, 32, 57]
Produced 75, Buffer: [98, 32, 57, 75]
Consumed 98, Buffer: [32, 57, 75]
Produced 73, Buffer: [32, 57, 75, 73]
Produced 36, Buffer: [32, 57, 75, 73, 36]
Buffer is full. Waiting...
```

## EXPERIMENT - 09

**Aim:** Write a program to simulate the concept of Dining-Philosophers problem

**Procedure:**

1. Setup: Create a group of n philosophers and n chopsticks.
2. Initialization:
    - Each philosopher gets a unique identifier and references to their left and right chopsticks.
3. Start Threads:
    - Begin a thread for each philosopher.
4. Philosopher's Lifecycle:
    - Each philosopher enters an infinite loop representing their lifecycle:
        1. Think: Output that the philosopher is thinking.
        2. Wait for Chopsticks: Wait until both chopsticks are available.
        3. Pick Up Chopsticks: Acquire locks for both the left and right chopsticks.
        4. Eat: Output that the philosopher is eating.
        5. Put Down Chopsticks: Release locks for both the left and right chopsticks.
5. Repeat Indefinitely:
    - Repeat this loop indefinitely.

**Source code:**

```
import time
import random

NUM_PHILOSOPHERS = 5
MAX_EATING_TIME = 3

forks = [True] * NUM_PHILOSOPHERS  # True indicates the fork is available

def philosopher(index): # Defines the philosopher function.
    for _ in range(MAX_EATING_TIME):
        think(index)
        eat(index)

def think(index): # Defines the think function.
    print(f"Philosopher {index} is thinking...")
    time.sleep(random.uniform(0.1, 1))
```

```
def eat(index): # Defines the eat function.
    print(f"Philosopher {index} is hungry and trying to pick up forks...")
    left_fork = index
    right_fork = (index + 1) % NUM_PHILOSOPHERS
    while True:
        if forks[left_fork] and forks[right_fork]:
            forks[left_fork] = False
            forks[right_fork] = False
            print(f"Philosopher {index} is eating...")
            time.sleep(random.uniform(0.1, 1))
            forks[left_fork] = True
            forks[right_fork] = True
            break
        else:
            print(f"Philosopher {index} couldn't acquire forks. Retrying...")
            time.sleep(0.1)

if __name__ == "__main__":
    for i in range(NUM_PHILOSOPHERS):
        philosopher(i)
    print("Dinner is over. Philosophers are full and content.")
```

**Output:**

```
PS C:\Users\D A GURUPRIYAN\Documents\MOS> & "C:/Users/D A GURUPRIYAN/App
Philosopher 0 is thinking...
Philosopher 0 is hungry and trying to pick up forks...
Philosopher 0 is eating...
Philosopher 0 is thinking...
Philosopher 0 is hungry and trying to pick up forks...
Philosopher 0 is eating...
Philosopher 0 is thinking...
Philosopher 0 is hungry and trying to pick up forks...
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 1 is hungry and trying to pick up forks...
Philosopher 1 is eating...
Philosopher 1 is thinking...
Philosopher 1 is hungry and trying to pick up forks...
Philosopher 1 is eating...
Philosopher 1 is thinking...
Philosopher 1 is hungry and trying to pick up forks...
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 2 is hungry and trying to pick up forks...
Philosopher 2 is eating...
Philosopher 2 is thinking...
Philosopher 2 is hungry and trying to pick up forks...
Philosopher 2 is eating...
Philosopher 2 is thinking...
Philosopher 2 is hungry and trying to pick up forks...
Philosopher 2 is eating...
Philosopher 3 is thinking...
Philosopher 3 is hungry and trying to pick up forks...
Philosopher 3 is eating...
Philosopher 3 is thinking...
Philosopher 3 is hungry and trying to pick up forks...
Philosopher 3 is eating...
Philosopher 3 is thinking...
Philosopher 3 is hungry and trying to pick up forks...
Philosopher 3 is eating...
Philosopher 4 is thinking...
Philosopher 4 is hungry and trying to pick up forks...
Philosopher 4 is eating...
Philosopher 4 is thinking...
Philosopher 4 is hungry and trying to pick up forks...
Philosopher 4 is eating...
Philosopher 4 is thinking...
Philosopher 4 is hungry and trying to pick up forks...
Philosopher 4 is eating...
Dinner is over. Philosophers are full and content.
```