INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

OPERATING SYSTEMS LABORATORY								
III Semester: CSE / IT / CSE (AI&ML) / CSE (DS) / CSE (CS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACSD10	**Core**	0	0	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisites: **There are no prerequisites to take this course.**								

I. COURSE OVERVIEW:
The course covers some of the design aspects of operating system concepts. Topics covered include process scheduling, memory management, deadlocks, disk scheduling strategies, and file allocation methods. The main objective of the course is to teach the students how to select and design algorithms that are appropriate for problems that they might encounter in real life.

II. COURSE OBJECTIVES
The students will try to learn:

I. The functionalities of main components in operating systems and analyze the algorithms used in process management.
II. Algorithms used in memory management and I/O management
III. Different methods for preventing or avoiding deadlocks and File systems.

III. COURSE OUTCOMES:
At the end of the course students should be able to:

CO1 Acquire knowledge of the operating system structure and process
CO2 Analyze the performance of process scheduling algorithms
CO3 Evaluate the process memory requirement and its fragmentation.
CO4 Analyze the safe state and deadlock mechanism
CO5 Analyze the performance of the disk scheduling algorithms
CO6 Ability to simulate file structures and allocation methods

OPERATING SYSTEMS LABORATORY (ACSD10)

CONTENTS

IV. SYLLABUS:

EXERCISES FOR OPERATING SYSTEMS LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 The Busy Printer

Imagine a IARE computer lab with a single printer that serves multiple students. The lab operates on a first-come, first-served basis for print jobs.

One morning, the lab opens, and the printer is ready to accept jobs. The following print jobs arrive in sequence:

Job A: Submitted by A, consisting of 10 pages.

Job B: Submitted by B, consisting of 5 pages.

Job C: Submitted by C, consisting of 3 pages.

Job D: Submitted by D, consisting of 7 pages.

Each job arrives at the lab at different times, and they start printing immediately upon arrival. The printer processes each job in the order it arrives, and each job completes printing without interruption.

Your task is to simulate the FCFS scheduling algorithm for these print jobs and calculate the total time taken to complete all printing tasks, assuming that the printer prints at a rate of 1 page per second.

Input: The arrival time of each job (in seconds). The number of pages to be printed for each job.

Job A: Arrival time = 0 seconds, Pages = 10

Job B: Arrival time = 1 second, Pages = 5

Job C: Arrival time = 3 seconds, Pages = 3

Job D: Arrival time = 5 seconds, Pages = 7

Output: The total time (in seconds) taken to complete all printing tasks.

27 seconds

Assumptions:

The printer starts immediately on each job's arrival.

No additional delay between jobs.

Explanation:

Job A arrives first at time 0 and takes 10 seconds to print (10 pages).

Job B arrives at time 1 and starts printing immediately after Job A completes. It takes 5 seconds to print (5 pages).

Job C arrives at time 3 and starts immediately after Job B completes. It takes 3 seconds to print (3 pages).

Job D arrives at time 5 and starts immediately after Job C completes. It takes 7 seconds to print (7 pages).

Thus, the total time taken to complete all print jobs is 27 seconds.

```
def calculate_total_time(arrival_times, pages):
    current_time = 0
    total_time = 0

    for i in range(len(arrival_times)):
        # Write code here

# Driver Code
arrival_times = [0, 1, 3, 5]
pages = [10, 5, 3, 7]

total_time = calculate_total_time(arrival_times, pages)

print(f"The total time taken to complete all printing tasks is: {total_time} seconds")
```

1.2 The Software Developer's Tasks

Sophia, a software developer, has several programming tasks to complete in her project. Each task requires a different amount of time to finish. She decides to use the SJF scheduling algorithm to manage her tasks efficiently.
Here are the tasks she needs to complete:

- **Task A:** Writing a simple utility function, which takes 3 hours.
- **Task B:** Debugging a critical issue in the main application, which takes 5 hours.
- **Task C:** Refactoring a complex module, which takes 7 hours.
- **Task D:** Testing and documenting a new feature, which takes 4 hours.

Sophia decides to implement the SJF algorithm for scheduling these tasks based on their execution times.

Your task is to simulate the SJF scheduling algorithm for Sophia's tasks and calculate the total time required to complete all tasks, assuming that each task starts immediately after the previous one is completed.

Input: Execution times of each task.
Task A: 3 hours
Task B: 5 hours
Task C: 7 hours
Task D: 4 hours

Output: The total time (in hours) required to complete all tasks.
19 hours

Assumptions: The tasks are executed in the order of their shortest to longest execution times using SJF. Each task starts immediately after the previous one is completed.

Explanation:

Using SJF, the tasks will be scheduled in the following order:

Task A (3 hours)

Task D (4 hours)

Task B (5 hours)

Task C (7 hours)

Thus, the total time required to complete all tasks is 19 hours.

```
def calculate_total_time(execution_times):
    # Write code here
    return total_time

# Driver Code
execution_times = [3, 5, 7, 4]
total_time = calculate_total_time(execution_times)

print(f"The total time required to complete all tasks is: {total_time} hours")
```

1.3 Managing a Restaurant's Orders

Imagine you are managing a busy restaurant where orders from different tables need to be processed. Each order requires a different amount of time to be prepared and served. To efficiently manage the kitchen staff and ensure timely service, you decide to implement the Round Robin scheduling algorithm for processing orders.

Here are the details of the current orders:

Order from Table 1: Burger and Fries, which takes 5 minutes to prepare.

Order from Table 2: Salad, which takes 3 minutes to prepare.

Order from Table 3: Pizza, which takes 8 minutes to prepare.

Order from Table 4: Pasta, which takes 6 minutes to prepare.

You decide to implement Round Robin with a time quantum of 4 minutes. This means each order will receive processing time in chunks of 4 minutes until it is completed or until it reaches its final processing time.

Your task is to simulate the Round Robin scheduling algorithm for processing these orders and calculate the total time required to complete all orders, considering the time quantum.

Input: Processing times of each order. Time quantum for Round Robin scheduling.

Order from Table 1: Burger and Fries (5 minutes)

Order from Table 2: Salad (3 minutes)

Order from Table 3: Pizza (8 minutes)

Order from Table 4: Pasta (6 minutes)

Time Quantum: 4 minutes

Output: The total time (in minutes) required to complete all orders.

22 minutes

Assumptions:

Each order starts processing immediately.

If an order cannot be completed within one quantum, it will continue in subsequent rounds.

Process orders in the order they were received.

Explanation:

Round 1:

Table 1 (Burger and Fries) - Processed for 4 minutes.

Table 2 (Salad) - Processed for 3 minutes (completed).

Table 3 (Pizza) - Processed for 4 minutes.

Table 4 (Pasta) - Processed for 4 minutes.

Round 2:

Table 1 (Burger and Fries) - Processed for remaining 1 minute.

Table 3 (Pizza) - Processed for remaining 4 minutes.

Table 4 (Pasta) - Processed for remaining 2 minutes (completed).

Thus, the total time required to complete all orders using Round Robin scheduling with a time quantum of 4 minutes is 22 minutes.

```
from collections import deque

def calculate_total_time(orders, time_quantum):
    # Write code here
    return total_time

# Driver Code
orders = [5, 3, 8, 6]
time_quantum = 4
total_time = calculate_total_time(orders, time_quantum)
print(f"The total time required to complete all orders is: {total_time} minutes")
```

1.4 Emergency Room Prioritization

You are managing an emergency room in a hospital where patients arrive with different medical conditions. Each patient requires a different level of urgency for treatment. To ensure critical cases are handled promptly, you decide to implement the Priority Scheduling algorithm for patient treatment.

Here are the details of the patients currently in the emergency room:

Patient A: Has suffered a heart attack and requires immediate attention, priority level 1.

Patient B: Has a broken arm and needs urgent treatment, priority level 2.

Patient C: Has a high fever and needs attention soon, priority level 3.

Patient D: Has a minor cut and can wait, priority level 4.

You decide to implement Priority Scheduling where patients with higher priority levels are treated first. If two patients have the same priority level, they are treated in the order they arrived.

Your task is to simulate the Priority Scheduling algorithm for treating these patients and calculate the total time required to treat all patients, considering their priority levels.

Input: Priority levels of each patient. Treatment times for each patient.

Patient A: Priority 1, Treatment time 10 minutes

Patient B: Priority 2, Treatment time 8 minutes

Patient C: Priority 3, Treatment time 15 minutes

Patient D: Priority 4, Treatment time 5 minutes

Output: The total time (in minutes) required to treat all patients.

38 minutes

Assumptions: Each patient starts treatment immediately upon arrival.

Patients are treated based on their priority levels (lower number = higher priority).

If two patients have the same priority level, they are treated in the order they arrived.

Explanation:

Patients are treated in the following order based on their priority:

Patient A (Priority 1, 10 minutes)

Patient B (Priority 2, 8 minutes)

Patient C (Priority 3, 15 minutes)

Patient D (Priority 4, 5 minutes)

Thus, the total time required to treat all patients using Priority Scheduling is 38 minutes.

```
from queue import PriorityQueue

def calculate_total_time(patients):
    # Write code here
    return total_time

# Drive Code
patients = [(1, 10),  # Patient A: Priority 1, Treatment time 10 minutes
            (2, 8),   # Patient B: Priority 2, Treatment time 8 minutes
            (3, 15),  # Patient C: Priority 3, Treatment time 15 minutes
            (4, 5)    # Patient D: Priority 4, Treatment time 5 minutes
           ]

total_time = calculate_total_time(patients)
print(f"The total time required to treat all patients is: {total_time} minutes")
```

2. Multi-Level Queue Scheduling

2.1 Managing System and User Processes

You are managing a computer system that runs various types of processes, categorized into system processes and user processes. System processes are critical for the functioning of the operating system and are given higher priority compared to user processes. To efficiently manage these processes, you decide to implement a Multi-Level Queue Scheduling algorithm with FCFS scheduling for each queue.

Here's the breakdown of the processes currently in the system:

System Process Queue:
Process A: Handles file system operations, takes 5 units of time to complete.
Process B: Manages memory allocation, takes 3 units of time to complete.
Process C: Performs system updates, takes 7 units of time to complete.

User Process Queue:
Process D: Runs a word processor, takes 4 units of time to complete.
Process E: Executes a media player, takes 2 units of time to complete.
Process F: Performs data analysis, takes 6 units of time to complete.

Implement the Multi-Level Queue Scheduling algorithm where system processes are scheduled with higher priority compared to user processes. Within each queue (system and user), processes are scheduled using FCFS.

Your task is to simulate the scheduling of these processes and calculate the total time required to complete all processes, considering the priorities and FCFS scheduling within each queue.

Input: Processing times for each process in the system and user queues.
System Processes:
Process A: 5 units
Process B: 3 units
Process C: 7 units

User Processes:
Process D: 4 units
Process E: 2 units
Process F: 6 units

Output: The total time (in units) required to complete all processes.
27 units

Assumptions: Each process starts execution immediately upon arrival. System processes have higher priority over user processes. FCFS scheduling is applied within each queue (system and user).

Explanation:
System Process Queue:
Process A (5 units)
Process B (3 units)
Process C (7 units)

User Process Queue:
Process D (4 units)
Process E (2 units)
Process F (6 units)
The total time required to complete all processes using Multi-Level Queue Scheduling with FCFS within each queue is 27 units.

```
from queue import Queue

def calculate_total_time(system_processes, user_processes):
    # Write code here
    return total_time

# Driver Code
system_processes = {'Process A': 5,'Process B': 3,'Process C': 7,}

user_processes = {'Process D': 4, 'Process E': 2, 'Process F': 6,}

total_time = calculate_total_time (system_processes, user_processes)
print(f"The total time required to complete all processes is: {total_time} units")
```

2.2 Managing Job Scheduling in a Computing Center

You are managing a computing center that processes jobs submitted by various departments of an organization. Jobs are categorized into two priority levels: high priority (system jobs) and low priority (user jobs). Each job has a specific processing time required for completion.

Here are the details of the jobs currently in the system:

High Priority (System) Jobs:
Job A: Performs critical database backups, takes 8 units of time to complete.
Job B: Handles real-time monitoring tasks, takes 5 units of time to complete.
Job C: Executes system updates, takes 10 units of time to complete.

Low Priority (User) Jobs:
Job D: Runs periodic report generation, takes 6 units of time to complete.
Job E: Processes email delivery, takes 3 units of time to complete.
Job F: Executes batch data processing, takes 7 units of time to complete.

Implement the Multi-Level Queue Scheduling algorithm where high priority (system) jobs are scheduled with higher priority compared to low priority (user) jobs. Within each queue (high and low priority), jobs are scheduled using FCFS.

Your task is to simulate the scheduling of these jobs and calculate the total time required to complete all jobs, considering the priorities and FCFS scheduling within each queue.

Input: Processing times for each job in the high and low priority queues.
High Priority Jobs:
Job A: 8 units
Job B: 5 units
Job C: 10 units

Low Priority Jobs:
Job D: 6 units

Job E: 3 units
Job F: 7 units

Output: The total time (in units) required to complete all jobs.
39 units
Assumptions: Each job starts execution immediately upon arrival. High priority (system) jobs have higher priority over low priority (user) jobs. FCFS scheduling is applied within each queue (high and low priority).

Explanation:
High Priority (System) Jobs:
Job A (8 units)
Job B (5 units)
Job C (10 units)

Low Priority (User) Jobs:
Job D (6 units)
Job E (3 units)
Job F (7 units)

The total time required to complete all jobs using Multi-Level Queue Scheduling with FCFS within each queue is 39 units.

```
from queue import Queue

def calculate_total_time(high_priority_jobs, low_priority_jobs):
    # Write code here
    return total_time

# Write code here
high_priority_jobs = {'Job A': 8, 'Job B': 5, 'Job C': 10,}

low_priority_jobs = {'Job D': 6, 'Job E': 3, 'Job F': 7,}

total_time = calculate_total_time(high_priority_jobs, low_priority_jobs)

print(f"The total time required to complete all jobs is: {total_time} units")
```

2.3 Managing Print Jobs in a Shared Printing Environment

You are managing a shared printing environment where print jobs from different departments are queued up for processing. Print jobs are categorized into two priority levels: high priority (system jobs) and low priority (user jobs). Each job has a specific processing time required for printing.

Here are the details of the print jobs currently in the system:

High Priority (System) Print Jobs:
Job A: Urgent report for management, takes 15 units of time to print.
Job B: Critical financial statement, takes 10 units of time to print.
Job C: Emergency document for legal department, takes 20 units of time to print.

Low Priority (User) Print Jobs:
Job D: Regular office memo, takes 5 units of time to print.
Job E: Marketing flyer, takes 8 units of time to print.
Job F: Personal document for an employee, takes 3 units of time to print.

Implement the Multi-Level Queue Scheduling algorithm where high priority (system) print jobs are scheduled with higher priority compared to low priority (user) print jobs. Within each queue (high and low priority), print jobs are scheduled using FCFS.

Your task is to simulate the scheduling of these print jobs and calculate the total time required to complete all print jobs, considering the priorities and FCFS scheduling within each queue.

Input: Printing times for each job in the high and low priority queues.
High Priority Print Jobs:
Job A: 15 units
Job B: 10 units
Job C: 20 units

Low Priority Print Jobs:
Job D: 5 units
Job E: 8 units
Job F: 3 units

Output: The total time (in units) required to complete all print jobs.
61 units

Assumptions: Each print job starts printing immediately upon arrival. High priority (system) print jobs have higher priority over low priority (user) print jobs. FCFS scheduling is applied within each queue (high and low priority).

Explanation:

High Priority (System) Print Jobs:
Job A (15 units)
Job B (10 units)
Job C (20 units)

Low Priority (User) Print Jobs:
Job D (5 units)
Job E (8 units)
Job F (3 units)

The total time required to complete all print jobs using Multi-Level Queue Scheduling with FCFS within each queue is 61 units.

```
from queue import Queue

def calculate_total_time(high_priority_jobs, low_priority_jobs):
```

```
    # Write code here
    return total_time

# Driver Code
high_priority_jobs = {'Job A': 15, 'Job B': 10, 'Job C': 20,}

low_priority_jobs = {'Job D': 5, 'Job E': 8, 'Job F': 3,}

total_time = calculate_total_time(high_priority_jobs, low_priority_jobs)

print(f"The total time required to complete all print jobs is: {total_time} units")
```

2.4 Task Scheduling in a Multi-User System

You are managing a multi-user system where tasks from different users need to be scheduled for processing. Tasks are categorized into two priority levels: high priority (system tasks) and low priority (user tasks). Each task has a specific processing time required for completion.

Here are the details of the tasks currently in the system:
High Priority (System) Tasks:
Task A: Performs critical system maintenance, takes 12 units of time to complete.
Task B: Executes essential security updates, takes 8 units of time to complete.
Task C: Handles real-time data processing, takes 15 units of time to complete.
Low Priority (User) Tasks:
Task D: Runs a background data synchronization process, takes 6 units of time to complete.
Task E: Processes user-generated reports, takes 4 units of time to complete.
Task F: Executes regular data backups, takes 10 units of time to complete.

Implement the Multi-Level Queue Scheduling algorithm where high priority (system) tasks are scheduled with higher priority compared to low priority (user) tasks. Within each queue (high and low priority), tasks are scheduled using FCFS.

Your task is to simulate the scheduling of these tasks and calculate the total time required to complete all tasks, considering the priorities and FCFS scheduling within each queue.

Input: Processing times for each task in the high and low priority queues.
High Priority Tasks:
Task A: 12 units
Task B: 8 units
Task C: 15 units

Low Priority Tasks:
Task D: 6 units
Task E: 4 units
Task F: 10 units

Output: The total time (in units) required to complete all tasks.
55 units

Assumptions: Each task starts processing immediately upon arrival. High priority (system) tasks have higher priority over low priority (user) tasks. FCFS scheduling is applied within each queue (high and low priority).

Explanation:
High Priority (System) Tasks:
Task A (12 units)
Task B (8 units)
Task C (15 units)

Low Priority (User) Tasks:
Task D (6 units)
Task E (4 units)
Task F (10 units)

The total time required to complete all tasks using Multi-Level Queue Scheduling with FCFS within each queue is 55 units.

```
from queue import Queue

def calculate_total_time(high_priority_tasks, low_priority_tasks):
    # Write code here
    return total_time

# Driver Code
high_priority_tasks = {'Task A': 12, 'Task B': 8, 'Task C': 15,}

low_priority_tasks = {'Task D': 6, 'Task E': 4, 'Task F': 10, }

total_time = calculate_total_time(high_priority_tasks, low_priority_tasks)

print(f"The total time required to complete all tasks is: {total_time} units")
```

2.5 Job Scheduling in a Computing Cluster

You are managing a computing cluster that processes jobs submitted by various departments of a research institution. Jobs are categorized into three priority levels: high priority (critical jobs), medium priority (standard jobs), and low priority (background jobs). Each job has a specific processing time required for completion.

Here are the details of the jobs currently in the system:

High Priority (Critical) Jobs:
Job A: Performs complex simulations for a time-sensitive research project, takes 20 units of time to complete.
Job B: Analyzes large datasets for an urgent analysis, takes 15 units of time to complete.
Job C: Executes critical experiments with strict deadlines, takes 25 units of time to complete.

Medium Priority (Standard) Jobs:

Job D: Runs routine data processing tasks, takes 10 units of time to complete.
Job E: Processes experimental results for ongoing studies, takes 12 units of time to complete.
Job F: Executes regular system maintenance tasks, takes 8 units of time to complete.

Low Priority (Background) Jobs:
Job G: Performs non-critical data backups, takes 5 units of time to complete.
Job H: Runs periodic log file analysis, takes 4 units of time to complete.
Job I: Executes routine software updates, takes 6 units of time to complete.

Implement the Multi-Level Queue Scheduling algorithm where jobs are scheduled based on their priority levels: high priority jobs are scheduled first, followed by medium priority jobs, and then low priority jobs. Within each priority level, jobs are scheduled using FCFS.

Your task is to simulate the scheduling of these jobs and calculate the total time required to complete all jobs, considering the priorities and FCFS scheduling within each priority level.

Input: Processing times for each job in the high, medium, and low priority queues.
High Priority Jobs:
Job A: 20 units
Job B: 15 units
Job C: 25 units

Medium Priority Jobs:
Job D: 10 units
Job E: 12 units
Job F: 8 units

Low Priority Jobs:
Job G: 5 units
Job H: 4 units
Job I: 6 units

Output: The total time (in units) required to complete all jobs.
105 units

Assumptions: Each job starts execution immediately upon arrival. Jobs within higher priority levels have higher priority over jobs in lower priority levels. FCFS scheduling is applied within each priority level.

Explanation:
High Priority Jobs:
Job A (20 units)
Job B (15 units)
Job C (25 units)

Medium Priority Jobs:
Job D (10 units)
Job E (12 units)
Job F (8 units)

Low Priority Jobs:
Job G (5 units)
Job H (4 units)
Job I (6 units)

The total time required to complete all jobs using Multi-Level Queue Scheduling with FCFS within each priority level is 105 units.

```
from queue import Queue

def calculate_total_time(high_priority_jobs, medium_priority_jobs,
low_priority_jobs):
    # Write code here
    return total_time

# Driver Code
high_priority_jobs = {'Job A': 20, 'Job B': 15, 'Job C': 25,}

medium_priority_jobs = {'Job D': 10, 'Job E': 12, 'Job F': 8,}

low_priority_jobs = {'Job G': 5, 'Job H': 4, 'Job I': 6,}

total_time = calculate_total_time(high_priority_jobs, medium_priority_jobs,
low_priority_jobs)

print(f"The total time required to complete all jobs is: {total_time} units")
```

3. File Allocation Strategies

3.1 Managing Student Records in a School Database

You are tasked with designing a file management system for a school's student database. The database contains records of students' personal information, academic performance, and attendance history. Each student record is stored as a file with specific attributes.

Implement the Sequential File Allocation strategy to manage the storage and access of student records in the school database.
Here's the scenario of the school database system:

The school has records for three students:
Student A: John Doe, ID 101, Grade 10, Address: 123 Main Street
Student B: Jane Smith, ID 102, Grade 11, Address: 456 Elm Street
Student C: Michael Brown, ID 103, Grade 9, Address: 789 Oak Avenue

Implement and simulate the Sequential File Allocation strategy for storing and accessing these student records in the school database. Consider the following aspects:

File Structure: Each student record contains fields for Name, ID, Grade, and Address.
Storage: Files are stored sequentially one after another in contiguous blocks of disk space.

Access: A file allocation table (FAT) keeps track of the starting block address and the length of each student record file.

Your task is to design a data structure or representation for the student records. Implement the Sequential File Allocation strategy to store and retrieve student records. Calculate the total disk space used and evaluate the efficiency of the strategy based on storage utilization and access times.

Assumptions: Disk space is divided into fixed-size blocks. Each student record is stored in whole blocks (no partial blocks).

The Sequential File Allocation strategy should handle new record insertions, deletions, and updates efficiently, considering disk fragmentation and access patterns. To simulate the Sequential File Allocation strategy for managing student records in a school database, we'll create a Python program. In this program, we'll represent each student record as a file with fields for Name, ID, Grade, and Address. We'll use a list to simulate disk blocks and a file allocation table (FAT) to keep track of file locations.

```
class StudentRecord:
    def __init__(self, name, student_id, grade, address):
        self.name = name
        self.student_id = student_id
        self.grade = grade
        self.address = address

    def __str__(self):
        return f"Name: {self.name}, ID: {self.student_id}, Grade: {self.grade},
Address: {self.address}"

class SequentialFileAllocation:
    def __init__(self, block_size):
        self.disk_blocks = []
        self.fat = {}
        self.block_size = block_size
        self.next_free_block = 0

    def add_record(self, student_record):
        # Write code here
        return True

    def get_record(self, student_id):
        # Write code here
        return None

    def calculate_record_size(self, student_record):
         # Write code here
        return 1

    def print_disk_status(self):
        # Write code here

# Driver Code
block_size = 1
```

```
file_system = SequentialFileAllocation(block_size)
student_records = [
        StudentRecord ("John Doe", 101, 10, "123 Main Street"),
        StudentRecord ("Jane Smith", 102, 11, "456 Elm Street"),
        StudentRecord ("Michael Brown", 103, 9, "789 Oak Avenue")
    ]
for record in student_records:
    file_system.add_record (record)

file_system.print_disk_status()

student_id = 102
retrieved_record = file_system.get_record (student_id)
if retrieved_record:
   print(f"\n Retrieved Record for ID {student_id}:")
   print(retrieved_record)
else:
   print(f"\n Record with ID {student_id} not found.")
```

3.2 Managing Medical Records in a Hospital Information System

You have been tasked with designing a file management system for a hospital's electronic medical records (EMR). The EMR system contains detailed records of patients' medical histories, treatments, and diagnostic results. Each patient record is stored as a file with specific attributes.

Implement the Indexed File Allocation strategy to efficiently manage the storage and access of patient records in the hospital's EMR system.

Here's the scenario of the hospital information system:

The hospital manages records for several patients:
Patient A: John Smith, Age 45, Medical ID 1001, Address: 123 Hospital Road
Patient B: Jane Doe, Age 32, Medical ID 1002, Address: 456 Clinic Avenue
Patient C: Michael Johnson, Age 58, Medical ID 1003, Address: 789 Medical Plaza

Implement and simulate the Indexed File Allocation strategy for storing and accessing these patient records in the hospital's EMR system. Consider the following aspects:

File Structure: Each patient record contains fields for Name, Age, Medical ID, and Address.
Storage: Each patient record has its own index block that contains pointers to the actual blocks of disk space where its data is stored.
Index Table: Maintain an index table that maps each patient record to its respective index block, facilitating efficient access.
Access: Support operations such as insertion, deletion, and retrieval of patient records based on their Medical ID.

Your task is to design a data structure or representation for patient records and index blocks. Implement the Indexed File Allocation strategy to store and retrieve patient records. Calculate the total disk space used and evaluate the efficiency of the strategy based on storage utilization and access times.

Assumptions: Disk space is divided into fixed-size blocks. Each patient record is stored in whole blocks (no partial blocks). The Indexed File Allocation strategy should handle new record insertions, deletions, and updates efficiently, considering disk fragmentation and access patterns.

To implement the Indexed File Allocation strategy for managing patient records in a hospital's electronic medical records (EMR) system, we'll create a Python program. This program will simulate the storage and retrieval of patient records using an index table and index blocks to efficiently manage disk space and access.

```
class PatientRecord:
    def __init__(self, name, age, medical_id, address):
        self.name = name
        self.age = age
        self.medical_id = medical_id
        self.address = address

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}, Medical ID: {self.medical_id},
Address: {self.address}"

class IndexedFileAllocation:
    def __init__(self, block_size):
        self.disk_blocks = []
        self.index_table = {}
        self.index_block_size = block_size
        self.next_free_block = 0

    def add_record(self, patient_record):
        # Write code here
        return True

    def get_record(self, medical_id):
        # Write code here
        return None

    def calculate_record_size(self, patient_record):
        # Write code here
        return 1

    def print_disk_status(self):
        # Write code here

# Driver Code
index_block_size = 1
file_system = IndexedFileAllocation(index_block_size)

patient_records = [
        PatientRecord("John Smith", 45, 1001, "123 Hospital Road"),
        PatientRecord("Jane Doe", 32, 1002, "456 Clinic Avenue"),
        PatientRecord("Michael Johnson", 58, 1003, "789 Medical Plaza")
    ]
```

```
for record in patient_records:
    file_system.add_record(record)

file_system.print_disk_status()

medical_id = 1002
retrieved_record = file_system.get_record(medical_id)
if retrieved_record:
   print(f"\nRetrieved Record for Medical ID {medical_id}:")
   print(retrieved_record)
else:
   print(f"\nRecord with Medical ID {medical_id} not found.")
```

3.3 Managing Digital Media Files in a Multimedia Application

You are developing a multimedia application that allows users to manage and store various types of digital media files, including images, videos, and audio clips. Each file is treated as a separate entity with its own attributes and content.

Implement the Linked File Allocation strategy to efficiently manage the storage and access of digital media files in your multimedia application.

Here's the scenario of the multimedia application:
The application needs to manage the following digital media files:

File A: Landscape.jpg (Image file), size 5 MB
File B: Concert.mp4 (Video file), size 50 MB
File C: Song.mp3 (Audio file), size 8 MB

Implement and simulate the Linked File Allocation strategy for storing and accessing these digital media files in the multimedia application. Consider the following aspects:

File Structure: Each digital media file has specific attributes (e.g., file name, type, size) and content.
Storage: Files are stored in non-contiguous blocks of disk space, and each block contains a pointer to the next block of the file.

File Allocation Table (FAT): Maintain a FAT to keep track of the starting block address of each file.
Access: Support operations such as insertion, deletion, and retrieval of files based on their attributes.

Your task is to design a data structure or representation for digital media files and disk blocks. Implement the Linked File Allocation strategy to store and retrieve digital media files. Calculate the total disk space used and evaluate the efficiency of the strategy based on storage utilization and access times.

Assumptions: Disk space is divided into blocks of fixed size. Each digital media file is stored in multiple blocks using pointers to link consecutive blocks. The Linked File Allocation strategy should handle new file insertions, deletions, and updates efficiently, considering disk fragmentation and access patterns.

```
from threading import Semaphore
```

```
class DiskBlock:
    def __init__(self, block_id):
        self.block_id = block_id
        self.next_block = None


class MediaFile:
    def __init__(self, file_name, file_type, size):
        self.file_name = file_name
        self.file_type = file_type
        self.size = size
        self.start_block = None  # Points to the first block in the chain

    def __str__(self):
        return f"{self.file_name} ({self.file_type}), Size: {self.size} MB"


class FileAllocationTable:
    def __init__(self):
        self.fat = {}

    def allocate_blocks(self, media_file, disk_blocks):
        # Write code here

    def delete_file(self, media_file, disk_blocks):
         # Write code here

def simulate_linked_file_allocation():
    # Write code here

# Driver Code
simulate_linked_file_allocation()
```

3.4 DigitalArchive

You are a software engineer working for a digital library company called "DigitalArchive". DigitalArchive specializes in storing and managing vast collections of digital media files for various clients, including museums, libraries, and private collectors. Your task is to design a file allocation strategy that efficiently manages these files on DigitalArchive's storage system.

DigitalArchive has recently signed a contract with a renowned museum that wants to digitize and archive its extensive collection of historical photographs. The museum's collection includes thousands of photographs ranging from early 20th-century portraits to landscapes and architectural images.

Requirements:
File Types: Each photograph is stored as a digital image file (e.g., JPG format).
File Sizes: The sizes of photographs vary, with average file sizes ranging from 5 MB to 20 MB.
Storage System: DigitalArchive uses a storage system that organizes files into blocks of fixed size (e.g., 10 MB per block).
File Allocation Strategy: Sequential File Allocation

Sequential Allocation: In this strategy, files are stored sequentially on the storage system. When a new file arrives, it is allocated blocks of disk space one after another. Each file is assigned contiguous blocks of the required size.

Tasks to Implement: Design a program or algorithm that simulates the sequential file allocation strategy for storing the museum's photograph collection. Implement functions to allocate blocks for new photographs, manage the allocation of disk space, and retrieve information about allocated files. Ensure the program handles scenarios such as adding new photographs, calculating total disk space used, and displaying the allocation status.

Assume the storage system has 100 blocks of 10 MB each. Implement a function to allocate blocks sequentially for new photographs based on their sizes. Display the allocation status after each operation (e.g., adding a new photograph, deleting a photograph).

Additional Considerations: Evaluate the efficiency of the sequential allocation strategy in terms of disk space utilization and access times for retrieving files. Consider how fragmentation might affect storage efficiency over time as more files are added and deleted.

```
class DigitalArchive:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.disk_blocks = [False] * total_blocks

    def allocate_blocks(self, file_name, file_size):
        # Write code here

    def delete_file(self, file_name):
        # Write code here

    def calculate_blocks_needed(self, file_size):
        # Write code here

    def display_allocation_status(self):
        # Write code here

# Driver Code
digital_archive = DigitalArchive (100)

digital_archive.allocate_blocks ("Portrait1.jpg", 15)
digital_archive.allocate_blocks ("Landscape1.jpg", 10)
digital_archive.allocate_blocks ("Architecture1.jpg", 7)

digital_archive.display_allocation_status()

digital_archive.delete_file("Landscape1.jpg")

print("\nAfter deleting Landscape1.jpg:")
digital_archive.display_allocation_status()
```

3.5 EnterpriseX

You are a systems engineer working for a large multinational corporation that specializes in developing operating systems for enterprise-level servers. Your team is tasked with designing a file allocation strategy that optimizes data access and storage efficiency for the company's new flagship operating system, OS EnterpriseX.

OS EnterpriseX is designed to handle large-scale data processing and storage for corporate clients. One of the critical requirements is to efficiently manage and access large files, such as databases and multimedia content, stored on disk drives. The system needs to ensure quick access to specific data blocks while maintaining efficient storage utilization.

Requirements:
File Types: Files managed by OS EnterpriseX include database files (structured data) and multimedia files (e.g., videos, images).
Storage System: The operating system uses a disk storage system divided into blocks of fixed size (e.g., 8 KB per block).
Index Sequential File Allocation Strategy: This strategy involves maintaining an index (typically a B-tree or a hash table) that allows direct access to data blocks associated with each file. The index helps in quickly locating the starting block of a file and its subsequent blocks, providing efficient data retrieval.
File Allocation Strategy: Index Sequential File Allocation
Index Management: Maintain an index structure that maps each file to its corresponding disk blocks. The index allows for direct access to specific blocks using file identifiers or keys.
Sequential Access: Files are stored sequentially in terms of logical organization, but physical storage is optimized using the index structure.
Tasks to Implement:

Design a program or algorithm that simulates the index sequential file allocation strategy for managing and accessing files in OS EnterpriseX. Implement functions to allocate blocks for new files, manage the index structure, retrieve data blocks based on file identifiers, and display the current allocation status.

Evaluate the efficiency of the index sequential strategy in terms of access times and storage utilization compared to other strategies (e.g., purely sequential or direct allocation). Assume the storage system has 1000 blocks of 8 KB each. Implement functions to allocate blocks using the index sequential strategy based on file sizes and manage the index structure for quick access.

Consider scenarios where files are updated, deleted, or new files are added, and evaluate how the index structure adapts to these operations. Evaluate the impact of fragmentation and disk space utilization with the index sequential strategy.

```
class OperatingSystem:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.disk_blocks = [False] * total_blocks  # False indicates block is free
        self.index = {}  # Index to map file identifiers to disk blocks

    def allocate_blocks(self, file_name, file_size):
        # Write code here
```

```
    def delete_file(self, file_name):
        # Write code here

    def find_free_blocks(self, required_blocks):
        # Write code here

    def calculate_blocks_needed(self, file_size):
        # Write code here

    def total_free_blocks(self):
        # Write code here

    def display_allocation_status(self):
        # Write code here

# Driver Code
os = OperatingSystem(1000)

os.allocate_blocks ("database.db", 300)
os.allocate_blocks ("video.mp4", 600)
os.allocate_blocks ("image.jpg", 150)

os.display_allocation_status()
os.delete_file("video.mp4")

print("\n After deleting video.mp4:")
os.display_allocation_status()
```

4. Memory Management

4.1 Memory Variable Technique (MVT)

You are a system administrator responsible for managing a computer system that employs the Memory Variable Technique (MVT) for memory management. Your system receives requests from multiple users to run different processes, each requiring a specific amount of memory.

Your system has a total of 1000 KB of memory available, divided into fixed-size partitions to accommodate different process sizes. The Memory Variable Technique dynamically allocates partitions to processes based on their size, aiming to minimize internal fragmentation and optimize memory usage.

Memory Partitions: The system has three memory partitions with sizes as follows:
Partition 1: 300 KB
Partition 2: 500 KB
Partition 3: 200 KB

Process Requests: Users submit requests to run processes of varying sizes (in KB).

Memory Allocation: Implement algorithms to allocate memory to processes based on the MVT strategy:
Allocate memory to a process in the smallest partition that can accommodate its size.
Track and display the allocation status after each operation (allocation or deallocation).

Tasks to Implement:
Design a program or algorithm that simulates the Memory Variable Technique (MVT) for memory allocation. Implement functions to allocate memory to processes, manage the allocation status, deallocate memory when processes finish, and display the current memory allocation status. Evaluate the efficiency of the MVT strategy in terms of memory utilization and fragmentation management.

Implement functions to handle memory allocation requests from users based on process sizes and available partitions. Display the memory allocation status after each operation (allocation or deallocation).

Additional Considerations:
Consider scenarios where memory requests exceed available partition sizes and implement appropriate handling (e.g., fragmentation management, rejection of oversized requests).

```
class MemoryVariableTechnique:
    def __init__(self, partitions):
        self.partitions = partitions
        self.memory_map = {1: [False] * partitions[1],
                           2: [False] * partitions[2],
                           3: [False] * partitions[3]}
        self.processes = {}

    def allocate_memory(self, process_id, size):
        # Write code here

    def deallocate_memory(self, process_id):
        # Write code here

    def display_memory_status(self):
        # Write code here

# Driver Code
mvt = MemoryVariableTechnique({1: 300, 2: 500, 3: 200})

mvt.allocate_memory(1, 150)
mvt.allocate_memory(2, 400)
mvt.allocate_memory(3, 100)

mvt.display_memory_status()

mvt.deallocate_memory(2)

print("\nAfter deallocating Process 2:")
mvt.display_memory_status()
```

4.2 Best Fit Memory Allocation

You are a software developer working on an operating system project that requires implementing memory management techniques. One of the critical components is the Best Fit memory allocation algorithm, which aims to allocate memory blocks to processes in a way that minimizes wastage and fragmentation.

Your operating system is designed to manage a system with a total of 800 KB of memory available. Processes of varying sizes are submitted to the system for execution, each requiring a specific amount of memory. Your task is to implement and simulate the Best Fit memory allocation algorithm to efficiently allocate memory blocks to these processes.

Memory Blocks: The system has a total of 8 memory blocks available, each with a fixed size.
Block 1: 100 KB
Block 2: 200 KB
Block 3: 50 KB
Block 4: 150 KB
Block 5: 300 KB
Block 6: 80 KB
Block 7: 120 KB
Block 8: 200 KB
Process Requests: Users submit requests to run processes of varying sizes (in KB).
Best Fit Algorithm: Implement the Best Fit algorithm to allocate the smallest possible block that can accommodate each process size. If no suitable block is found, reject the process request.
Memory Utilization: Track and display the allocation status after each operation (allocation or rejection).
Tasks to Implement:

Design a program or algorithm that simulates the Best Fit memory allocation strategy. Implement functions to allocate memory blocks to processes, manage the allocation status, reject processes when appropriate, and display the current memory allocation status. Evaluate the efficiency of the Best Fit algorithm in terms of memory utilization and fragmentation management.

Implement functions to handle memory allocation requests from users based on process sizes and available memory blocks. Display the memory allocation status after each operation (allocation or rejection).

Additional Considerations:
Consider scenarios where multiple requests are submitted simultaneously and evaluate how the Best Fit algorithm manages memory blocks efficiently. Evaluate the impact of fragmentation and memory utilization with the Best Fit algorithm compared to other allocation strategies.

```
class BestFitMemoryManager:
    def __init__(self, memory_blocks):
        self.memory_blocks = memory_blocks
        self.available_blocks = {i: block for i, block in enumerate(memory_blocks)}
        self.processes = {}

    def allocate_memory(self, process_id, size):
        # Write code here

    def deallocate_memory(self, process_id):
        # Write code here

    def display_memory_status(self):
        # Write code here

# Driver Code
```

```
memory_manager = BestFitMemoryManager([100, 200, 50, 150, 300, 80, 120, 200])

memory_manager.allocate_memory(1, 70)
memory_manager.allocate_memory(2, 180)
memory_manager.allocate_memory(3, 250)

memory_manager.display_memory_status()

memory_manager.deallocate_memory(2)

print("\nAfter deallocating Process 2:")
memory_manager.display_memory_status()
```

4.3 Worst Fit Memory Allocation

You are a system analyst tasked with designing memory management techniques for a new operating system. One of the techniques to implement is the Worst Fit memory allocation algorithm, which prioritizes allocating the largest available memory block to processes.

Your operating system is designed to manage a system with a total of 1200 KB of memory available, divided into fixed-size memory blocks. Processes of varying sizes are submitted to the system for execution, each requiring a specific amount of memory. Your task is to implement and simulate the Worst Fit memory allocation algorithm to efficiently allocate memory blocks to these processes.

Memory Blocks: The system has a total of 10 memory blocks available, each with a fixed size.
Block 1: 150 KB
Block 2: 300 KB
Block 3: 100 KB
Block 4: 200 KB
Block 5: 250 KB
Block 6: 50 KB
Block 7: 350 KB
Block 8: 180 KB
Block 9: 120 KB
Block 10: 200 KB
Process Requests: Users submit requests to run processes of varying sizes (in KB).
Worst Fit Algorithm: Implement the Worst Fit algorithm to allocate the largest possible block that can accommodate each process size. If no suitable block is found, reject the process request.
Memory Utilization: Track and display the allocation status after each operation (allocation or rejection).

Tasks to Implement: Design a program or algorithm that simulates the Worst Fit memory allocation strategy. Implement functions to allocate memory blocks to processes, manage the allocation status, reject processes when appropriate, and display the current memory allocation status. Evaluate the efficiency of the Worst Fit algorithm in terms of memory utilization and fragmentation management.

Implement functions to handle memory allocation requests from users based on process sizes and available memory blocks. Display the memory allocation status after each operation (allocation or rejection).
Additional Considerations:

Consider scenarios where multiple requests are submitted simultaneously and evaluate how the Worst Fit algorithm manages memory blocks efficiently. Evaluate the impact of fragmentation and memory utilization with the Worst Fit algorithm compared to other allocation strategies.

```
class WorstFitMemoryManager:
    def __init__(self, memory_blocks):
        self.memory_blocks = memory_blocks
        self.available_blocks = {i: block for i, block in enumerate(memory_blocks)}
        self.processes = {}

    def allocate_memory(self, process_id, size):
        # Write code here

    def deallocate_memory(self, process_id):
        # Write code here

    def display_memory_status(self):
        # Write code here

# Driver Code
memory_manager = WorstFitMemoryManager([150, 300, 100, 200, 250, 50, 350, 180, 120,
200])

memory_manager.allocate_memory(1, 180)
memory_manager.allocate_memory(2, 400)
memory_manager.allocate_memory(3, 120)

memory_manager.display_memory_status()

memory_manager.deallocate_memory(2)

print("\nAfter deallocating Process 2:")
memory_manager.display_memory_status()
```

4.4 Multiprogramming with a Fixed Number of Tasks (MFT)

You are a systems engineer working on a legacy mainframe system that employs the MFT memory management technique. The system is designed to handle multiple processes simultaneously by dividing memory into fixed-size partitions, each capable of holding a single process.

Your mainframe system has a total of 6400 KB of memory available, divided into fixed-size partitions. Processes of varying sizes are submitted to the system for execution, each requiring a specific amount of memory. Your task is to implement and simulate the MFT memory management technique to efficiently allocate memory partitions to these processes.

Memory Partitions: The system memory is divided into fixed-size partitions to accommodate processes.
Partition Size: 800 KB each.
Total Partitions: 8 partitions in total.
Process Requests: Users submit requests to run processes of varying sizes (in KB).
MFT Algorithm: Implement the MFT algorithm to allocate the smallest available partition that can accommodate each process size. If no suitable partition is found, reject the process request.

Memory Utilization: Track and display the allocation status after each operation (allocation or rejection).
Tasks to Implement:

Design a program or algorithm that simulates the MFT memory management technique. Implement functions to allocate memory partitions to processes, manage the allocation status, reject processes when appropriate, and display the current memory allocation status. Evaluate the efficiency of the MFT technique in terms of memory utilization and management of partitions.

Implement functions to handle memory allocation requests from users based on process sizes and available partitions. Display the memory allocation status after each operation (allocation or rejection).

Additional Considerations:
Consider scenarios where multiple requests are submitted simultaneously and evaluate how the MFT algorithm manages partitions efficiently. Evaluate the impact of fragmentation and memory utilization with the MFT technique compared to other allocation strategies.

```
class MFTMemoryManager:
    def __init__(self, num_partitions, partition_size):
        self.num_partitions = num_partitions
        self.partition_size = partition_size
        self.available_partitions = [True] * num_partitions
        self.processes = {}

    def allocate_memory(self, process_id, size):
        # Write code here

    def deallocate_memory(self, process_id):
        # Write code here

    def display_memory_status(self):
        # Write code here

# Driver Code
memory_manager = MFTMemoryManager(num_partitions=8, partition_size=800)

memory_manager.allocate_memory(1, 600)
memory_manager.allocate_memory(2, 900)
memory_manager.allocate_memory(3, 400)

memory_manager.display_memory_status()

memory_manager.deallocate_memory(2)

print("\nAfter deallocating Process 2:")
memory_manager.display_memory_status()
```

4.5 Simulating Paging Memory Management

You are a software engineer tasked with implementing memory management for a new operating system that utilizes the Paging technique. Paging divides the physical memory into fixed-size blocks (pages) and manages processes by allocating these pages dynamically.

Your operating system has a total of 4000 KB of physical memory available, divided into fixed-size pages. Processes of varying sizes are submitted to the system for execution, each requiring a specific amount of memory. Your task is to implement and simulate the Paging memory management technique to efficiently allocate memory pages to these processes.

Memory Pages: The system memory is divided into fixed-size pages to accommodate processes.

Page Size: 200 KB each.

Total Pages: 20 pages in total.

Process Requests: Users submit requests to run processes of varying sizes (in KB).

Paging Algorithm: Implement the Paging algorithm to allocate the required number of pages to each process dynamically. If no sufficient contiguous pages are available, reject the process request.

Memory Utilization: Track and display the allocation status after each operation (allocation or rejection).

Design a program or algorithm that simulates the Paging memory management technique. Implement functions to allocate memory pages to processes, manage the allocation status, reject processes when appropriate, and display the current memory allocation status. Evaluate the efficiency of the Paging technique in terms of memory utilization and management of pages.

Implement functions to handle memory allocation requests from users based on process sizes and available pages. Display the memory allocation status after each operation (allocation or rejection).

Additional Considerations:

Consider scenarios where multiple requests are submitted simultaneously and evaluate how the Paging algorithm manages pages efficiently. Evaluate the impact of fragmentation and memory utilization with the Paging technique compared to other allocation strategies.

```
class PagingMemoryManager:
    def __init__(self, num_pages, page_size):
        self.num_pages = num_pages
        self.page_size = page_size
        self.available_pages = [True] * num_pages
        self.processes = {}

    def allocate_memory(self, process_id, size):
        # Write code here

    def deallocate_memory(self, process_id):
        # Write code here

    def display_memory_status(self):
        # Write code here

# Driver Code
```

```
memory_manager = PagingMemoryManager(num_pages=20, page_size=200)

memory_manager.allocate_memory(1, 400)
memory_manager.allocate_memory(2, 600)
memory_manager.allocate_memory(3, 300)

memory_manager.display_memory_status()

memory_manager.deallocate_memory(2)

print("\nAfter deallocating Process 2:")
memory_manager.display_memory_status()
```

5. Contiguous Memory Allocation

5.1 Futuristic Space Station

In a futuristic space station, the central computer manages memory allocation using the worst-fit technique. The memory is divided into fixed-size blocks, and whenever a program requests memory allocation, the system searches for the largest available block that can accommodate the program's size. One day, the space station receives requests from three different programs:

Program A requires 150 units of memory.

Program B requires 300 units of memory.

Program C requires 200 units of memory.

The memory blocks available in the system are as follows:

Block 1: 400 units

Block 2: 250 units

Block 3: 350 units

Block 4: 200 units

Block 5: 150 units

Assume the worst-fit algorithm starts its search from the beginning of the memory and scans sequentially to find the largest block that fits each program's requirement.

Task: Determine how the worst-fit algorithm assigns memory blocks to each program, and calculate the remaining memory after all programs are allocated.

Explanation:

Program A (150 units): The worst-fit algorithm will select Block 1 (400 units) since it's the largest block available that can accommodate Program A. After allocating 150 units to Program A, Block 1 will have 250 units remaining.

Program B (300 units): Now, the system will search for the largest block that fits Program B's requirement.

The largest block available is Block 3 (350 units). After allocating 300 units to Program B, Block 3 will have 50 units remaining.

Program C (200 units): Finally, Program C needs a block of 200 units. The largest block available now is Block 2 (250 units). After allocating 200 units to Program C, Block 2 will have 50 units remaining.

Remaining Memory:

Block 1: 250 units (after Program A)

Block 2: 50 units (after Program C)

Block 3: 50 units (after Program B)

Block 4: 200 units

Block 5: 150 units

Thus, after allocating memory to all three programs using the worst-fit algorithm, the space station's computer system has 250 units + 50 units + 50 units = 350 units of memory remaining.

```
def worst_fit_allocation(memory_blocks, program_requests):
    # Write code here

# Driver Code
memory_blocks = [400, 250, 350, 200, 150]

program_requests = {'Program A': 150, 'Program B': 300,'Program C': 200 }

allocations, remaining_memory = worst_fit_allocation(memory_blocks, program_requests)

for program, allocated_memory in allocations.items():
     print(f"{program} allocated {allocated_memory} units of memory.")

print("\n Remaining Memory in Memory Blocks:")
for i in range(len(memory_blocks)):
     print(f"Block {i+1}: {memory_blocks[i]} units")
```

5.2 CentralAI

In a futuristic city where advanced artificial intelligence (AI) systems govern public services, a central AI hub named CentralAI operates using a sophisticated memory management system. CentralAI oversees various subsystems, each requiring different amounts of memory to function efficiently. To optimize its memory usage, CentralAI employs the best-fit contiguous memory allocation technique. CentralAI manages a total of 8000 units of memory. Throughout the day, it receives memory requests from different subsystems in the following sequence:

SubsysA requests 1500 units of memory.

SubsysB requests 1000 units of memory.

SubsysC requests 700 units of memory.

SubsysD requests 2200 units of memory.

SubsysE requests 500 units of memory.

SubsysF requests 1200 units of memory.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Best-Fit Allocation Process: Show step-by-step how CentralAI allocates memory to each subsystem using the best-fit technique. Detail which blocks of memory are allocated to each subsystem and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the best-fit technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the best-fit technique in this scenario. Are there any disadvantages or concerns CentralAI should be aware of?

Explanation: Illustration of Best-Fit Allocation Process:

CentralAI starts with 8000 units of memory available.

SubsysA (1500 units) would be allocated the smallest block (if any is available), leaving 6500 units.

SubsysB (1000 units) would then take the smallest available block that fits (5500 units left).

SubsysC (700 units) would take the smallest available block (4800 units left).

SubsysD (2200 units) would take the smallest available block (2600 units left).

SubsysE (500 units) would take the smallest available block (2100 units left).

SubsysF (1200 units) would take the smallest available block (900 units left).

Detailed steps should show how the best-fit method minimizes wasted memory by allocating the smallest block that fits each subsystem's request.

Memory Utilization:

After all requests are processed, calculate the sum of memory allocated to each subsystem to find the total memory utilized.

Fragmentation Analysis:

Discuss potential fragmentation issues with best-fit, where smaller gaps of memory are left scattered after allocation, potentially making it difficult to allocate larger blocks that may arrive later. Mention strategies such as compaction or dynamic memory management to address fragmentation over time.

```
class MemoryBlock:
    def __init__(self, start, size):
        self.start = start
        self.size = size
        self.allocated = False
        self.process_name = None

    def is_free(self):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here
    def __str__(self):
        status = "Free" if not self.allocated else f"Allocated to {self.process_name}"
        return f"[Start: {self.start}, Size: {self.size}, Status: {status}]"
```

```
class MemoryManager:
    def __init__(self, total_memory):
        self.total_memory = total_memory
        self.memory_blocks = [MemoryBlock(0, total_memory)]

    def allocate_memory(self, process_name, size):
        # Write code here

    def print_memory_status(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(8000)

requests = [("SubsysA", 1500), ("SubsysB", 1000), ("SubsysC", 700),("SubsysD", 2200),
        ("SubsysE", 500), ("SubsysF", 1200)]

for request in requests:
    process_name, size = request
    allocated = memory_manager.allocate_memory(process_name, size)
    if allocated:
       memory_manager.print_memory_status()

memory_manager.print_total_memory_used()
```

5.3 MetroCentral

In a bustling metropolis where a central command center, MetroCentral, manages all public transportation systems using advanced computer systems, efficient memory management is critical. MetroCentral employs the first-fit contiguous memory allocation technique to handle memory requests from various subsystems that control different aspects of the transportation network. MetroCentral operates with a total of 5000 units of memory. Throughout the day, it receives memory requests from different subsystems in the following sequence:

TrafficControl requests 1200 units of memory.

RoutePlanning requests 800 units of memory.

VehicleMonitoring requests 1500 units of memory.

PassengerInformation requests 600 units of memory.

MaintenanceLogistics requests 700 units of memory.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the First-Fit Allocation Process: Show step-by-step how MetroCentral allocates memory to each subsystem using the first-fit technique. Detail which blocks of memory are allocated to each subsystem and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the first-fit technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the first-fit technique in this scenario. Are there any disadvantages or concerns MetroCentral should be aware of?

Explanation:

Illustration of First-Fit Allocation Process:

MetroCentral starts with 5000 units of memory available.

TrafficControl (1200 units) would be allocated the first available block that fits (entire memory block of 5000 units used).

RoutePlanning (800 units) would be allocated the next available block that fits (4000 units left).

VehicleMonitoring (1500 units) would be allocated the next available block that fits (2500 units left).

PassengerInformation (600 units) would be allocated the next available block that fits (1900 units left).

MaintenanceLogistics (700 units) would be allocated the next available block that fits (1200 units left).

Detailed steps should show how the first-fit method allocates memory based on the first available block that meets or exceeds the requested size.

Memory Utilization:

After all requests are processed, calculate the sum of memory allocated to each subsystem to find the total memory utilized.

Fragmentation Analysis:

Discuss potential fragmentation issues with first-fit, where smaller gaps of memory might be left unused between allocated blocks, potentially leading to inefficient memory usage over time.

Mention strategies such as compaction or dynamic memory management to address fragmentation and optimize memory utilization.

```
class MemoryBlock:
    def __init__(self, start, size):
        self.start = start
        self.size = size
        self.allocated = False
        self.process_name = None

    def is_free(self):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here
class MemoryManager:
```

```
    def __init__(self, total_memory):
        self.total_memory = total_memory
        self.memory_blocks = [MemoryBlock(0, total_memory)]

    def allocate_memory(self, process_name, size):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(5000)

requests = [
        ("TrafficControl", 1200),
        ("RoutePlanning", 800),
        ("VehicleMonitoring", 1500),
        ("PassengerInformation", 600),
        ("MaintenanceLogistics", 700)
        ]

for request in requests:
    process_name, size = request
    allocated = memory_manager.allocate_memory(process_name, size)
    if allocated:
       memory_manager.print_memory_status()

memory_manager.print_total_memory_used()
```

5.4 AI Lab

In a research institute focused on artificial intelligence (AI) advancements, the AI Lab manages its computational resources using a first-fit contiguous memory allocation technique. The AI Lab is responsible for running various experiments and simulations, each requiring different amounts of memory. The AI Lab operates with a total of 6000 units of memory. Throughout the day, it receives memory requests from different research projects in the following sequence:

Project A requests 1500 units of memory.

Project B requests 1000 units of memory.

Project C requests 700 units of memory.

Project D requests 2200 units of memory.

Project E requests 500 units of memory.

Project F requests 1200 units of memory.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the First-Fit Allocation Process: Show step-by-step how the AI Lab allocates memory to each research project using the first-fit technique. Detail which blocks of memory are allocated to each project and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the first-fit technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the first-fit technique in this scenario. Are there any disadvantages or concerns the AI Lab should be aware of?

Explanation: The AI Lab starts with 6000 units of memory available.

Project A (1500 units) would be allocated the first available block that fits (entire memory block of 6000 units used).

Project B (1000 units) would be allocated the next available block that fits (4500 units left).

Project C (700 units) would be allocated the next available block that fits (3800 units left).

Project D (2200 units) would be allocated the next available block that fits (1600 units left).

Project E (500 units) would be allocated the next available block that fits (1100 units left).

Project F (1200 units) would be allocated the next available block that fits (0 units left).

Detailed steps should show how the first-fit method allocates memory based on the first available block that meets or exceeds the requested size.

Memory Utilization:

After all requests are processed, calculate the sum of memory allocated to each project to find the total memory utilized.

Fragmentation Analysis:

Discuss potential fragmentation issues with first-fit, where smaller gaps of memory might be left unused between allocated blocks, potentially leading to inefficient memory usage over time. Mention strategies such as compaction or dynamic memory management to address fragmentation and optimize memory utilization.

```
class MemoryBlock:
    def __init__(self, start, size):
        self.start = start
        self.size = size
        self.allocated = False
        self.process_name = None

    def is_free(self):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here
    def __str__(self):
        # Write code here
```

```
class MemoryManager:
    def __init__(self, total_memory):
        # Write code here

    def allocate_memory(self, process_name, size):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(6000)

requests = [
        ("Project A", 1500),
        ("Project B", 1000),
        ("Project C", 700),
        ("Project D", 2200),
        ("Project E", 500),
        ("Project F", 1200)
          ]

for request in requests:
    process_name, size = request
    allocated = memory_manager.allocate_memory(process_name, size)
    if allocated:
       memory_manager.print_memory_status()

memory_manager.print_total_memory_used()
```

5.5 MedTech Hospital

In a bustling city where a central hospital, MedTech Hospital, manages its patient records and medical data using sophisticated computer systems, efficient memory management is crucial. MedTech Hospital employs the best-fit contiguous memory allocation technique to optimize its use of memory resources. MedTech Hospital operates with a total of 8000 units of memory. Throughout a busy day, it receives memory requests from different departments and systems in the following sequence:

Emergency Department requests 2000 units of memory.

Cardiology Department requests 1500 units of memory.

Laboratory Information System requests 1200 units of memory.

Radiology Department requests 1800 units of memory.

Patient Management System requests 1000 units of memory.

Pharmacy System requests 600 units of memory.

Surgical Services requests 2200 units of memory.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Best-Fit Allocation Process: Show step-by-step how MedTech Hospital allocates memory to each department or system using the best-fit technique. Detail which blocks of memory are allocated to each request and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the best-fit technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the best-fit technique in this scenario. Are there any disadvantages or concerns MedTech Hospital should be aware of?

Explanation: MedTech Hospital starts with 8000 units of memory available.

Emergency Department (2000 units) would be allocated the smallest available block that fits (entire memory block of 8000 units used).

Cardiology Department (1500 units) would be allocated the smallest available block that fits (6500 units left).

Laboratory Information System (1200 units) would be allocated the smallest available block that fits (5300 units left).

Radiology Department (1800 units) would be allocated the smallest available block that fits (3500 units left).

Patient Management System (1000 units) would be allocated the smallest available block that fits (2500 units left).

Pharmacy System (600 units) would be allocated the smallest available block that fits (1900 units left).

Surgical Services (2200 units) would be allocated the smallest available block that fits (0 units left).

Detailed steps should show how the best-fit method allocates memory based on the smallest available block that meets or exceeds the requested size.

Memory Utilization:

After all requests are processed, calculate the sum of memory allocated to each department or system to find the total memory utilized.

Fragmentation Analysis:

Discuss potential fragmentation issues with best-fit, where smaller gaps of memory might be left unused between allocated blocks, potentially leading to inefficient memory usage over time. Mention strategies such as compaction or dynamic memory management to address fragmentation and optimize memory utilization.

```
class MemoryBlock:
    def __init__(self, start, size):
        self.start = start
        self.size = size
        self.allocated = False
        self.process_name = None

    def is_free(self):
        # Write code here

    def allocate(self, process_name):
        # Write code here
```

```
    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here

class MemoryManager:
    def __init__(self, total_memory):
        self.total_memory = total_memory
        self.memory_blocks = [MemoryBlock(0, total_memory)]

    def allocate_memory(self, process_name, size):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(8000)

requests = [
        ("Emergency Department", 2000),
        ("Cardiology Department", 1500),
        ("Laboratory Information System", 1200),
        ("Radiology Department", 1800),
        ("Patient Management System", 1000),
        ("Pharmacy System", 600),
        ("Surgical Services", 2200)
          ]

for request in requests:
    process_name, size = request
    allocated = memory_manager.allocate_memory(process_name, size)
    if allocated:
       memory_manager.print_memory_status()

memory_manager.print_total_memory_used()
```

6. Paging Memory Management

6.1 CloudTech

In a fast-growing tech startup, CloudTech Inc., which specializes in cloud computing services, efficient memory management is critical to ensure optimal performance and resource utilization. CloudTech Inc. employs the paging technique to manage memory across its cloud servers. CloudTech Inc. operates multiple cloud servers, each equipped with a paging system. The main memory of each server is divided into fixed-size pages, and processes running on these servers request memory in terms of these pages. Each page can hold a fixed amount of data.

Each cloud server has a total of 100 pages of main memory. Each page can hold up to 4 units of data. Processes running on the servers request memory in terms of the number of pages they need. Throughout a busy day, several processes make memory requests in the following sequence:

Process A requests 25 pages.

Process B requests 15 pages.

Process C requests 30 pages.

Process D requests 12 pages.

Process E requests 20 pages.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Paging Memory Management Process: Show step-by-step how CloudTech Inc. allocates memory to each process using the paging technique. Detail which pages are allocated to each process and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the paging technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the paging technique in this scenario. Are there any disadvantages or concerns CloudTech Inc. should be aware of?

Explanation: Each cloud server starts with 100 pages of main memory.

Process A (25 pages) would be allocated consecutive pages starting from the beginning (pages 0 to 24).

Process B (15 pages) would be allocated the next available consecutive pages (pages 25 to 39).

Process C (30 pages) would be allocated consecutive pages (pages 40 to 69).

Process D (12 pages) would be allocated consecutive pages (pages 70 to 81).

Process E (20 pages) would be allocated consecutive pages (pages 82 to 101).

Detailed steps should show how the paging technique allocates memory based on consecutive pages for each process request.

Memory Utilization:

After all requests are processed, calculate the total number of pages allocated to processes to find the total memory utilized.

```
class Page:
    def __init__(self, page_id, process_name=None):
        # Write code here
```

```
    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here

class MemoryManager:
    def __init__(self, num_pages, page_size):
        self.num_pages = num_pages
        self.page_size = page_size
        self.pages = [Page(page_id) for page_id in range(num_pages)]

    def allocate_memory(self, process_name, num_pages_requested):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(num_pages=100, page_size=4)

requests = [
        ("Process A", 25),
        ("Process B", 15),
        ("Process C", 30),
        ("Process D", 12),
        ("Process E", 20)
          ]

for request in requests:
    process_name, num_pages_requested = request
    allocated = memory_manager.allocate_memory(process_name, num_pages_requested)
    if allocated:
       print(f"Allocated {num_pages_requested} pages for {process_name}")
       memory_manager.print_memory_status()
    else:
       print(f"Not enough memory available for {process_name} (requested
{num_pages_requested} pages)")

memory_manager.print_total_memory_used()
```

6.2 EduTech University

In a large educational institute, EduTech University, which handles vast amounts of student and administrative data, efficient memory management is crucial for maintaining smooth operations of its IT infrastructure. EduTech University utilizes the paging technique to manage memory across its various

departments and systems. EduTech University operates a centralized server system with a total of 200 pages of main memory. Each page can accommodate up to 8 units of data. Throughout a typical academic day, different departments and systems within the university make memory requests in the following sequence:

Student Records System requests 40 pages.

Faculty Management System requests 25 pages.

Library Information System requests 30 pages.

Online Learning Platform requests 35 pages.

Research Database requests 50 pages.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Paging Memory Management Process: Show step-by-step how EduTech University allocates memory to each department or system using the paging technique. Detail which pages are allocated to each request and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the paging technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the paging technique in this scenario. Are there any disadvantages or concerns EduTech University should be aware of?

Explanation: EduTech University starts with 200 pages of main memory.

Student Records System (40 pages) would be allocated consecutive pages starting from the beginning (pages 0 to 39).

Faculty Management System (25 pages) would be allocated the next available consecutive pages (pages 40 to 64).

Library Information System (30 pages) would be allocated consecutive pages (pages 65 to 94).

Online Learning Platform (35 pages) would be allocated consecutive pages (pages 95 to 129).

Research Database (50 pages) would be allocated consecutive pages (pages 130 to 179).

Detailed steps should show how the paging technique allocates memory based on consecutive pages for each department or system request.

Memory Utilization:

After all requests are processed, calculate the total number of pages allocated to each department or system to find the total memory utilized.

```
class Page:
    def __init__(self, page_id, process_name=None):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
```

```
        # Write code here

class MemoryManager:
    def __init__(self, num_pages, page_size):
        # Write code here

    def allocate_memory(self, process_name, num_pages_requested):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(num_pages=200, page_size=8)

requests = [
        ("Student Records System", 40),
        ("Faculty Management System", 25),
        ("Library Information System", 30),
        ("Online Learning Platform", 35),
        ("Research Database", 50)
          ]

for request in requests:
    process_name, num_pages_requested = request
    allocated = memory_manager.allocate_memory(process_name, num_pages_requested)
    if allocated:
       print(f"Allocated {num_pages_requested} pages for {process_name}")
       memory_manager.print_memory_status()
    else:
       print(f"Not enough memory available for {process_name} (requested
{num_pages_requested} pages)")
memory_manager.print_total_memory_used()
```

6.3 GameTech Studios

In a bustling gaming company, GameTech Studios, which hosts multiplayer online games, efficient memory management is crucial to ensure seamless gameplay experiences for millions of players worldwide. GameTech Studios employs the paging technique to manage memory across its gaming servers. GameTech Studios operates multiple gaming servers, each equipped with a paging system. The main memory of each server is divided into fixed-size pages, and game sessions running on these servers request memory in terms of these pages. Each page can store a fixed amount of game data.

Each gaming server has a total of 256 pages of main memory.

Each page can hold up to 16 units of game data.

Throughout a peak gaming period, several game sessions make memory requests in the following sequence:

Game Session 1 requests 32 pages.

Game Session 2 requests 20 pages.

Game Session 3 requests 40 pages.

Game Session 4 requests 18 pages.

Game Session 5 requests 25 pages.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Paging Memory Management Process: Show step-by-step how GameTech Studios allocates memory to each game session using the paging technique. Detail which pages are allocated to each request and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the paging technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the paging technique in this scenario. Are there any disadvantages or concerns GameTech Studios should be aware of?

Explanation:

Each gaming server starts with 256 pages of main memory.

Game Session 1 (32 pages) would be allocated consecutive pages starting from the beginning (pages 0 to 31).

Game Session 2 (20 pages) would be allocated the next available consecutive pages (pages 32 to 51).

Game Session 3 (40 pages) would be allocated consecutive pages (pages 52 to 91).

Game Session 4 (18 pages) would be allocated consecutive pages (pages 92 to 109).

Game Session 5 (25 pages) would be allocated consecutive pages (pages 110 to 134).

Detailed steps should show how the paging technique allocates memory based on consecutive pages for each game session request.

Memory Utilization:

After all requests are processed, calculate the total number of pages allocated to each game session to find the total memory utilized.

```
class Page:
    def __init__(self, page_id, process_name=None):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here

class MemoryManager:
```

```
    def __init__(self, num_pages, page_size):
        # Write code here

    def allocate_memory(self, process_name, num_pages_requested):
        # Write code here

    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(num_pages=256, page_size=16)

requests = [
        ("Game Session 1", 32),
        ("Game Session 2", 20),
        ("Game Session 3", 40),
        ("Game Session 4", 18),
        ("Game Session 5", 25)
           ]

for request in requests:
    process_name, num_pages_requested = request
    allocated = memory_manager.allocate_memory(process_name, num_pages_requested)
    if allocated:
       print(f"Allocated {num_pages_requested} pages for {process_name}")
       memory_manager.print_memory_status()
    else:
       print(f"Not enough memory available for {process_name} (requested
{num_pages_requested} pages)")

memory_manager.print_total_memory_used()
```

6.4 MedCare Hospital

In a large hospital, MedCare Hospital, which manages extensive patient records and medical data, efficient memory management is critical to ensure timely access to patient information and smooth operation of medical services. MedCare Hospital utilizes the paging technique to manage memory across its various departments and systems. MedCare Hospital operates a centralized IT system with a total of 300 pages of main memory. Each page can store a fixed amount of patient data. Throughout a busy day, different departments and systems within the hospital make memory requests in the following sequence:

Emergency Department requests 50 pages.

Radiology Department requests 35 pages.

Laboratory Department requests 45 pages.

Patient Records System requests 60 pages.

Surgery Department requests 40 pages.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Paging Memory Management Process: Show step-by-step how MedCare Hospital allocates memory to each department or system using the paging technique. Detail which pages are allocated to each request and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the paging technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the paging technique in this scenario. Are there any disadvantages or concerns MedCare Hospital should be aware of?

Explanation:

MedCare Hospital starts with 300 pages of main memory.

Emergency Department (50 pages) would be allocated consecutive pages starting from the beginning (pages 0 to 49).

Radiology Department (35 pages) would be allocated the next available consecutive pages (pages 50 to 84).

Laboratory Department (45 pages) would be allocated consecutive pages (pages 85 to 129).

Patient Records System (60 pages) would be allocated consecutive pages (pages 130 to 189).

Surgery Department (40 pages) would be allocated consecutive pages (pages 190 to 229).

Detailed steps should show how the paging technique allocates memory based on consecutive pages for each department or system request.

Memory Utilization:

After all requests are processed, calculate the total number of pages allocated to each department or system to find the total memory utilized.

```
class Page:
    def __init__(self, page_id, process_name=None):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here

class MemoryManager:
    def __init__(self, num_pages, page_size):
        # Write code here

    def allocate_memory(self, process_name, num_pages_requested):
        # Write code here
```

```
    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
memory_manager = MemoryManager(num_pages=300, page_size=1)

requests = [
        ("Emergency Department", 50),
        ("Radiology Department", 35),
        ("Laboratory Department", 45),
        ("Patient Records System", 60),
        ("Surgery Department", 40)
          ]

for request in requests:
    process_name, num_pages_requested = request
    allocated = memory_manager.allocate_memory(process_name, num_pages_requested)
    if allocated:
       print(f"Allocated {num_pages_requested} pages for {process_name}")
       memory_manager.print_memory_status()
    else:
       print(f"Not enough memory available for {process_name} (requested
{num_pages_requested} pages)")

memory_manager.print_total_memory_used()
```

6.5 ShopMart

In a bustling online shopping platform, ShopMart, which manages a vast inventory of products and customer transactions, efficient memory management is crucial to ensure smooth operations and timely access to product data. ShopMart employs the paging technique to manage memory across its servers. ShopMart operates multiple servers to handle customer orders and manage product information. Each server has a total of 512 pages of main memory, and each page can store a fixed amount of product data. Throughout a busy day of operations, different parts of ShopMart's system make memory requests in the following sequence:

Product Catalog Management requests 80 pages.

Order Processing System requests 120 pages.

Customer Database requests 150 pages.

Payment Processing Gateway requests 100 pages.

Inventory Tracking System requests 60 pages.

Assume that the memory is initially empty, and all requests arrive sequentially without any memory being released in between.

Illustrate the Paging Memory Management Process: Show step-by-step how ShopMart allocates memory to each system component using the paging technique. Detail which pages are allocated to each request and explain the decision-making process for each allocation.

Memory Utilization: Calculate and report the total amount of memory utilized after all requests have been processed using the paging technique.

Fragmentation Analysis: Discuss any potential fragmentation issues that might arise with the paging technique in this scenario. Are there any disadvantages or concerns ShopMart should be aware of?

Explanation:

ShopMart starts with 512 pages of main memory per server.

Product Catalog Management (80 pages) would be allocated consecutive pages starting from the beginning (pages 0 to 79).

Order Processing System (120 pages) would be allocated the next available consecutive pages (pages 80 to 199).

Customer Database (150 pages) would be allocated consecutive pages (pages 200 to 349).

Payment Processing Gateway (100 pages) would be allocated consecutive pages (pages 350 to 449).

Inventory Tracking System (60 pages) would be allocated consecutive pages (pages 450 to 509).

Detailed steps should show how the paging technique allocates memory based on consecutive pages for each system component request.

Memory Utilization:

After all requests are processed, calculate the total number of pages allocated to each system component to find the total memory utilized.

```
class Page:
    def __init__(self, page_id, process_name=None):
        # Write code here

    def allocate(self, process_name):
        # Write code here

    def deallocate(self):
        # Write code here

    def __str__(self):
        # Write code here

class MemoryManager:
    def __init__(self, num_pages, page_size):
        # Write code here

    def allocate_memory(self, process_name, num_pages_requested):
        # Write code here
    def print_memory_status(self):
        # Write code here

    def print_total_memory_used(self):
        # Write code here

# Driver Code
```

```
memory_manager = MemoryManager(num_pages=512, page_size=1)

requests = [
        ("Product Catalog Management", 80),
        ("Order Processing System", 120),
        ("Customer Database", 150),
        ("Payment Processing Gateway", 100),
        ("Inventory Tracking System", 60)
           ]

for request in requests:
    process_name, num_pages_requested = request
    allocated = memory_manager.allocate_memory(process_name, num_pages_requested)
    if allocated:
       print(f"Allocated {num_pages_requested} pages for {process_name}")
       memory_manager.print_memory_status()
    else:
       print(f"Not enough memory available for {process_name} (requested
{num_pages_requested} pages)")

memory_manager.print_total_memory_used()
```

7. Resource Allocation

7.1 UrbanOS - Resource Allocation Graph (RAG)

In a bustling city, UrbanOS, which operates a complex multitasking operating system, efficient resource management is crucial to ensure smooth operation of various processes and applications. UrbanOS employs a Resource Allocation Graph (RAG) to manage resource allocation and avoid deadlock situations. UrbanOS manages resources such as CPU time, memory, and peripherals across multiple processes running concurrently. Each process may request and release different resources dynamically throughout its execution.

Process Management: UrbanOS supports several processes including:

Process A: Handles user interface interactions and graphics rendering.

Process B: Manages database transactions and file operations.

Process C: Performs complex calculations and simulations.

Process D: Controls network communications and data transfers.

Resource Requests:

Each process in UrbanOS can request multiple resources such as CPU time, memory blocks, and access to peripherals like printers or network devices. Processes dynamically request resources based on their current tasks and release them when no longer needed.

Resource Allocation Graph (RAG):

UrbanOS maintains a Resource Allocation Graph (RAG) to track which processes are currently allocated which resources and which processes are waiting for which resources. The RAG helps UrbanOS detect potential deadlock scenarios where processes are waiting indefinitely for resources held by others.

Illustrate the Resource Allocation Graph (RAG): Draw a diagram or outline showing how UrbanOS constructs and maintains the Resource Allocation Graph (RAG) for the processes described (Process A to Process D). Detail the allocation and request edges between processes and resources.

Deadlock Prevention: Explain how UrbanOS uses the Resource Allocation Graph (RAG) to prevent deadlock situations among processes. Provide examples of deadlock scenarios and how the RAG helps identify and resolve them.

Explanation:

Process A: Requests CPU time and memory blocks for graphics rendering.

Process B: Requests CPU time and file system access for database transactions.

Process C: Requests CPU time and memory blocks for simulations.

Process D: Requests network access and CPU time for network communications.

Draw edges showing which processes are currently allocated which resources and which processes are waiting for which resources (e.g., arrows from processes to resources indicating requests).

Deadlock Prevention:

Example Scenario: Process A holds CPU time and is waiting for memory blocks held by Process C. Process C holds memory blocks and is waiting for CPU time held by Process B. Process B, in turn, is waiting for file system access held by Process D, which is waiting for network access.

RAG Utilization: UrbanOS uses cycle detection in the RAG to identify deadlock situations. If a cycle exists (e.g., Process A -> Process C -> Process B -> Process D -> Process A), UrbanOS can preemptively break one or more edges to resolve the deadlock and allow processes to proceed.

Dynamic Resource Management:

Challenges: Balancing resource utilization across multiple processes, ensuring fair access, and avoiding resource starvation.

Strategies: UrbanOS employs algorithms like Banker's algorithm or deadlock detection algorithms based on the RAG to dynamically allocate and manage resources. It ensures that processes only request resources they can feasibly utilize without causing deadlock or starvation.

```
class Process:
    def __init__(self, pid, max_resources):
        # Write code here

    def request_resource(self, resource, amount):
        # Write code here

    def release_resource(self, resource, amount):
        # Write code here

    def __str__(self):
        # Write code here

class ResourceManager:
    def __init__(self, resources):
        # Write code here
```

```
    def add_process(self, process):
        # Write code here

    def request_resource(self, process_id, resource, amount):
        # Write code here

    def release_resource(self, process_id, resource, amount):
        # Write code here

    def print_processes_status(self):
        # Write code here

# Driver Code
resources = {"CPU": 1, "Memory": 1024}

process1 = Process(1, {"CPU": 1, "Memory": 512})
process2 = Process(2, {"CPU": 1, "Memory": 768})

resource_manager = ResourceManager(resources)
resource_manager.add_process(process1)
resource_manager.add_process(process2)

print("Initial State:")
resource_manager.print_processes_status()

if resource_manager.request_resource(1, "CPU", 1):
   print("Process 1 allocated CPU")
else:
   print("Process 1 failed to allocate CPU")

if resource_manager.request_resource(1, "Memory", 512):
   print("Process 1 allocated Memory")
else:
   print("Process 1 failed to allocate Memory")

resource_manager.print_processes_status()

if resource_manager.request_resource(2, "Memory", 768):
   print("Process 2 allocated Memory")
else:
   print("Process 2 failed to allocate Memory")

resource_manager.print_processes_status()

if resource_manager.release_resource(1, "Memory", 512):
   print("Process 1 released Memory")
else:
   print("Process 1 failed to release Memory")

resource_manager.print_processes_status()
```

7.2 UrbanOS - Wait-for-Graph (WFG)

UrbanOS, a sophisticated operating system powering a smart city, relies on efficient process management and dependency tracking to ensure seamless operations across its diverse infrastructure. One of the key mechanisms UrbanOS utilizes is the Wait-for-Graph (WFG), which helps manage dependencies and prevent potential deadlock situations among concurrent processes.

Process and Task Management:

UrbanOS oversees a myriad of critical processes that operate simultaneously to support city functions:

Process A: Manages real-time traffic signal adjustments at intersections.

Process B: Handles data aggregation and analytics for environmental sensors.

Process C: Coordinates emergency response dispatch and resource allocation.

Process D: Facilitates centralized billing and utility management.

Concurrency and Dependency Dynamics:

Each process in UrbanOS may depend on resources such as CPU time, memory, network access, and access to shared databases or sensors. Processes dynamically interact, requesting resources and potentially waiting for others to release resources before proceeding.

Wait-for-Graph (WFG) Implementation:

UrbanOS constructs a Wait-for-Graph (WFG) to visually represent dependencies among processes:

Nodes in the graph represent processes. Directed edges between nodes depict dependencies where one process is waiting for another to release a resource. The WFG aids UrbanOS in detecting cyclic dependencies and managing resource contention effectively.

Visualization of Wait-for-Graph (WFG): Illustrate how UrbanOS constructs and updates the Wait-for-Graph (WFG) for the processes described (Process A to Process D). Provide a detailed diagram or outline showing process nodes, dependency edges, and current dependencies.

Deadlock Prevention Strategies: Explain how UrbanOS uses the Wait-for-Graph (WFG) to prevent deadlock situations among processes. Provide examples of potential deadlock scenarios and demonstrate how the WFG helps identify and resolve these scenarios preemptively.

Real-Time Dependency Management: Discuss the challenges UrbanOS faces in managing dependencies and resource contention among concurrent processes. How does the WFG assist in optimizing process scheduling, ensuring timely execution, and maintaining system stability?

Explanation:

Visualization of Wait-for-Graph (WFG):

Process A: Requests CPU time and access to traffic data.

Process B: Requests memory and network access for sensor data processing.

Process C: Requests CPU time and emergency service resources.

Process D: Requests database access and CPU time for utility management.

Draw directed edges in the WFG showing dependencies where one process is waiting for another to release resources, reflecting current dependencies and potential conflicts.

Deadlock Prevention Strategies:

Example Scenario: Process A is waiting for access to traffic data held by Process B. Process B is waiting for CPU time held by Process C, which is waiting for emergency service resources held by Process D, forming a cyclic dependency.

WFG Utilization: UrbanOS employs cycle detection algorithms in the WFG to identify and break potential deadlock cycles. By preemptively releasing and reallocating resources or adjusting scheduling priorities, UrbanOS resolves deadlock scenarios proactively.

Real-Time Dependency Management:

Challenges: Balancing resource demands across processes, minimizing waiting times, and ensuring fair access to critical resources.

Strategies: UrbanOS uses the WFG to monitor and optimize process dependencies in real-time. It implements dynamic scheduling algorithms based on WFG insights to prioritize critical tasks, mitigate resource contention, and enhance overall system responsiveness and reliability.

```
class Process:
    def __init__(self, pid):
        # Write code here

    def add_dependency(self, process):
        # Write code here

    def remove_dependency(self, process):
        # Write code here

    def __str__(self):
        # Write code here

class WaitForGraph:
    def __init__(self):
        self.processes = {}

    def add_process(self, process):
        # Write code here

    def add_dependency(self, process_pid, dependency_pid):
        # Write code here

    def remove_dependency(self, process_pid, dependency_pid):
        # Write code here

    def detect_deadlocks(self):
        # Write code here

        def detect_cycle(process):
            # Write code here
        return False

    def print_wfg(self):
        # Write code here
```

```
# Driver Code
process1 = Process(1)
process2 = Process(2)
process3 = Process(3)
process4 = Process(4)

wfg = WaitForGraph()

wfg.add_process(process1)
wfg.add_process(process2)
wfg.add_process(process3)
wfg.add_process(process4)

wfg.add_dependency(1, 2)
wfg.add_dependency(2, 3)
wfg.add_dependency(3, 4)
wfg.add_dependency(4, 1)

wfg.print_wfg()

if wfg.detect_deadlocks():
   print("Deadlock detected in the system.")
   wfg.remove_dependency(4, 1)
   print("Deadlock resolved.")
else:
   print("No deadlock detected.")

wfg.print_wfg()
```

7.3 The Library Conference

Imagine a conference being held at a university library, where several research teams are working on their projects. The library has a set of resources:

3 computers (R1, R2, R3)

2 projectors (P1, P2)

Each team needs access to both a computer and a projector to complete their work. The teams are as follows:

Team A: Needs 1 computer and 1 projector.

Team B: Needs 1 computer and 1 projector.

Team C: Needs 1 computer and 1 projector.

At any given time, each team is holding a resource while waiting for another resource to complete their work. The following events occur:

Team A is using computer R1 and waiting for projector P1.

Team B is using projector P2 and waiting for computer R2.

Team C is using computer R3 and waiting for projector P2.

Determine if there is a deadlock in the system. If so, describe the circular wait condition and suggest a way to prevent or resolve the deadlock.

Deadlock Detection: To detect deadlock, we need to analyze the resource allocation and the wait conditions.

Team A: Holding R1, waiting for P1.

Team B: Holding P2, waiting for R2.

Team C: Holding R3, waiting for P2.

If we construct a resource allocation graph:

Team A → P1 (waiting for P1)

Team B → R2 (waiting for R2)

Team C → P2 (waiting for P2)

We notice that:

Team A is waiting for P1, which is held by Team B.

Team B is waiting for R2, which is held by Team C.

Team C is waiting for P2, which is held by Team B.

This forms a circular wait condition: Team A → Team B → Team C → Team B

Thus, a deadlock is present.

Resolution Strategy:

Prevention: Implement policies to avoid circular wait. For example, enforce a rule where each team must request all required resources at once.

Avoidance: Use a dynamic approach to resource allocation, such as the Banker's Algorithm, to ensure resources are allocated only if it does not lead to a potential deadlock.

Detection and Recovery: If deadlock detection is used, identify the deadlocked teams and preempt resources by forcibly reallocating them to break the circular wait.

```
import threading
import time

resources = {'computers': ['R1', 'R2', 'R3'], 'projectors': ['P1', 'P2']}

team_requests = {
    'TeamA': {'computer': 'R1', 'projector': 'P1'},
    'TeamB': {'computer': 'R2', 'projector': 'P2'},
    'TeamC': {'computer': 'R3', 'projector': 'P2'}}

allocated_resources = {
    'TeamA': {'computer': None, 'projector': None},
    'TeamB': {'computer': None, 'projector': None},
    'TeamC': {'computer': None, 'projector': None}}

resource_locks = {
    'computers': {res: threading.Lock() for res in resources['computers']},
```

```
    'projectors': {res: threading.Lock() for res in resources['projectors']} }

def request_resource(team, resource_type, resource):
    # Write code here

def release_resource(team, resource_type, resource):
    # Write code here

def team_work(team):
    # Write code here

threads = []
for team in team_requests.keys():
    thread = threading.Thread(target=team_work, args=(team,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Simulation complete.")
```

7.4 The Dining Philosophers

In a dining hall, there are 5 philosophers sitting around a round table. Each philosopher needs two utensils to eat: one in the left hand and one in the right hand. The table has 5 utensils, each placed between two philosophers.

The philosophers are:

Philosopher 1: Needs utensils U1 and U2.

Philosopher 2: Needs utensils U2 and U3.

Philosopher 3: Needs utensils U3 and U4.

Philosopher 4: Needs utensils U4 and U5.

Philosopher 5: Needs utensils U5 and U1.

Each philosopher picks up the utensil on their left and then tries to pick up the utensil on their right. If they cannot pick up both utensils, they put down the one they have and wait.

Analyze the system for potential deadlock. What strategies can be used to prevent or resolve the deadlock?

Explanation:

Deadlock Detection: To check for deadlock, we observe that:

Philosopher 1 picks up U1 and waits for U2.

Philosopher 2 picks up U2 and waits for U3.

Philosopher 3 picks up U3 and waits for U4.

Philosopher 4 picks up U4 and waits for U5.

Philosopher 5 picks up U5 and waits for U1.

In this scenario, we have a circular wait:

Philosopher 1 → Philosopher 2 → Philosopher 3 → Philosopher 4 → Philosopher 5 → Philosopher 1

Thus, a deadlock is present.

Resolution Strategy:

Prevention: One common approach is to impose an order on the utensils and ensure that philosophers always pick up the lower-numbered utensil first. For example, Philosopher 1 picks up U1 and then U2, Philosopher 2 picks up U2 and then U3, and so on. This approach breaks the circular wait condition.

Avoidance: Use an algorithm like the Banker's Algorithm to ensure that no philosopher will enter a state where they wait indefinitely for resources.

Detection and Recovery: Implement an algorithm to periodically check for deadlocks. If detected, a recovery strategy such as terminating and restarting one or more philosophers can be employed.

```
import threading

class Philosopher(threading.Thread):
    def __init__(self, philosopher_id, left_utensil, right_utensil):
        # Write code here

    def run(self):
        # Write code here

    def think(self):
        # Write code here

    def eat(self):
        # Write code here

# Driver Code
num_philosophers = 5
utensils = [threading.Lock() for _ in range(num_philosophers)]
philosophers = []

for i in range(num_philosophers):
    left_utensil = utensils[i]
    right_utensil = utensils[(i + 1) % num_philosophers]
    philosopher = Philosopher(i + 1, left_utensil, right_utensil)
    philosophers.append(philosopher)

for philosopher in philosophers:
    philosopher.start()
```

7.5 Banker's Algorithm

The Banker's algorithm is a deadlock avoidance algorithm used in operating systems to manage resource allocation and prevent deadlocks. It ensures that resource requests are granted in a way that avoids the possibility of deadlocks. Let's consider a system with five processes (P1, P2, P3, P4, P5) and three resource types (R1, R2, R3). The available resources and maximum resource requirements for each process are as follows:

Available: R1(3), R2(2), R3(2) P1: R1(1), R2(2), R3(2) P2: R1(2), R2(0), R3(1) P3: R1(1), R2(1), R3(1) P4: R1(2), R2(1), R3(0) P5: R1(0), R2(0), R3(2)

To determine if the system is in a safe state using the Banker's algorithm, we need to simulate resource allocation and check if a safe sequence can be formed. Here are the steps involved:

Initialize data structures:

Available: Set it to the available resources.

Allocation: Set it to the current allocation of resources to processes.

Maximum: Set it to the maximum resource requirements of each process.

Need: Calculate it as Maximum - Allocation for each process.

Define a work array and a finish array. Initially, set all elements of the finish array to false.

Find a process P_i such that:

- Finish[i] is false.
- Need[i] <= Available.

If such a process is found, allocate its resources:

- Available = Available + Allocation[i]
- Set Finish[i] to true
- Add Pi to the safe sequence.

Repeat steps 3 and 4 until all processes are either finished or no process can be allocated resources.

If all processes are successfully allocated resources and a safe sequence is formed, then the system is in a safe state. Otherwise, if there is no safe sequence, the system is in an unsafe state, indicating a potential deadlock. In the given example, you can follow the steps of the Banker's algorithm to determine if the system is in a safe state. By simulating resource allocation and checking for a safe sequence, you can assess whether the system can avoid deadlocks.

```
def is_safe_state(processes, available, allocation, need):
    # Write code here

# Driver Code
processes = ['P1', 'P2', 'P3', 'P4', 'P5']

available = [3, 2, 2]

allocation = [
    [1, 2, 2],
    [2, 0, 1],
    [1, 1, 1],
    [2, 1, 0],
    [0, 0, 2]
]

maximum = [
    [1, 2, 2],
    [2, 0, 1],
```

```
    [1, 1, 1],
    [2, 1, 0],
    [0, 0, 2]
          ]

need = [[maximum[i][j] - allocation[i][j] for j in range(len(available))] for i in
range(len(processes))]

is_safe, safe_sequence = is_safe_state(processes, available, allocation, need)

if is_safe:
    print("The system is in a safe state.")
    print("Safe sequence:", safe_sequence)
else:
    print("The system is in an unsafe state. Deadlock may occur.")
```

8. Disk Scheduling

8.1 The Disk Access Dilemma

In a busy computer lab at Techville University, a large disk drive serves the needs of multiple students. The disk drive handles requests from several applications running on different computers. Each request involves accessing a specific track on the disk. The disk drive operates under the First-Come, First-Served (FCFS) scheduling algorithm, meaning that requests are handled in the order they are received, regardless of their location on the disk.

Imagine the disk drive has 1000 tracks, numbered 0 through 999. The disk's current head position is at track 150. Over the course of a few minutes, the following disk access requests are made by different applications:

Request 1: Track 200

Request 2: Track 50

Request 3: Track 800

Request 4: Track 300

Request 5: Track 100

The requests are received in the order listed above.

Problem:

1. Calculate the total head movement required to satisfy all the requests using the FCFS algorithm.
2. Determine the order of tracks that the disk head will visit, starting from the initial position.

Solution:

1. Calculate the Total Head Movement:
    - Initial Head Position: Track 150
    - Order of Requests:
        - Request 1: Track 200
        - Request 2: Track 50

        - Request 3: Track 800
        - Request 4: Track 300
        - Request 5: Track 100
    - Movement Calculation:
        - Move from Track 150 to Track 200: $|200-150|=50$
        - Move from Track 200 to Track 50: $|50-200|=150$
        - Move from Track 50 to Track 800: $|800-50|=750$
        - Move from Track 800 to Track 300: $|300-800|=500$
        - Move from Track 300 to Track 100: $|100-300|=200$
    - Total Head Movement:

      50 + 150 + 750 + 500 + 200 = 1650

2. Order of Tracks Visited:
    - Start at Track 150
    - Move to Track 200 (Request 1)
    - Move to Track 50 (Request 2)
    - Move to Track 800 (Request 3)
    - Move to Track 300 (Request 4)
    - Move to Track 100 (Request 5)

   Order of Tracks Visited: 150→200→50→800→300→100

```
def fcfs_disk_scheduling(initial_head, requests):
    # Write code here

# Driver Code
initial_head_position = 150
disk_requests = [200, 50, 800, 300, 100]

total_head_movement, tracks_visited = fcfs_disk_scheduling(initial_head_position,
disk_requests)

print("Total Head Movement:", total_head_movement)
print("Order of Tracks Visited:", tracks_visited)
```

8.2 The SSTF Disk Scheduling Challenge

The goal of SSTF is to reduce the total seek time by always selecting the request closest to the current head position. This approach minimizes the movement of the disk head between requests.

Process:

1. **Initial State:** Start with the initial head position (750 in this case).
2. **Select the Closest Request:** At each step, identify the request that is closest to the current head position.

3. **Move to the Closest Request:** Move the disk head to this closest request and mark it as processed.
4. **Update Position and Repeat:** Update the current head position to the new track and repeat the process until all requests are serviced.

Step-by-Step Simulation:
Initial Position: Track 750
Requests: [1200, 500, 900, 1500, 300]
Execution: Current Head: 750
Distances to Requests:
To Track 1200: |1200 - 750| = 450
To Track 500: |500 - 750| = 250
To Track 900: |900 - 750| = 150
To Track 1500: |1500 - 750| = 750
To Track 300: |300 - 750| = 450
Closest Request: Track 900 (150 units away)

Move to Track 900
Update Position: 900
Total Movement So Far: 150
Remaining Requests: [1200, 500, 1500, 300]

Current Head: 900
Distances to Requests:
To Track 1200: |1200 - 900| = 300
To Track 500: |500 - 900| = 400
To Track 1500: |1500 - 900| = 600
To Track 300: |300 - 900| = 600
Closest Request: Track 1200 (300 units away)

Move to Track 1200
Update Position: 1200
Total Movement So Far: 150 + 300 = 450
Remaining Requests: [500, 1500, 300]

Current Head: 1200
Distances to Requests:
To Track 500: |500 - 1200| = 700
To Track 1500: |1500 - 1200| = 300
To Track 300: |300 - 1200| = 900
Closest Request: Track 1500 (300 units away)

Move to Track 1500
Update Position: 1500
Total Movement So Far: 450 + 300 = 750
Remaining Requests: [500, 300]

Current Head: 1500
Distances to Requests:

To Track 500: |500 - 1500| = 1000
To Track 300: |300 - 1500| = 1200
Closest Request: Track 500 (1000 units away)

Move to Track 500
Update Position: 500
Total Movement So Far: 750 + 1000 = 1750
Remaining Request: [300]

Current Head: 500
Distance to Request: Track 300: |300 - 500| = 200
Move to Track 300
Total Movement So Far: 1750 + 200 = 1950

Summary of Results:
Order of Tracks Visited: [750, 900, 1200, 1500, 500, 300]
Total Head Movement: 1950 units

```
def sstf_disk_scheduling(initial_head, requests):
    # Write code here

# Driver Code
initial_head_position = 750
disk_requests = [1200, 500, 900, 1500, 300]

total_head_movement, tracks_visited = sstf_disk_scheduling(initial_head_position,
disk_requests)

print("Total Head Movement:", total_head_movement)
print("Order of Tracks Visited:", tracks_visited)
```

8.3 FutureTech Corporation

In the Data Center of FutureTech Corporation, a large disk drive is used to store and retrieve data for various applications. To manage disk access requests efficiently, the disk drive uses the SCAN disk scheduling algorithm. This algorithm is often referred to as the "elevator algorithm" because it works similarly to an elevator in a building: it moves in one direction, servicing requests until it reaches the end, and then reverses direction to service requests on the way back.

The disk has 5000 tracks, numbered from 0 to 4999. The disk head starts at track 2500 and is currently moving in the direction of higher-numbered tracks. Over the course of the day, the following disk access requests are made:

Request 1: Track 2800

Request 2: Track 1500

Request 3: Track 3500

Request 4: Track 4000

Request 5: Track 1000

Problem: Simulate the SCAN algorithm to determine the order in which the disk head will process the requests. Calculate the total head movement required to service all the requests using the SCAN algorithm, given that the disk head is initially moving towards higher-numbered tracks.

Direction of Movement: The disk head starts at track 2500 and is moving towards the higher-numbered tracks. Once it reaches the end of the disk (track 4999), it will reverse direction and service requests on its way back towards track 0.

Requests: Process the requests in the order the head encounters them as it moves in the initial direction and then in the reverse direction.

Head Movement Calculation: Keep track of the total distance moved by the disk head from its initial position to the final position after all requests have been serviced.

Explanation: To solve this problem, follow these steps:
Identify Requests in the Path:
Initial Direction (Higher Tracks):
Start at Track 2500.
Moving towards higher tracks, service the requests that fall within this direction until reaching the end of the disk.
Requests in the Initial Direction:
Track 2800
Track 3500
Track 4000
Reverse Direction (Lower Tracks):
After reaching the end of the disk (Track 4999), reverse direction to service the remaining requests.
Requests in the Reverse Direction:
Track 1500
Track 1000
Simulate the Movement:
Move from Track 2500 to Track 2800: $|2800-2500| = 300$
Move from Track 2800 to Track 3500: $|3500-2800|=700$
Move from Track 3500 to Track 4000: $|4000-3500|=500$
Move from Track 4000 to the End of the Disk (Track 4999): $|4999-4000|=999$
Reverse Direction and Move from Track 4999 to Track 1500: $|1500-4999|=3499$
Move from Track 1500 to Track 1000: $|1000-1500|=500$
Total Head Movement Calculation: $300+700+500+999+3499+500=5498$ units
Order of Tracks Visited:
Start at Track 2500
Move to Track 2800
Move to Track 3500
Move to Track 4000
Move to Track 4999 (end of the disk)
Reverse direction
Move to Track 1500
Move to Track 1000
Summary:
Order of Tracks Visited: [2500, 2800, 3500, 4000, 4999, 1500, 1000]
Total Head Movement: 5498 units

The SCAN algorithm ensures that each request is serviced in an orderly fashion, minimizing the potential delays and ensuring that requests are handled in a systematic manner, similar to how an elevator services floors.

```
def scan_disk_scheduling(initial_head, requests, direction='up', max_track=4999):
    # Write code here

# Driver Code
initial_head_position = 2500
disk_requests = [2800, 1500, 3500, 4000, 1000]

total_head_movement,  tracks_visited  =  scan_disk_scheduling(initial_head_position,
disk_requests, direction='up')

print("Total Head Movement:", total_head_movement)
print("Order of Tracks Visited:", tracks_visited)
```

8.4 The C-SCAN Disk Scheduling Odyssey

In the Advanced Data Center at StellarTech Enterprises, a state-of-the-art disk drive is used to handle high-priority data access requests from various applications. The disk drive uses the Circular SCAN (C-SCAN) disk scheduling algorithm to manage these requests. C-SCAN is designed to provide a more uniform wait time by only servicing requests in one direction and then quickly returning to the beginning of the disk to continue servicing. The disk in the StellarTech data center has 10,000 tracks, numbered from 0 to 9999. The disk head starts at track 4000 and is currently moving in the direction of higher-numbered tracks.

Over the day, the following disk access requests are made:

Request 1: Track 4200

Request 2: Track 1000

Request 3: Track 6000

Request 4: Track 7500

Request 5: Track 2000

Simulate the C-SCAN algorithm to determine the order in which the disk head will process the requests.

Calculate the total head movement required to service all the requests using the C-SCAN algorithm.

Direction of Movement: The disk head starts at track 4000 and is moving towards the higher-numbered tracks. Once it reaches the end of the disk (track 9999), it will quickly move back to track 0 and continue servicing requests from there.

Requests: Process the requests in the direction the head is currently moving (towards higher tracks) and then wrap around to handle requests starting from the beginning of the disk.

Head Movement Calculation: Keep track of the total distance moved by the disk head from its initial position to the final position after all requests have been serviced.

Explanation: To solve this problem, follow these steps:

Identify Requests in the Path:

Initial Direction (Higher Tracks):

Start at Track 4000.

Moving towards higher tracks, service the requests that fall within this direction until reaching the end of the disk.

Requests in the Initial Direction:

Track 4200

Track 6000

Track 7500

Wrap Around:

After reaching the end of the disk (Track 9999), wrap around to the start of the disk (Track 0) to continue servicing the remaining requests.

Requests after Wrapping Around:

Track 1000

Track 2000

Simulate the Movement:

Move from Track 4000 to Track 4200: $|4200-4000|=200$

Move from Track 4200 to Track 6000: $|6000-4200|=1800$

Move from Track 6000 to Track 7500: $|7500-6000|=1500$

Move from Track 7500 to the End of the Disk (Track 9999): $|9999-7500|=2499$

Wrap Around to Track 0: $|9999-0|=9999$

Move from Track 0 to Track 1000: $|1000-0|=1000$

Move from Track 1000 to Track 2000: $|2000-1000|=1000$

Total Head Movement Calculation: $200+1800+1500+2499+9999+1000+1000=19,998$ units

Order of Tracks Visited:

Start at Track 4000

Move to Track 4200

Move to Track 6000

Move to Track 7500

Move to Track 9999 (end of the disk)

Wrap around to Track 0

Move to Track 1000

Move to Track 2000

Order of Tracks Visited: [4000, 4200, 6000, 7500, 9999, 0, 1000, 2000]

Total Head Movement: 19,998 units

The C-SCAN algorithm provides a systematic way of servicing disk requests in one direction and then quickly wrapping around to the beginning to ensure fair treatment of all requests, even those that are far from the current head position.

```
def cscan_disk_scheduling(initial_head, requests, max_track=9999):
    # Write code here

# Driver Code
initial_head_position = 4000
disk_requests = [4200, 1000, 6000, 7500, 2000]

total_head_movement, tracks_visited = cscan_disk_scheduling(initial_head_position,
disk_requests)

print("Total Head Movement:", total_head_movement)
print("Order of Tracks Visited:", tracks_visited)
```

8.5 The C-SCAN Disk Scheduling Quest at TechFusion Labs

At TechFusion Labs, a cutting-edge research facility, disk drives are crucial for storing and managing vast amounts of scientific data. To efficiently handle disk access requests, the facility uses the Circular SCAN (C-SCAN) disk scheduling algorithm. This algorithm is chosen for its ability to provide fair access to disk resources by continuously moving in one direction and wrapping around to handle all pending requests. The disk drive at TechFusion Labs has 8,000 tracks, numbered from 0 to 7999. The disk head starts at track 3500 and is currently moving towards higher-numbered tracks.

Throughout the day, the following disk access requests are made:

Request A: Track 3800
Request B: Track 600
Request C: Track 7000
Request D: Track 1500
Request E: Track 2500

Simulate the C-SCAN algorithm to determine the order in which the disk head will process the requests. Calculate the total head movement required to service all the requests using the C-SCAN algorithm, given that the disk head starts at track 3500 and moves in the direction of higher-numbered tracks.

Direction of Movement: The disk head starts at track 3500 and is moving towards higher-numbered tracks. Once it reaches the end of the disk (track 7999), it will quickly return to track 0 and continue servicing requests from there.

Requests: Process the requests in the direction the head is currently moving (towards higher tracks) and then wrap around to handle the remaining requests starting from the beginning of the disk.

Head Movement Calculation: Keep track of the total distance moved by the disk head from its initial position to the final position after all requests have been serviced.

Explanation:

Identify Requests in the Path:

Initial Direction (Higher Tracks):

Start at Track 3500.

Moving towards higher tracks, service the requests that fall within this direction until reaching the end of the disk.

Requests in the Initial Direction:

Track 3800
Track 6000
Track 7000

Wrap Around:
After reaching the end of the disk (Track 7999), wrap around to the start of the disk (Track 0) to continue servicing the remaining requests.
Requests after Wrapping Around:
Track 600
Track 1500
Track 2500
Simulate the Movement:
Move from Track 3500 to Track 3800: |3800−3500|=300
Move from Track 3800 to Track 6000: |6000−3800|=2200
Move from Track 6000 to Track 7000: |7000−6000|=1000
Move from Track 7000 to the End of the Disk (Track 7999): |7999−7000|=999
Wrap Around to Track 0: |7999−0|=7999
Move from Track 0 to Track 600: |600−0|=600
Move from Track 600 to Track 1500: |1500−600|=900
Move from Track 1500 to Track 2500: |2500−1500|=1000
Total Head Movement Calculation: 300+2200+1000+999+7999+600+900+1000=14,998 units
Order of Tracks Visited:
Start at Track 3500
Move to Track 3800
Move to Track 6000
Move to Track 7000
Move to Track 7999 (end of the disk)
Wrap around to Track 0
Move to Track 600
Move to Track 1500
Move to Track 2500

Order of Tracks Visited: [3500, 3800, 6000, 7000, 7999, 0, 600, 1500, 2500]
Total Head Movement: 14,998 units

The C-SCAN algorithm ensures that the disk head continuously moves in one direction, wraps around to the beginning of the disk, and processes all pending requests in a fair and systematic manner. This helps in providing uniform wait times and efficient request handling.

```
def cscan_disk_scheduling(initial_head, requests, max_track=7999):
    # Write code here

initial_head_position = 3500
disk_requests = [3800, 600, 7000, 1500, 2500]

total_head_movement, tracks_visited = cscan_disk_scheduling(initial_head_position,
disk_requests)

print("Total Head Movement:", total_head_movement)
print("Order of Tracks Visited:", tracks_visited)
```

9. Concurrency Control

9.1 The Tale of the Library Management System

Imagine a large university library called UniLib, which has an automated system to manage its book inventory and handle user requests. The system must ensure that book information is consistently updated and accessible to all users, even when multiple requests are being processed simultaneously.

- **Book Checkouts:** When a user checks out a book, the system must update the book's availability status.
- **Book Returns:** When a user returns a book, the system must update the book's status to available.
- The library has the following setup:
- **Book Inventory:** A central inventory database that tracks the availability of each book.
- **User Requests:** Users can check out or return books at any time, and multiple requests can be processed concurrently.

The library system faces these challenges:

Concurrency Control: Multiple users may attempt to check out or return the same book simultaneously, leading to potential conflicts and inconsistent book availability information.

Data Integrity: Ensuring that the book's status is accurately updated and that no two users can perform conflicting operations on the same book.

Challenges:

- **Concurrency Control Mechanisms:** How should the system handle concurrent requests to prevent inconsistencies and ensure that book statuses are correctly updated?
- **Locking Strategies:** What strategies should be used to lock resources (e.g., books) during updates to avoid conflicts and ensure data integrity?
- **Deadlock Prevention:** How can the system avoid deadlocks where users are waiting indefinitely for resources to be released?

Explanation:

To address these challenges, the library system can implement concurrency control techniques and locking strategies:

Concurrency Control Mechanisms:

Use Locking: Employ exclusive locks on books to ensure that only one operation (check out or return) can modify a book's status at a time.

Locking Strategies:

Pessimistic Locking: Lock the book's record when a user starts a checkout or return operation and release the lock only after the operation is completed.

Optimistic Locking: Use version numbers or timestamps to check if the book's status has changed before updating it, rolling back if a conflict is detected.

Deadlock Prevention:

Implement a deadlock prevention strategy such as timeout-based locking where a request is aborted if it cannot acquire a lock within a specified time.

```
import threading
import time
import random

class BookInventory:
    def __init__(self):
```

```
        # Write code here

    def return_book(self, book):
        # Write code here

def user_request(inventory, book, action):
    # Write code here

# Driver Code
inventory = BookInventory()

threads = []
books = ['Book1', 'Book2', 'Book3']
actions = ['checkout', 'return']

for i in range(10):
    book = random.choice(books)
    action = random.choice(actions)
    t = threading.Thread(target=user_request, args=(inventory, book, action))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("All operations completed.")
```

9.2 The Tale of the Restaurant Reservation System

Imagine a popular restaurant named Gourmet Haven that uses an automated reservation system to manage table bookings. The restaurant has a limited number of tables, and the reservation system must handle concurrent booking requests efficiently while ensuring that no table is double-booked.

Gourmet Haven operates with:

- **Tables:** The restaurant has 10 tables, each with a unique ID.
- **Reservation System:** A central system that processes reservation requests from multiple customers.

Reservation System faces the following challenges:

- **Concurrency:** Multiple customers might attempt to book the same table at the same time.
- **Data Integrity:** The system must ensure that each table is reserved for only one customer at a time and that all reservation details are correctly updated.

Challenges:

- **Concurrency Control:** How should the system handle simultaneous booking requests to prevent double-booking of tables?
- **Locking Mechanisms:** What strategies should be employed to lock table records during reservation processing to ensure data consistency?
- **Transaction Management:** How can the system manage transactions to ensure that all operations related to a reservation (e.g., updating table status, notifying customers) are completed successfully?

Explanation:

To address these challenges, the reservation system can implement the following strategies:

Concurrency Control Mechanisms:

- **Use Optimistic Locking:** Implement versioning or timestamps to detect if a table's status has changed before finalizing a reservation.
- **Use Pessimistic Locking:** Lock the table's record while processing the reservation to prevent other requests from modifying it simultaneously.

Locking Strategies:

- **Exclusive Locks:** Lock the table record for the duration of the reservation process to prevent double-booking.

Transaction Management:

- **Atomic Transactions:** Ensure that the reservation process is atomic—either all operations are completed successfully, or none are, rolling back if any part of the transaction fails.

```
import threading
import time
import random

class ReservationSystem:
    def __init__(self, num_tables):
        # Write code here

    def reserve_table(self, table_id):
        # Write code here

    def release_table(self, table_id):
        # Write code here

def customer_request(reservation_system, table_id):
    # Write code here

# Driver Code
reservation_system = ReservationSystem(num_tables=10)

threads = []
table_ids = [f'Table{i+1}' for i in range(10)]

for _ in range(20):  # Simulate 20 customer requests
    table_id = random.choice(table_ids)
    t = threading.Thread(target=customer_request, args=(reservation_system,
table_id))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("All reservations completed.")
```

9.3 The Tale of the Online Shopping Cart System

Imagine a popular online shopping platform called ShopEase. The platform has a feature that allows customers to add items to their shopping cart and proceed to checkout. Due to high traffic, the system needs to handle multiple customers accessing and modifying their shopping carts concurrently.
The Problem: ShopEase must ensure:

- **Concurrency:** Multiple customers may attempt to add or remove the same item from their cart simultaneously.
- **Data Integrity:** The system must maintain accurate item counts and prevent inconsistencies in the shopping cart data.

ShopEase operates with:

- **Shopping Carts:** Each customer has a unique shopping cart that can be modified.
- **Inventory System:** The inventory tracks the availability of items and is updated when items are added or removed from carts.

Challenges:

- **Concurrency Control:** How should the system handle simultaneous updates to the same shopping cart or inventory item to prevent data inconsistencies?
- **Locking Mechanisms:** What locking strategies should be employed to ensure that operations on shopping carts and inventory items are synchronized and do not conflict?
- **Transaction Management:** How can the system manage transactions to ensure that all operations related to adding or removing items from the cart are completed successfully or rolled back if any part fails?

Explanation:
To address these challenges, the shopping cart system can implement the following strategies:
Concurrency Control Mechanisms:

- **Pessimistic Locking:** Lock the shopping cart and inventory items while modifications are being made to prevent other operations from conflicting.
- **Optimistic Locking:** Use versioning or timestamps to detect changes in the cart or inventory before committing updates.

Locking Strategies:

- **Exclusive Locks:** Apply exclusive locks on shopping carts and inventory items to ensure that only one modification operation can occur at a time.

Transaction Management:

- **Atomic Transactions:** Ensure that the process of updating the cart and inventory is atomic, meaning all operations must succeed together, or none should apply if there is an error.

```
import threading
import time
import random

class InventorySystem:
    def __init__(self):
        # Write code here

    def add_item(self, item, quantity):
        # Write code here

    def remove_item(self, item, quantity):
        # Write code here

class ShoppingCart:
    def __init__(self):
```

```
        # Write code here

    def add_to_cart(self, user_id, item, quantity):
        # Write code here

    def remove_from_cart(self, user_id, item, quantity):
        # Write code here

def customer_action(user_id, cart, inventory):
    # Write code here

# Driver Code
inventory = InventorySystem()
cart = ShoppingCart()

threads = []
user_ids = [f'User{i+1}' for i in range(10)]

for user_id in user_ids:
    for _ in range(5):
        t = threading.Thread(target=customer_action, args=(user_id, cart, inventory))
        threads.append(t)
        t.start()

for t in threads:
    t.join()

print("All customer actions completed.")
```

9.4 The Tale of the Collaborative Document Editing System

Imagine a collaborative document editing platform called EditTogether. This platform allows multiple users to edit the same document simultaneously. To maintain consistency and prevent conflicts, the system must manage concurrent access to the document's content effectively.

The Problem: EditTogether needs to handle:

Concurrent Edits: Multiple users might edit the same part of the document at the same time.

Data Integrity: Ensure that all changes are accurately recorded and that no edits are lost or overwritten improperly.

EditTogether operates with:

- **Documents:** Users can open and edit shared documents.
- **Editors:** Each user acts as an editor who can make changes to the document.

Challenges:

- **Concurrency Control:** How should the system handle simultaneous edits to the same section of the document to prevent data corruption and loss?
- **Locking Mechanisms:** What strategies should be employed to lock sections of the document during editing to ensure that changes are applied correctly and consistently?
- **Conflict Resolution:** How can the system detect and resolve conflicts when multiple users attempt to edit the same content concurrently?

Explanation:

To address these challenges, the document editing system can implement the following strategies:

Concurrency Control Mechanisms:

- **Optimistic Locking:** Use version numbers or timestamps to detect if the document or a section has changed since a user started editing. Apply changes only if the document is in the same state as when editing started.
- **Pessimistic Locking:** Lock sections of the document while a user is editing to prevent other users from making changes to the same section.

Locking Strategies:

- **Fine-Grained Locking:** Lock specific sections or paragraphs of the document rather than the entire document to allow multiple users to edit different parts concurrently.
- **Exclusive Locks:** Use exclusive locks on sections being edited to ensure that only one user can make changes at a time.

Conflict Resolution:

- **Change Tracking:** Track changes made by users and detect conflicts when merging changes.
- **Merge Strategies:** Implement merge algorithms to integrate concurrent changes, providing users with options to resolve conflicts manually if necessary.

```
import threading
import time
import random

class Document:
    def __init__(self):
        # Write code here

    def edit_line(self, line_number, new_text):
        # Write code here

    def show_document(self):
        # Write code here

def user_edit(document, user_id):
    # Write code here

# Driver Code
document = Document()

threads = []
user_ids = [f'User{i+1}' for i in range(10)]

for user_id in user_ids:
    t = threading.Thread(target=user_edit, args=(document, user_id))
    threads.append(t)
    t.start()

for t in threads:
```

```
    t.join()

document.show_document()

print("All user edits completed.")
```

9.5 The Tale of the Bank Account Management System

Imagine a large bank called SecureBank that handles thousands of transactions every day. The bank uses an automated system to manage customer accounts. The system must ensure that transactions are processed accurately and that account balances are correctly updated even when multiple transactions occur simultaneously.

SecureBank needs to handle:

- **Concurrent Transactions:** Multiple transactions may be processed on the same account at the same time.
- **Data Integrity:** Ensure that account balances are updated correctly and that no transactions are lost or incorrectly applied.

SecureBank operates with:

- **Accounts:** Each customer has a unique account with a balance.
- **Transactions:** Transactions include deposits and withdrawals that modify account balances.

Challenges:

- **Concurrency Control:** How should the system manage simultaneous transactions on the same account to prevent inconsistencies and ensure correct balance updates?
- **Locking Mechanisms:** What locking strategies should be used to synchronize access to account balances and prevent race conditions?
- **Transaction Management:** How can the system ensure that each transaction is processed atomically and that any failure in the transaction does not leave the account in an inconsistent state?

Explanation:

To address these challenges, the bank system can implement the following strategies:

Concurrency Control Mechanisms:

- **Pessimistic Locking:** Lock the account record while a transaction is being processed to prevent other transactions from modifying the same account concurrently.
- **Optimistic Locking:** Use versioning or timestamps to check if the account balance has changed before applying a transaction, rolling back if a conflict is detected.

Locking Strategies:

- **Exclusive Locks:** Use exclusive locks on account records during transactions to ensure that only one transaction can modify an account's balance at a time.

Transaction Management:

- **Atomic Transactions:** Ensure that each transaction is atomic, meaning that it either fully completes or is fully rolled back if any part fails.

```
import threading
import time
import random

class BankAccount:
    def __init__(self, balance):
        # Write code here

    def deposit(self, amount):
        # Write code here

    def withdraw(self, amount):
        # Write code here

def perform_transaction(account, transaction_type, amount):
    # Write code here

# Driver Code
account = BankAccount(balance=1000)

threads = []
transaction_types = ['deposit', 'withdraw']

for _ in range(20):
    transaction_type = random.choice(transaction_types)
    amount = random.randint(50, 200)
    t = threading.Thread(target=perform_transaction, args=(account, transaction_type,
amount))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
print("All transactions completed.")
```

10. Page Replacement Algorithms

10.1 The Story of a Busy Café and Its Orders

Imagine a cozy café called Café FIFO. This café is known for its efficient handling of orders, thanks to its unique ordering system. The café staff use a FIFO queue to manage their incoming orders, ensuring that the first order received is also the first one to be processed.

Let's delve into a day at Café FIFO to understand how this system works.

Characters:

- **A**, the café manager.
- **B**, the barista.
- **C**, a customer who places orders.

It’s a bustling morning at Café FIFO. The café has a small counter with a display that can only show a maximum of 4 orders at a time. When more orders come in and the counter is full, the oldest order (the one that’s been there the longest) gets processed and removed to make space for the new ones. This is how the café manages its orders efficiently.

Here's a sequence of orders placed:

Order 1 - Coffee

Order 2 - Tea

Order 3 - Muffin

Order 4 - Croissant

Order 5 - Smoothie

The café processes these orders using the FIFO method, which means:

Order 1 (Coffee) was placed first and is displayed on the counter.

Order 2 (Tea) is next on the counter.

Order 3 (Muffin) is added after Tea.

Order 4 (Croissant) comes in, filling up the counter’s capacity of 4.

Now, when Order 5 (Smoothie) arrives, the counter is full. According to FIFO, the oldest order on the counter (Order 1 - Coffee) will be processed and removed to make space for the new order.

To summarize the process:

Order 1 (Coffee) is processed and removed.

Order 5 (Smoothie) is added to the counter.

The counter now has: Order 2 (Tea), Order 3 (Muffin), Order 4 (Croissant), and Order 5 (Smoothie).

At the end of the day, the café’s order log shows the following sequence of order arrivals and processing. If you were to check the remaining orders on the counter, which ones would be there and why?

Explanation:

To determine the remaining orders, let's follow the FIFO order replacement process step-by-step:

- Order 1 (Coffee) is processed and removed when Order 5 (Smoothie) arrives.
- The remaining orders after Order 1 is processed are:
    - Order 2 (Tea)
    - Order 3 (Muffin)
    - Order 4 (Croissant)
    - Order 5 (Smoothie)

Thus, at the end of the day, the remaining orders on the counter, in FIFO order, are:

- Order 2 (Tea)
- Order 3 (Muffin)
- Order 4 (Croissant)
- Order 5 (Smoothie)

In this program, we'll maintain a queue to represent the counter in the café, which can hold a maximum of 4 orders at a time. When a new order arrives and the counter is full, the oldest order (the one that arrived first) will be processed and removed to make space for the new order.

```
from collections import deque

class CafeFIFO:
    def __init__(self, capacity):
                # Write code here

    def add_order(self, order):
                # Write code here

    def display_orders(self):
                # Write code here

# Driver Code
cafe = CafeFIFO(capacity=4)

orders = ["Coffee", "Tea", "Muffin", "Croissant", "Smoothie"]

for order in orders:
    cafe.add_order(order)

cafe.display_orders()
```

10.2 The Tale of the Library and Its Book Shelves

Imagine a small, quaint library called The Library of Timeless Tales. The library has a special section where books are displayed on a shelf with a fixed capacity. The shelf can only hold a maximum of 5 books at a time. When a new book comes in and the shelf is already full, the oldest book on the shelf is removed to make space for the new one.

The librarian, Emma, uses the FIFO policy to manage the books on the shelf. Emma's job is to ensure that the shelf is organized according to the order in which books arrive.

One day, Emma receives a sequence of book requests that need to be displayed on the shelf:

Book A - "The Great Adventure"

Book B - "Mystery of the Lost City"

Book C - "Journey Through Time"

Book D - "Secrets of the Ocean"

Book E - "The Final Frontier"

Book F - "Legends of the Hidden Realm"

Emma needs to display these books on the shelf. Since the shelf can only hold up to 5 books at a time, when a new book arrives and the shelf is full, the oldest book (the one that was on the shelf the longest) must be removed to make room for the new book.

At the end of the day, after all the books have been processed, what books will remain on the shelf?

Explanation:

To solve this problem, we will use a FIFO queue to simulate the shelf management. Here's a step-by-step breakdown:

Book A ("The Great Adventure") is placed on the shelf.

Book B ("Mystery of the Lost City") is added next.

Book C ("Journey Through Time") is added.

Book D ("Secrets of the Ocean") is added.

Book E ("The Final Frontier") is added. At this point, the shelf is full.

When Book F ("Legends of the Hidden Realm") arrives:

Book A (the oldest) will be removed from the shelf.

Book F is then added to the shelf.

After processing all the books, the remaining books on the shelf are:

Book B ("Mystery of the Lost City")

Book C ("Journey Through Time")

Book D ("Secrets of the Ocean")

Book E ("The Final Frontier")

Book F ("Legends of the Hidden Realm")

```
from collections import deque

class LibraryShelf:
    def __init__(self, capacity):
        # Write code here

    def add_book(self, book):
        # Write code here

    def display_books(self):
        # Write code here

# Driver Code
library_shelf = LibraryShelf(capacity=5)

books = [
    "The Great Adventure",
    "Mystery of the Lost City",
    "Journey Through Time",
    "Secrets of the Ocean",
    "The Final Frontier",
    "Legends of the Hidden Realm"
       ]

for book in books:
    library_shelf.add_book(book)
```

```
library_shelf.display_books()
```

10.3 The Story of the Busy Café and Its Special Recipe Book

Imagine a popular café called Café Nostalgia. The café is famous for its unique and varied menu, which changes daily. The chef, Linda, keeps a special recipe book on a counter with a limited space of 4 slots. This book contains recipes that Linda consults frequently, but the counter can only hold up to 4 recipes at a time. When Linda needs to refer to a new recipe and the counter is full, she must remove the recipe that hasn't been used for the longest time to make space for the new one. One day, Linda receives a sequence of recipe requests. She uses the LRU (Least Recently Used) policy to manage her recipe book on the counter.

Linda receives the following sequence of recipe requests for the day:

Recipe 1 - "Pumpkin Spice Latte"

Recipe 2 - "Apple Pie"

Recipe 3 - "Blueberry Muffin"

Recipe 4 - "Cinnamon Roll"

Recipe 2 - "Apple Pie"

Recipe 5 - "Maple Pancakes"

Recipe 1 - "Pumpkin Spice Latte"

Linda's job is to ensure that the recipe book reflects the most recently used recipes according to the LRU policy.

At the end of the day, after processing all the recipe requests, which recipes will remain on the counter, and in what order?

Explanation:

To solve this problem, we need to simulate the LRU page replacement algorithm. The LRU policy keeps track of the usage order of recipes, removing the least recently used recipe when the counter is full.

Here's a step-by-step breakdown of how the recipes will be managed:

Recipe 1 ("Pumpkin Spice Latte") is added to the counter.

Recipe 2 ("Apple Pie") is added.

Recipe 3 ("Blueberry Muffin") is added.

Recipe 4 ("Cinnamon Roll") is added. At this point, the counter is full.

When Recipe 2 ("Apple Pie") is requested again:

Since Recipe 2 is already on the counter and it's the most recently used, it remains on the counter.

The state of the counter remains: Recipe 1, Recipe 2, Recipe 3, Recipe 4.

When Recipe 5 ("Maple Pancakes") is requested:

The counter is full, so the least recently used recipe, Recipe 1 ("Pumpkin Spice Latte"), is removed.

Recipe 5 is added to the counter.

The state of the counter becomes: Recipe 2, Recipe 3, Recipe 4, Recipe 5.

When Recipe 1 ("Pumpkin Spice Latte") is requested again:

Recipe 1 is no longer on the counter, so it is re-added, and the least recently used recipe, Recipe 3 ("Blueberry Muffin"), is removed.

The final state of the counter becomes: Recipe 2, Recipe 4, Recipe 5, Recipe 1.

```
from collections import OrderedDict

class CafeRecipeBook:
    def __init__(self, capacity):
        # Write code here

    def add_recipe(self, recipe):
        # Write code here

    def display_recipes(self):
        # Write code here

# Driver Code
cafe_recipe_book = CafeRecipeBook(capacity=4)

recipes = [
    "Pumpkin Spice Latte",
    "Apple Pie",
    "Blueberry Muffin",
    "Cinnamon Roll",
    "Apple Pie",
    "Maple Pancakes",
    "Pumpkin Spice Latte"
         ]

for recipe in recipes:
    cafe_recipe_book.add_recipe(recipe)

cafe_recipe_book.display_recipes()
```

10.4 The Tale of the Art Gallery and Its Exhibition

Imagine a chic art gallery named Gallery of Modern Arts. The gallery features a special digital display board that can showcase up to 6 artworks at a time. The display board is updated frequently to reflect the most popular and recent artworks. Sophia, the gallery curator, manages the display using the LRU (Least Recently Used) policy. When a new artwork needs to be showcased and the board is already full, Sophia must remove the artwork that has been displayed the longest and add the new one.

Sophia receives the following sequence of artwork requests throughout the day:

Artwork 1 - "Sunset Over the Lake"

Artwork 2 - "Abstract Dreams"

Artwork 3 - "Cityscape at Dusk"

Artwork 4 - "Portrait of the Artist"

Artwork 5 - "Golden Fields"

Artwork 6 - "Mystic Mountains"

Artwork 2 - "Abstract Dreams"

Artwork 7 - "Twilight Reflections"

Artwork 1 - "Sunset Over the Lake"

Artwork 8 - "Harmony in Blue"

Sophia needs to manage the display board such that it shows the most recently used artworks according to the LRU policy. At the end of the day, after processing all the artwork requests, which artworks will remain on the display board, and in what order?

Explanation:

To solve this problem using the LRU page replacement algorithm, we'll simulate how the display board is updated. The LRU policy ensures that the least recently used artwork is removed when the board is full.

Here's a step-by-step breakdown:

Artwork 1 ("Sunset Over the Lake") is added to the display board.

Artwork 2 ("Abstract Dreams") is added.

Artwork 3 ("Cityscape at Dusk") is added.

Artwork 4 ("Portrait of the Artist") is added.

Artwork 5 ("Golden Fields") is added.

Artwork 6 ("Mystic Mountains") is added. At this point, the display board is full.

When Artwork 2 ("Abstract Dreams") is requested again:

Artwork 2 is already on the board and is moved to the most recently used position.

When Artwork 7 ("Twilight Reflections") is requested:

The board is full, so the least recently used artwork, Artwork 3 ("Cityscape at Dusk"), is removed.

Artwork 7 is added to the board.

When Artwork 1 ("Sunset Over the Lake") is requested:

Artwork 1 is already on the board and is moved to the most recently used position.

When Artwork 8 ("Harmony in Blue") is requested:

The board is full, so the least recently used artwork, Artwork 4 ("Portrait of the Artist"), is removed.

Artwork 8 is added to the board.

The final state of the display board, after processing all artwork requests, will be:

Artwork 2 ("Abstract Dreams")

Artwork 7 ("Twilight Reflections")

Artwork 1 ("Sunset Over the Lake")

Artwork 5 ("Golden Fields")

Artwork 6 ("Mystic Mountains")

Artwork 8 ("Harmony in Blue")

```
from collections import OrderedDict

class ArtGalleryDisplay:
    def __init__(self, capacity):
        # Write code here

    def add_artwork(self, artwork):
        # Write code here

    def display_artworks(self):
        # Write code here

# Driver Code
art_gallery_display = ArtGalleryDisplay(capacity=6)

artworks = [
    "Sunset Over the Lake",
    "Abstract Dreams",
    "Cityscape at Dusk",
    "Portrait of the Artist",
    "Golden Fields",
    "Mystic Mountains",
    "Abstract Dreams",
    "Twilight Reflections",
    "Sunset Over the Lake",
    "Harmony in Blue"
           ]

for artwork in artworks:
    art_gallery_display.add_artwork(artwork)

art_gallery_display.display_artworks()
```

10.5 The Tale of the Library and Its Popular Books

Imagine a charming library called The Book Haven. The library has a special digital display that showcases the top 5 most popular books at any given time. The display is updated based on how frequently each book is checked out. Liam, the library manager, uses the LFU (Least Frequently Used) policy to manage the display. If a new book needs to be featured on the display and the display is already full, Liam must remove the book that is checked out the least number of times to make room for the new one.

Throughout the day, the library receives the following sequence of book checkouts:

Book A - "The Great Gatsby" (Checked out 3 times)

Book B - "1984" (Checked out 5 times)

Book C - "To Kill a Mockingbird" (Checked out 2 times)

Book D - "Pride and Prejudice" (Checked out 4 times)

Book E - "The Catcher in the Rye" (Checked out 1 time)

Book F - "Moby Dick" (Checked out 6 times)

Book G - "The Odyssey" (Checked out 3 times)

Book H - "War and Peace" (Checked out 2 times)

Liam needs to manage the display so that it shows the most frequently checked out books according to the LFU policy. At the end of the day, after processing all the book checkouts, which books will remain on the display, and in what order?

Explanation:

To solve this problem using the LFU page replacement algorithm, we need to keep track of how frequently each book is checked out. When the display is full, the least frequently checked out book (or the one with the lowest frequency) should be removed to make space for the new book.

Here's a step-by-step breakdown of how the books will be managed:

Book A ("The Great Gatsby") is added to the display.

Book B ("1984") is added.

Book C ("To Kill a Mockingbird") is added.

Book D ("Pride and Prejudice") is added.

Book E ("The Catcher in the Rye") is added. At this point, the display is full.

When Book F ("Moby Dick") is checked out:

Book E ("The Catcher in the Rye") is the least frequently checked out book (checked out only 1 time). It is removed to make room for Book F.

Book F is added to the display.

When Book G ("The Odyssey") is checked out:

Book C ("To Kill a Mockingbird") and Book H ("War and Peace") both have a frequency of 2. However, Book C was removed earlier, so it has a higher chance of being added back.

Book G is added to the display, and Book H is added as well, replacing the least frequently checked out book among those with the same frequency.

The final state of the display, considering the frequencies of checkouts, will be:

Book B ("1984") - 5 times

Book F ("Moby Dick") - 6 times

Book D ("Pride and Prejudice") - 4 times

Book G ("The Odyssey") - 3 times

Book H ("War and Peace") - 2 times

```
from collections import defaultdict, Counter
import heapq

class LibraryDisplay:
    def __init__(self, capacity):
        # Write code here

    def add_book(self, book, times_checked_out):
        # Write code here

    def display_books(self):
        # Write code here

# Driver Code
library_display = LibraryDisplay(capacity=5)

book_checkouts = [
    ("The Great Gatsby", 3),
    ("1984", 5),
    ("To Kill a Mockingbird", 2),
    ("Pride and Prejudice", 4),
    ("The Catcher in the Rye", 1),
    ("Moby Dick", 6),
    ("The Odyssey", 3),
    ("War and Peace", 2)
]

for book, times_checked_out in book_checkouts:
    library_display.add_book(book, times_checked_out)

library_display.display_books()
```

11. Process Synchronization

11.1 The Tale of the Bakery and Its Busy Kitchen

Imagine a bustling bakery named Sweet Delights. The bakery is renowned for its delicious pastries and cakes, which are made in a busy kitchen. The kitchen has a single oven that can only bake one item at a time. The bakery operates with multiple bakers working simultaneously, and they need to use the oven to bake their items. Lily, the head baker, needs to ensure that the oven is used efficiently and fairly. She uses a process synchronization mechanism to manage access to the oven so that no two bakers use it at the

In the bakery, there are three bakers: A, B, and C. They need to bake the following items:

A wants to bake a Chocolate Cake.

B wants to bake a Fruit Tart.

C wants to bake a Cheese Croissant.

Each baker will request access to the oven, bake their item, and then release the oven for the next baker.

Challenges:

Mutual Exclusion: Only one baker should be able to use the oven at any given time.

Progress: If no baker is using the oven, and there are bakers who want to use it, then the next baker should get access to it.

Fairness: Every baker should get a chance to use the oven in the order they requested.

Explanation:

To solve this problem, Lily can use a semaphore-based synchronization mechanism to ensure that these conditions are met. Here's how she can implement this:

Initialize a semaphore to control access to the oven. The semaphore will be initialized to 1 to ensure mutual exclusion. Implement the process synchronization so that when a baker wants to use the oven, they wait (i.e., block) if the oven is already in use, and proceed only when they are granted access. Use a queue or list to keep track of the order in which bakers request access to the oven to ensure fairness.

```
import threading
import time
from queue import Queue

oven_semaphore = threading.Semaphore(1)

request_queue = Queue()

def baker(name, bake_time):
    # Write code here

# Driver Code
baker_threads = [
    threading.Thread(target=baker, args=("A", 2)),
    threading.Thread(target=baker, args=("B", 3)),
    threading.Thread(target=baker, args=("C", 1))
]

for thread in baker_threads:
    thread.start()

for thread in baker_threads:
    thread.join()

print("All bakers have finished using the oven.")
```

11.2 The Tale of the Busy Coffee Shop and Its Coffee Machines

Imagine a popular coffee shop called Brewed Bliss. The coffee shop has two coffee machines, Machine 1 and Machine 2, but only one machine can be used at a time to prepare a customer's order. The shop is busy with several baristas working simultaneously, and they need to use the machines to prepare coffee drinks for customers. John, the coffee shop manager, needs to ensure that the coffee machines are used efficiently

and fairly. He uses a process synchronization mechanism to manage access to the coffee machines so that no two baristas use the same machine at the same time, and each barista gets a chance to use the machines in an orderly manner. In the coffee shop, there are three baristas: Emma, Liam, and Olivia. Each barista has the following orders to prepare:

- Emma wants to use Machine 1 to prepare a Latte.
- Liam wants to use Machine 2 to prepare a Cappuccino.
- Olivia wants to use Machine 1 to prepare an Espresso.

Each barista will request access to the coffee machines, use them to prepare their drink, and then release the machines for the next barista.

Challenges:

Mutual Exclusion: Only one barista should be able to use a particular coffee machine at any given time.

Progress: If a coffee machine is free and there are baristas waiting to use it, then the next barista should be able to use the machine.

Fairness: Each barista should get a fair chance to use the coffee machines in the order they requested.

Explanation:

To solve this problem, John can use semaphores and a queue-based synchronization mechanism to ensure that these conditions are met. Here's a step-by-step approach:

Initialize semaphores to control access to each coffee machine. Each semaphore will be initialized to 1 to ensure mutual exclusion for each machine.

Implement a queue to manage the order in which baristas request access to the coffee machines to ensure fairness. Ensure that when a barista finishes using a machine, the next barista in the queue gets a chance to use it, maintaining fairness.

```
import threading
import time
from collections import deque

machine_1_semaphore = threading.Semaphore(1)
machine_2_semaphore = threading.Semaphore(1)

request_queue = deque()

queue_lock = threading.Lock()

def barista(name, machine, preparation_time):
    # Write code here

# Driver Code
barista_threads = [
    threading.Thread(target=barista, args=("Emma", "Machine 1", 2)),
    threading.Thread(target=barista, args=("Liam", "Machine 2", 3)),
    threading.Thread(target=barista, args=("Olivia", "Machine 1", 1))
                   ]

for thread in barista_threads:
    thread.start()
```

```
for thread in barista_threads:
    thread.join()

print("All baristas have finished using the coffee machines.")
```

11.3 The Tale of the Conference Room and Its Reservations

Imagine a company called Tech Innovations, which has a highly sought-after conference room for meetings and presentations. The conference room can be reserved by employees for various meetings, but it can only be used by one employee at a time. Sarah, the office manager, needs to manage the reservations for the conference room to ensure that no two employees are double-booked and that the reservations are handled in a fair manner. She uses a synchronization mechanism to handle access to the conference room so that no two employees can reserve the room at the same time.

In the company, there are three employees: A, B, and C. Each employee wants to reserve the conference room for a meeting at different times:

A wants to reserve the conference room from 10:00 AM to 11:00 AM.

B wants to reserve the conference room from 11:00 AM to 12:00 PM.

C wants to reserve the conference room from 12:00 PM to 1:00 PM.

Each employee will request access to the conference room, use it during their reserved time, and then release it for the next employee.

Challenges:

Mutual Exclusion: Only one employee should be able to use the conference room at any given time.

Progress: If the conference room is free and there are employees waiting to use it, then the next employee should be granted access to it.

Fairness: Each employee should get the conference room for the time they reserved, in the order of their reservation.

Explanation:

To solve this problem, Sarah can use semaphores and a queue-based synchronization mechanism to ensure that these conditions are met. Here's how it can be implemented:

- Initialize a semaphore to control access to the conference room. The semaphore will be initialized to 1 to ensure mutual exclusion.
- Implement a queue to manage the order in which employees request the conference room to ensure fairness.
- Ensure that when an employee finishes using the conference room, the next employee in the queue gets access to it.

```
import threading
import time
from collections import deque

conference_room_semaphore = threading.Semaphore(1)

request_queue = deque()
```

```
queue_lock = threading.Lock()

def employee(name, start_time, end_time):
    # Write code here


# Driver Code
employee_threads = [
    threading.Thread(target=employee, args=("A", "10:00", "11:00")),
    threading.Thread(target=employee, args=("B", "11:00", "12:00")),
    threading.Thread(target=employee, args=("C", "12:00", "13:00"))
]

for thread in employee_threads:
    thread.start()

for thread in employee_threads:
    thread.join()

print("All employees have finished using the conference room.")
```

11.4 The Tale of the Restaurant Kitchen and Its Limited Resources

Imagine a busy restaurant called Gourmet Haven. The kitchen in this restaurant is equipped with a limited number of critical resources for cooking: one stove and one refrigerator. Only one cook can use the stove at a time to prepare dishes, and only one cook can access the refrigerator at a time to retrieve ingredients. Tom, the head chef, needs to ensure that the cooks can use these resources efficiently and fairly. He uses a process synchronization mechanism to manage access to the stove and refrigerator so that no two cooks use the same resource at the same time, and each cook gets a fair chance to use the resources.

In the kitchen, there are three cooks: J, M, and R. They each need to use the stove and refrigerator for their cooking tasks:

J needs to use the stove to prepare a pasta and the refrigerator to get some cheese.

M needs to use the stove to prepare a stir-fry and the refrigerator to get some vegetables.

R needs to use the stove to prepare a soup and the refrigerator to get some herbs.

Each cook will request access to the stove and refrigerator, use them for their tasks, and then release the resources for the next cook.

Challenges:

Mutual Exclusion: Only one cook should be able to use the stove or the refrigerator at any given time.

Progress: If a resource is free and there are cooks waiting to use it, then the next cook should be granted access.

Fairness: Each cook should get access to both the stove and refrigerator in the order they requested, without causing deadlock.

Explanation:

To solve this problem, Tom can use semaphores and a queue-based synchronization mechanism to ensure that these conditions are met. Here's a step-by-step approach:

- Initialize semaphores to control access to the stove and refrigerator. Each semaphore will be initialized to 1 to ensure mutual exclusion.
- Implement a queue to manage the order in which cooks request access to the resources to ensure fairness.
- Ensure that when a cook finishes using a resource, the next cook in the queue gets access to it, maintaining fairness and preventing deadlock.

```
import threading
import time
from collections import deque

stove_semaphore = threading.Semaphore(1)
refrigerator_semaphore = threading.Semaphore(1)

request_queue = deque()

queue_lock = threading.Lock()

def cook(name, stove_time, refrigerator_time):
    # Write code here


# Driver Code
cook_threads = [
    threading.Thread(target=cook, args=("J", 2, 1)),
    threading.Thread(target=cook, args=("M", 3, 2)),
    threading.Thread(target=cook, args=("R", 1, 1))
                ]

for thread in cook_threads:
    thread.start()

for thread in cook_threads:
    thread.join()

print("All cooks have finished using the stove and refrigerator.")
```

11.5 The Tale of the Garden and Its Watering Schedule

Imagine a community garden named Green Oasis. The garden has a single water pump that can be used to water the plants. Due to the importance of keeping the plants well-watered, the garden has established strict guidelines to ensure that only one gardener uses the water pump at a time. Linda, the head gardener, needs to manage the use of the water pump so that no two gardeners use it simultaneously and that each gardener gets their turn to water their plants. She uses a process synchronization mechanism to handle access to the water pump fairly and efficiently.

In the garden, there are three gardeners: S, J, and O. They each need to use the water pump for different periods to water their assigned garden plots:

S needs to use the water pump to water her plot for 30 minutes.

J needs to use the water pump to water his plot for 45 minutes.

O needs to use the water pump to water his plot for 20 minutes.

Each gardener will request access to the water pump, use it for their designated time, and then release it for the next gardener.

Challenges:

Mutual Exclusion: Only one gardener should be able to use the water pump at any given time.

Progress: If the water pump is free and there are gardeners waiting to use it, then the next gardener should be granted access.

Fairness: Each gardener should get access to the water pump in the order they requested, without causing deadlock.

Explanation:

To solve this problem, Linda can use semaphores and a queue-based synchronization mechanism to ensure that these conditions are met. Here's a step-by-step approach:

- Initialize a semaphore to control access to the water pump. The semaphore will be initialized to 1 to ensure mutual exclusion.
- Implement a queue to manage the order in which gardeners request access to the water pump to ensure fairness.
- Ensure that when a gardener finishes using the water pump, the next gardener in the queue gets access to it.

```
import threading
import time
from collections import deque

water_pump_semaphore = threading.Semaphore(1)

request_queue = deque()

queue_lock = threading.Lock()

def gardener(name, water_time):
    # Write code here

# Driver Code
gardener_threads = [
    threading.Thread(target=gardener, args=("Sophia", 30)),
    threading.Thread(target=gardener, args=("James", 45)),
```

```
    threading.Thread(target=gardener, args=("Oliver", 20))
]

for thread in gardener_threads:
    thread.start()
for thread in gardener_threads:
    thread.join()

print("All gardeners have finished using the water pump.")
```

12. Deadlock and Prevention

12.1 The Tale of the Printing Press and Its Inks

Imagine a large printing company called Print Master, which produces high-quality magazines and brochures. The company operates two printing presses: Press 1 and Press 2. Each press needs to use multiple types of inks to print various designs. However, each press can only use one ink type at a time. E and L, two senior printers at Print Master, often need to use the presses and inks for different projects. They face a problem with their printing operations due to resource contention, leading to potential deadlocks.

- E and L both need to use the presses and inks simultaneously for their respective projects:
- E needs Press 1 and Ink A to print a brochure and Press 2 and Ink B to print a flyer.
- L needs Press 2 and Ink B to print a poster and Press 1 and Ink A to print a catalog.

Due to the constraints, there is a chance of a deadlock scenario where neither Emma nor Lucas can proceed. For instance, if Emma holds Press 1 and Ink A while waiting for Press 2 and Ink B, and Lucas holds Press 2 and Ink B while waiting for Press 1 and Ink A, both will be stuck waiting indefinitely.

Challenges:

Deadlock Prevention: Ensure that the system is designed to avoid situations where both Emma and Lucas end up waiting for each other's resources indefinitely.

Resource Allocation: Manage the allocation of presses and inks such that deadlock situations are identified and handled efficiently.

Explanation:

To handle this problem, Print Master can use strategies to prevent deadlocks, such as resource allocation strategies and introducing rules to avoid circular wait conditions.

Strategy to Avoid Deadlock:

- **Introduce a Resource Allocation Protocol:** Create a protocol that ensures resources are allocated in a way that prevents deadlock. For instance, ensuring that both Emma and Lucas must request all resources they need at once.
- **Implement Resource Request and Release Management**: Use a system to track resource requests and releases to ensure that if a resource request cannot be fulfilled, the system can rollback or preempt resources.

```
import threading
import time
```

```
press_1_lock = threading.Lock()
press_2_lock = threading.Lock()
ink_a_lock = threading.Lock()
ink_b_lock = threading.Lock()

def use_resources(printer_name, need_press_1, need_press_2, need_ink_a, need_ink_b):
    # Write code here

# Driver Code
e_thread = threading.Thread(target=use_resources, args=("E", True, True, True, True))
l_thread = threading.Thread(target=use_resources, args=("L", True, True, True, True))

e_thread.start()
l_thread.start()

e_thread.join()
l_thread.join()

print("All jobs have been completed.")
```

12.2 The Tale of the Library and Its Book Reservations

Imagine a popular public library called Book Haven. The library has a special Rare Books Room with two highly sought-after books: Book A and Book B. Due to the rarity of these books, only one person can reserve each book at a time. Additionally, each person must use a special reading desk in the Rare Books Room. S and E, two avid readers, both want to read these rare books, but their requests might lead to a deadlock situation if not managed properly.

S and E both need access to the following resources:

- **S** needs Book A and the reading desk for 2 hours.
- **E** needs Book B and the reading desk for 1.5 hours.

Due to the constraints, there is a risk of a deadlock scenario. For example:

S might hold Book A and the reading desk while waiting for Book B.

E might hold Book B and the reading desk while waiting for Book A.

If both are waiting for each other's resources, neither can proceed, leading to a deadlock.

Challenges:

- **Deadlock Prevention:** Ensure that the reservation system avoids scenarios where both Sophia and Ethan end up waiting for each other's resources indefinitely.
- **Resource Allocation:** Manage the allocation of books and desks so that the library can handle requests efficiently without falling into deadlock.

Explanation:

To handle this problem, Book Haven can implement a strategy to prevent deadlocks by ensuring that resources are requested and allocated in a way that avoids circular wait conditions.

Strategy to Avoid Deadlock:

- **Request All Resources at Once:** Ensure that a reservation request is only granted if all required resources (books and desk) are available.
- **Resource Allocation Protocol:** Use a system where a person can only request the desk if both books are available, or request both books if the desk is available.

```
import threading
import time

book_a_lock = threading.Lock()
book_b_lock = threading.Lock()
desk_lock = threading.Lock()

def reserve_resources(reader_name, need_book_a, need_book_b, need_desk,
reading_time):
    # Write code here

# Driver Code
s_thread = threading.Thread(target=reserve_resources, args=("S", True, False, True,
2))
e_thread = threading.Thread(target=reserve_resources, args=("E", False, True, True,
1.5))

sophia_thread.start()
ethan_thread.start()

sophia_thread.join()
ethan_thread.join()

print("All reservations have been completed.")
```

12.3 The Tale of the Automated Warehouse and Its Robots

In a high-tech automated warehouse called SmartStore, there are multiple robots responsible for moving items from shelves to shipping areas. The warehouse has two critical resources: Robot A and Robot B. Each robot can operate independently, but there are times when both robots need to access specific storage bins to complete their tasks. A and B, two warehouse managers, are in charge of scheduling tasks for the robots. They need to ensure that the robots are used efficiently to avoid conflicts that could lead to deadlocks.

A and B have tasks that require the following resources:

- A needs Robot A to pick up items from Bin 1 and Robot B to pick up items from Bin 2.
- B needs Robot B to pick up items from Bin 2 and Robot A to pick up items from Bin 1.

If A starts her task with Robot A and Robot B, and B starts his task with Robot B and Robot A, a deadlock situation may occur where each manager waits indefinitely for the other robot to be freed.

Challenges:

- **Deadlock Prevention:** Ensure that the system is designed to avoid situations where both Alice and Bob end up waiting for each other's robots indefinitely.

- **Resource Allocation:** Manage the allocation of robots to ensure that their tasks are completed efficiently without deadlock.

Explanation:

To handle this problem, SmartStore can use a strategy to avoid deadlocks by ensuring that requests for resources do not lead to circular waiting conditions.

Strategy to Avoid Deadlock:

- **Resource Request Ordering:** Establish a protocol that all tasks must follow in which resources are requested in a consistent order to prevent circular wait conditions.
- **Avoid Hold and Wait:** Ensure that if a task requests additional resources, it must release its currently held resources if the required resources are not available.

```
import threading
import time

robot_a_lock = threading.Lock()
robot_b_lock = threading.Lock()
bin_1_lock = threading.Lock()
bin_2_lock = threading.Lock()

def task(name, need_robot_a, need_robot_b, need_bin_1, need_bin_2, work_time):
    # Write code here

# Driver Code
a_thread = threading.Thread(target=task, args=("A", True, True, True, True, 3))
b_thread = threading.Thread(target=task, args=("B", True, True, True, True, 2))

alice_thread.start()
bob_thread.start()

a_thread.join()
b_thread.join()

print("All tasks have been completed.")
```

12.4 The Tale of the Coffee Shop and Its Baristas

Imagine a popular coffee shop named Java Junction. The shop has two crucial resources that its baristas need to prepare drinks: Espresso Machine and Blender. These resources are shared, and only one barista can use each resource at a time. A and B, the baristas, have overlapping shifts and sometimes need to use both the Espresso Machine and the Blender simultaneously to prepare complex coffee orders.
A and B each have orders that require the following resources:

- A needs the Espresso Machine to make a Latte and the Blender to make a Smoothie.
- B needs the Blender to make a Frappuccino and the Espresso Machine to make an Espresso.

Due to their overlapping schedules, if Anna starts using the Espresso Machine and Ben starts using the Blender, a deadlock situation might occur:

- A might hold the Espresso Machine and be waiting for the Blender.
- B might hold the Blender and be waiting for the Espresso Machine.

Both baristas end up waiting indefinitely for the resources held by the other, causing a deadlock.

Challenges:

- **Deadlock Prevention:** Ensure that the resource allocation system avoids scenarios where both Anna and Ben are waiting for each other's resources indefinitely.
- **Resource Allocation:** Manage the allocation of the Espresso Machine and Blender to avoid deadlock and ensure efficient operation.

Explanation:
To handle this problem, Java Junction can implement a strategy to avoid deadlocks by ensuring that resources are requested and used in a way that avoids circular dependencies.

Strategy to Avoid Deadlock:

- **Resource Ordering:** Establish a rule that resources must be requested in a specific order, ensuring that circular wait conditions are avoided.
- **Resource Allocation Protocol:** Create a protocol where a barista must request all needed resources at once and will only proceed if all are available.

```
import threading
import time

espresso_machine_lock = threading.Lock()
blender_lock = threading.Lock()

def make_drink(barista_name, need_espresso_machine, need_blender, work_time):
    # Write code here

# Driver Code
a_thread = threading.Thread(target=make_drink, args=("A", True, True, 3))
b_thread = threading.Thread(target=make_drink, args=("B", True, True, 2))

anna_thread.start()
ben_thread.start()

anna_thread.join()
ben_thread.join()

print("All drinks have been prepared.")
```

12.5 The Tale of the Theater Production and Its Props

Imagine a local theater company named Stagecraft Productions. They are preparing for a major play and have two critical props that need to be managed: Prop X and Prop Y. Each prop is essential for different scenes and can only be used by one actor at a time. E and L, two actors, are preparing for their roles. They each need to use both props to rehearse their scenes. Emma and Liam's rehearsal schedules overlap, which might lead to a potential deadlock situation.

E and L have the following prop requirements:

- **E** needs Prop X to rehearse her Scene 1 and Prop Y to rehearse her Scene 2.
- **L** needs Prop Y to rehearse his Scene 3 and Prop X to rehearse his Scene 4.

If E starts rehearsing with Prop X and L starts rehearsing with Prop Y, they may end up in a deadlock:

- **E** might hold Prop X while waiting for Prop Y.
- **L** might hold Prop Y while waiting for Prop X.

This leads to a situation where both actors are waiting indefinitely for the props held by the other.

Challenges:

- **Deadlock Prevention:** Ensure that the rehearsal system avoids scenarios where both Emma and Liam are waiting for each other's props indefinitely.
- **Resource Allocation:** Manage the allocation of props to ensure efficient rehearsal schedules without deadlock.

Explanation:

To handle this problem, Stagecraft Productions can implement strategies to avoid deadlocks by ensuring that the props are requested and used in a way that avoids circular dependencies.

Strategy to Avoid Deadlock:

- **Resource Request Protocol:** Ensure that props are requested in a specific order, and that each actor must request all required props at once.
- **Avoid Hold and Wait:** Implement a policy where if an actor cannot get all the props they need, they must release any currently held props and retry.

```
import threading
import time

prop_x_lock = threading.Lock()
prop_y_lock = threading.Lock()

def rehearse(actor_name, need_prop_x, need_prop_y, rehearsal_time):
    # Write code here

# Driver Code
e_thread = threading.Thread(target=rehearse, args=("E", True, True, 3))
l_thread = threading.Thread(target=rehearse, args=("L", True, True, 2))

e_thread.start()
l_thread.start()

e_thread.join()
l_thread.join()

print("All rehearsals have been completed.")
```

13. Queuing Theory

13.1 Customer Service - M/M/1 Queue Analysis

You are a manager at a customer service center for a popular online retailer. Customers call in to inquire about their orders, request assistance with returns, or seek product information. Your task is to analyze the customer service operations using the M/M/1 queuing model to understand and optimize service efficiency.

At your customer service center:

- Customers arrive randomly and independently at an average rate of 10 customers per hour.
- Each customer's service request, on average, takes 6 minutes to resolve.
- There is one customer service representative available to handle customer inquiries.

Objective:

Model Selection: Apply the M/M/1 queuing model to analyze the customer service operations.

Performance Metrics: Calculate and analyze key metrics such as:

Average number of customers waiting in queue.

Average time a customer spends in the system (waiting + service time).

Utilization of the customer service representative.

Optimization Recommendations: Based on the analysis, propose improvements to enhance customer service efficiency.

Tasks to Implement:

Model Formulation: Use the M/M/1 queuing model formulas to calculate steady-state metrics.

Data Collection: Gather real-time or historical data on customer arrival rates and service times.

Analysis and Recommendations: Interpret queuing model outputs to identify bottlenecks and suggest operational improvements.

Implement functions or formulas to simulate customer arrivals and service times based on provided parameters. Use the M/M/1 queuing model to predict metrics such as average number of customers in queue and average customer waiting time. Evaluate scenarios (e.g., adjusting staffing levels) to optimize service efficiency based on queuing theory insights.

```
def mm1_queue(arrival_rate, service_time):
    # Write code here

# Driver Code
arrival_rate = 10 / 60
service_time = 6

avg_customers, avg_time_in_system, utilization = mm1_queue(arrival_rate,
service_time)

print(f"M/M/1 Queue - Average Customers: {avg_customers:.2f}, Average Time in System:
{avg_time_in_system:.2f} minutes, Utilization: {utilization:.2f}")
```

13.2 Telecommunication Company - M/M/c Queue Analysis

You are a manager at a busy telecommunications company that provides technical support to customers over the phone. Customers call in for assistance with network issues, billing inquiries, and service upgrades. Your task is to analyze the efficiency of your customer support operations using the M/M/c queuing model.

At your customer support center:

- Customers arrive randomly and independently at an average rate of 15 customers per hour.
- Each customer's service request, on average, takes 8 minutes to resolve.
- Your center has 3 customer service representatives available to handle customer inquiries simultaneously.

Objective:

Model Selection: Apply the M/M/c queuing model to analyze the customer support operations.

Performance Metrics: Calculate and analyze key metrics such as:

Average number of customers waiting in queue.

Average time a customer spends in the system (waiting + service time).

Utilization of the customer service representatives.

Optimization Recommendations: Based on the analysis, propose improvements to enhance customer service efficiency.

Tasks to Implement:

Model Formulation: Use the M/M/c queuing model formulas to calculate steady-state metrics.

Data Collection: Gather real-time or historical data on customer arrival rates and service times.

Analysis and Recommendations: Interpret queuing model outputs to identify bottlenecks and suggest operational improvements.

Implement functions or formulas to simulate customer arrivals and service times based on provided parameters. Use the M/M/c queuing model to predict metrics such as average number of customers in queue and average customer waiting time. Evaluate scenarios (e.g., adjusting staffing levels) to optimize service efficiency based on queuing theory insights.

```
def mmc_queue(arrival_rate, service_time, num_servers):
    # Write code here

# Driver Code
import math

arrival_rate = 15 / 60  # customers per minute
service_time = 8  # minutes
num_servers = 3

avg_customers, avg_time_in_system, utilization = mmc_queue(arrival_rate, service_time,
num_servers)

print(f"M/M/{num_servers} Queue - Average Customers: {avg_customers:.2f}, Average Time
in System: {avg_time_in_system:.2f} minutes, Utilization: {utilization:.2f}")
```

13.3 The Tale of the Busy Call Center and Its Call Routing System

Imagine a bustling call center named QuickConnect, which handles customer support for a large telecommunications company. The call center uses a sophisticated call routing system to manage incoming customer calls. The system's goal is to ensure that each call is handled efficiently by routing it to the appropriate customer service representative (CSR).

The call center has two main departments:

- Technical Support (for troubleshooting and technical issues)
- Customer Service (for account-related queries and general assistance)

Each department has its own set of CSRs, and each CSR can handle a certain number of calls at a time. The call center uses two types of queues:

- **Incoming Call Queue:** Where calls wait until they are assigned to a CSR.
- **Department Queue:** Each department has its own queue where calls wait until they are routed to an available CSR.

Technical Support has 3 CSRs, each able to handle up to 2 calls simultaneously.

Customer Service has 2 CSRs, each able to handle up to 3 calls simultaneously.

Calls arrive at the call center according to a Poisson process with an average arrival rate of 6 calls per minute. The service times for calls follow an exponential distribution:

- Technical Support: Average service time is 5 minutes.
- Customer Service: Average service time is 3 minutes.

Challenges:

- **Queue Management:** How should the call routing system manage the queues to ensure efficient handling of incoming calls and prevent long wait times?
- **System Performance:** How does the number of available CSRs and their call-handling capacity impact the average wait time for callers in each department?
- **Balancing Load:** What strategies can be used to balance the load between the two departments to prevent overload in one while the other has a low call volume?

Explanation:

To solve these challenges, we need to analyze the queuing system using concepts from queuing theory and operating systems. We can use the following methods:

Queue Management and Analysis:

Arrival Rate (λ): 6 calls per minute.

Service Rate (μ):

Technical Support: 3 CSRs × 2 calls per CSR per minute = 6 calls per minute.

Customer Service: 2 CSRs × 3 calls per CSR per minute = 6 calls per minute.

Using the M/M/c queuing model to calculate the average wait times and system utilization for each department.

Load Balancing:

Implement a dynamic call routing system that directs calls based on current wait times and the availability of CSRs.

Adjust the number of CSRs dynamically based on the call volume to optimize performance.

```
import numpy as np
import queue
import random
import threading
import time

arrival_rate = 6
technical_service_rate = 1 / 5
customer_service_rate = 1 / 3

technical_csrs = 3
customer_service_csrs = 2

incoming_queue = queue.Queue()
tech_support_queue = queue.Queue()
customer_service_queue = queue.Queue()

def call_arrival():
    # Write code here

def tech_support_service():
    # Write code here

def customer_service():
    # Write code here

def call_routing():
    # Write code here

# Driver Code
threads = []
for _ in range(technical_csrs):
    t = threading.Thread(target=tech_support_service)
    threads.append(t)
    t.start()

for _ in range(customer_service_csrs):
    t = threading.Thread(target=customer_service)
    threads.append(t)
    t.start()

t = threading.Thread(target=call_arrival)
threads.append(t)
t.start()

t = threading.Thread(target=call_routing)
threads.append(t)
t.start()

for t in threads:
```

```
    t.join()
```

13.4 The Tale of the Hospital Emergency Room (ER)

Imagine a busy hospital's Emergency Room (ER) named HealthFirst ER. The ER is designed to handle urgent medical cases and has several key components and constraints that affect how patients are treated.

The ER has two types of medical staff:

- **Doctors:** Who provide primary medical treatment?
- **Nurses:** Who assist with initial assessments and patient care?

The ER has the following setup:

- **Doctors:** There are 4 doctors, each of whom can attend to one patient at a time. The average time a doctor spends with a patient is 20 minutes.
- **Nurses:** There are 6 nurses, each of whom can handle one patient at a time. The average time a nurse spends with a patient is 10 minutes.

Patients arrive at the ER following a Poisson process with an average arrival rate of 8 patients per hour. Upon arrival, patients first see a nurse for initial assessment and then see a doctor for treatment.

The ER operates under these constraints:

- **Nurse Queue:** Patients wait here if no nurses are available.
- **Doctor Queue:** After the initial assessment, patients wait here if no doctors are available.

Challenges:

Queue Management: How should the ER manage the queues to ensure efficient handling of patients and minimize their wait times?

System Performance: How do the number of available doctors and nurses, and their patient-handling capacity, impact the average wait time for patients in the ER?

Load Balancing: How can the ER manage the flow of patients to ensure that neither the nurse nor the doctor queue becomes a bottleneck?

Explanation:

To address these challenges, we need to use concepts from queuing theory and operating systems:

Queue Management and Analysis:

Arrival Rate (λ): 8 patients per hour.

Service Rates (μ):

Nurses: 6 nurses × 1 patient per 10 minutes = 6 patients every 10 minutes or 36 patients per hour.

Doctors: 4 doctors × 1 patient per 20 minutes = 4 patients every 20 minutes or 12 patients per hour.

System Performance Analysis:

Use the M/M/c queuing model for both the nurse and doctor queues to calculate average wait times, system utilization, and other performance metrics.

Load Balancing:

Implement strategies to dynamically manage patient flow, such as adjusting the number of nurses or doctors based on patient arrival rates or queue lengths.

```
import numpy as np
```

```
import queue
import threading
import time

arrival_rate = 8  # patients per hour
nurse_service_rate = 1 / 10  # patients per minute
doctor_service_rate = 1 / 20  # patients per minute

nurse_count = 6
doctor_count = 4

nurse_queue = queue.Queue()
doctor_queue = queue.Queue()

def patient_arrival():
    # Write code here

def nurse_service():
    # Write code here

def doctor_service():
    # Write code here

# Driver Code
threads = []
for _ in range(nurse_count):
    t = threading.Thread(target=nurse_service)
    threads.append(t)
    t.start()

for _ in range(doctor_count):
    t = threading.Thread(target=doctor_service)
    threads.append(t)
    t.start()

t = threading.Thread(target=patient_arrival)
threads.append(t)
t.start()

for t in threads:
    t.join()
```

13.5 The Tale of the Online Gaming Server

Imagine a popular online multiplayer game named GameWorld. The game's server handles various player requests and needs to manage the load efficiently to ensure a smooth gaming experience. The server has two main tasks:

- **Matchmaking**: Pairing players together for matches.

- **Game State Management:** Keeping track of the state of ongoing games and ensuring that all players are synchronized.

GameWorld has the following setup:

Matchmaking System: The system pairs players based on their skill levels and game preferences. It has 3 matchmaking servers, each capable of handling up to 50 player requests per minute. The average time to match players is 1 minute.

Game State Management System: Once players are matched, their game states are managed by 2 game state servers. Each server can handle up to 30 game state updates per minute. The average time to update a game state is 2 minutes.

Players connect to the server and request matchmaking based on a Poisson process with an average arrival rate of 120 requests per minute. Once matched, these players generate game state updates also following a Poisson process with an average rate of 100 updates per minute.

Challenges:

- **Queue Management:** How should the server manage the matchmaking and game state update queues to ensure minimal wait times and efficient processing?
- **System Performance:** How do the number of matchmaking and game state servers and their capacities affect the average wait times and system utilization?
- **Load Balancing:** What strategies can be used to balance the load between matchmaking and game state management to avoid bottlenecks?

Explanation:

To address these challenges, use queuing theory and operating systems principles:

Queue Management and Analysis:

Arrival Rates (λ):

Matchmaking Requests: 120 requests per minute.

Game State Updates: 100 updates per minute.

Service Rates (μ):

Matchmaking Servers: 3 servers × 50 requests per server per minute = 150 requests per minute.

Game State Servers: 2 servers × 30 updates per server per minute = 60 updates per minute.

System Performance Analysis:

Use the M/M/c queuing model to calculate the average wait times, system utilization, and other performance metrics for both matchmaking and game state management.

Load Balancing:

Implement strategies such as dynamic load distribution and scaling resources based on current demand to balance the load effectively.

```
import numpy as np
import queue
import threading
import time

matchmaking_arrival_rate = 120
```

```
game_state_arrival_rate = 100
matchmaking_service_rate = 1 / 1
game_state_service_rate = 1 / 2

matchmaking_servers = 3
game_state_servers = 2

matchmaking_queue = queue.Queue()
game_state_queue = queue.Queue()

def player_request():
    # Write code here

def matchmaking_service():
    # Write code here

def game_state_service():
    # Write code here


# Driver Code
threads = []
for _ in range(matchmaking_servers):
    t = threading.Thread(target=matchmaking_service)
    threads.append(t)
    t.start()

for _ in range(game_state_servers):
    t = threading.Thread(target=game_state_service)
    threads.append(t)
    t.start()

t = threading.Thread(target=player_request)
threads.append(t)
t.start()

for t in threads:
    t.join()
```