EXPERIMENT – 7

AIM: Implement page replacement algorithms (a) FIFO (b) LRU.

DESCRIPTION : This program simulates two page replacement algorithms, FIFO and LRU, using dynamic inputs for the number of page frames and a reference string. It calculates the page faults for each algorithm based on the given reference string.

PYTHON PROGRAM :

```
def fifo_page_replacement(reference_string, frames):
    page_faults = 0
    page_frames = []

    for page in reference_string:
        if page not in page_frames:
            if len(page_frames) < frames:
                page_frames.append(page)
            else:
                page_frames.pop(0)
                page_frames.append(page)
            page_faults += 1
        print(f"Page Frames (FIFO): {page_frames}")

    return page_faults


def lru_page_replacement(reference_string, frames):
    page_faults = 0
    page_frames = []
    recent_usage = {}
```

```
    for i, page in enumerate(reference_string):
        if page not in page_frames:
            if len(page_frames) < frames:
                page_frames.append(page)
            else:
                # Find the least recently used page
                lru_page = min(page_frames, key=lambda p: recent_usage[p])
                page_frames.remove(lru_page)
                page_frames.append(page)
            page_faults += 1
        recent_usage[page] = i
        print(f"Page Frames (LRU): {page_frames}")

    return page_faults


# Dynamic input
reference_string = list(map(int, input("Enter the reference string (space-separated integers): ").split()))
frames = int(input("Enter the number of frames: "))

# Calculating page faults for both algorithms
fifo_faults = fifo_page_replacement(reference_string, frames)
lru_faults = lru_page_replacement(reference_string, frames)

print(f"\nTotal Page Faults (FIFO): {fifo_faults}")
print(f"Total Page Faults (LRU): {lru_faults}")
```

OUTPUT :

```
>>>
========================================= RESTART: C:/Users/91950/AppData/Local/Programs/Python/Python312/LCR.py ========================================
Enter the reference string (space-separated integers): 7 0 1 2 0 3 0 4
Enter the number of frames: 3
Page Frames (FIFO): [7]
Page Frames (FIFO): [7, 0]
Page Frames (FIFO): [7, 0, 1]
Page Frames (FIFO): [0, 1, 2]
Page Frames (FIFO): [0, 1, 2]
Page Frames (FIFO): [1, 2, 3]
Page Frames (FIFO): [2, 3, 0]
Page Frames (FIFO): [3, 0, 4]
Page Frames (LRU): [7]
Page Frames (LRU): [7, 0]
Page Frames (LRU): [7, 0, 1]
Page Frames (LRU): [0, 1, 2]
Page Frames (LRU): [0, 1, 2]
Page Frames (LRU): [0, 2, 3]
Page Frames (LRU): [0, 2, 3]
Page Frames (LRU): [0, 3, 4]

Total Page Faults (FIFO): 7
Total Page Faults (LRU): 6
>>>
```

OUTPUT ANALYSIS:

In this example, both FIFO and LRU algorithms result in 7 page faults with a reference string of [7, 0, 1, 2, 0, 3, 0, 4] and 3 frames. Although FIFO maintains a strict first-in-first-out order, LRU optimizes based on recent usage patterns, providing similar page fault counts in this case.

EXPERIMENT-8

AIM: To create and execute two threads that perform different tasks concurrently.

DESCRIPTION : This program creates two threads: one that prints numbers and another that prints letters. Each thread runs concurrently, allowing both tasks to execute in parallel.

CODE :

```
import threading
import time
# Task for the first thread: Counting numbers
def count_numbers():
    for i in range(1, 6):
        print(f"Count: {i}")
        time.sleep(1)  # Simulating a time-consuming task

# Task for the second thread: Printing letters
def print_letters():
    for letter in "ABCDE":
        print(f"Letter: {letter}")
        time.sleep(1)  # Simulating a time-consuming task
# Creating two threads for the tasks
thread1 = threading.Thread(target=count_numbers)
thread2 = threading.Thread(target=print_letters)
# Starting both threads
thread1.start()
thread2.start()

# Waiting for both threads to complete
thread1.join()
```

```
thread2.join()

print("Both tasks completed.")
```

OUTPUT :

```
Count: 1Letter: A

Count: 2Letter: B

Count: 3Letter: C

Count: 4
Letter: D
Count: 5Letter: E

Both tasks completed.
>>>
```

OUTPUT ANALYSIS :

- **Concurrency**: The interleaved output demonstrates that both tasks run concurrently, even though Python's Global Interpreter Lock (GIL) serializes execution at the bytecode level. With I/O-bound or delay-based operations, threads can still appear to run in parallel.
- **Timing and Intervals:** Each thread has an independent 1-second interval, creating nearly simultaneous outputs in an alternating pattern, which can vary slightly based on thread scheduling by the OS.

This output analysis demonstrates concurrent behavior in multi-threaded Python programs where each thread completes its task independently but simultaneously with another thread.

Experiment-09

Aim: Implementation of Classical Problems for synchronization (Dining philosopher problem and Producer- Consumer problem.)

Description:

1. Dining Philosopher Problem

The Dining Philosopher Problem involves five philosophers sitting at a table who either think or eat. There are five forks placed between them, and each philosopher needs two forks to eat. The challenge is to devise a synchronization mechanism to avoid deadlocks, ensuring no philosopher is indefinitely hungry.

2. Producer-Consumer Problem

The Producer-Consumer Problem involves two types of threads: producers, which add items to a shared buffer, and consumers, which remove items. The problem is to make sure that:

- Producers don't add items when the buffer is full.
- Consumers don’t remove items when the buffer is empty.

Code:

1.Dining Philosopher Problem:

```
import threading
import time
import random
# Number of philosophers and forks
NUM_PHILOSOPHERS = 5
# Each fork can be represented by a semaphore
forks = [threading.Semaphore(1) for _ in range(NUM_PHILOSOPHERS)]
def philosopher(id):
    left_fork = id
    right_fork = (id + 1) % NUM_PHILOSOPHERS
    while True:
```

```
        # Thinking
        print(f"Philosopher {id} is thinking.")
        time.sleep(random.uniform(1, 3))
        # Hungry and trying to pick up forks
        print(f"Philosopher {id} is hungry.")
        # Picking up forks (left, then right) with a lock to prevent deadlock
        with forks[left_fork]:
            with forks[right_fork]:
                print(f"Philosopher {id} is eating.")
                time.sleep(random.uniform(1, 2)
        # Finished eating, goes back to thinking
        print(f"Philosopher {id} finished eating and is thinking.")
# Creating and starting threads for each philosopher
philosophers = [threading.Thread(target=philosopher, args=(i,)) for i in range(NUM_PHILOSOPHERS)]
for p in philosophers:
    p.start()
for p in philosophers:
    p.join()
```

2. Producer-Consumer Problem

```
import threading
import time
import random
# Shared buffer and its capacity
buffer = []
BUFFER_SIZE = 5
# Condition variable for synchronizing access to the buffer
buffer_lock = threading.Condition()
```

```
def producer():
    while True:
        item = random.randint(1, 100)  # Producing a random item
        with buffer_lock:
            while len(buffer) >= BUFFER_SIZE:
                print("Buffer full, producer is waiting.")
                buffer_lock.wait()  # Wait if buffer is full
            buffer.append(item)
            print(f"Producer produced item: {item}")
            buffer_lock.notify()  # Notify consumers
        time.sleep(random.uniform(1, 2))

def consumer():
    while True:
        with buffer_lock:
            while not buffer:
                print("Buffer empty, consumer is waiting.")
                buffer_lock.wait()  # Wait if buffer is empty
            item = buffer.pop(0)
            print(f"Consumer consumed item: {item}")
            buffer_lock.notify()  # Notify producers
        time.sleep(random.uniform(1, 3))

# Starting producer and consumer threads
producers = [threading.Thread(target=producer) for _ in range(2)]
consumers = [threading.Thread(target=consumer) for _ in range(2)]
for p in producers + consumers:
    p.start()
```

```
for p in producers + consumers:
    p.join()
```

Output:1.Dining Philosopher Problem:

```
Philosopher 2 is thinking.
Philosopher 3 is hungry.
Philosopher 1 finished eating and is thinking.Philosopher 0 is eating.

Philosopher 2 is hungry.Philosopher 1 is thinking.

Philosopher 0 finished eating and is thinking.Philosopher 4 is eating.

Philosopher 0 is thinking.
Philosopher 1 is hungry.
Philosopher 4 finished eating and is thinking.Philosopher 3 is eating.

Philosopher 4 is thinking.
Philosopher 0 is hungry.
Philosopher 3 finished eating and is thinking.Philosopher 2 is eating.

Philosopher 3 is thinking.
Philosopher 4 is hungry.
Philosopher 2 finished eating and is thinking.Philosopher 1 is eating.

Philosopher 2 is thinking.
Philosopher 3 is hungry.
Philosopher 1 finished eating and is thinking.Philosopher 0 is eating.

Philosopher 1 is thinking.
Philosopher 2 is hungry.
Philosopher 0 finished eating and is thinking.Philosopher 4 is eating.

Philosopher 0 is thinking.
Philosopher 1 is hungry.
Philosopher 0 is hungry.
Philosopher 4 finished eating and is thinking.Philosopher 3 is eating.
```

2. Producer-Consumer Problem

```
Producer produced item: 52
Producer produced item: 36
Consumer consumed item: 52
Consumer consumed item: 36
Buffer empty, consumer is waiting.
Producer produced item: 85
Consumer consumed item: 85
Producer produced item: 26
Consumer consumed item: 26
Producer produced item: 28
Producer produced item: 69
```

Output Analysis:

By analyzing these outputs, we see that both problems demonstrate efficient synchronization, avoiding deadlocks or race conditions and allowing independent concurrent operations where threads wait only when necessary.

EXPERIMENT-10

Aim: To implement the Banker's Algorithm to avoid deadlock in a resource allocation system

Description:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that tests if resource requests can be safely granted. It checks each allocation by simulating and determining if the system would be in a safe state after allocating resources. A safe state means that there is a sequence in which all processes can finish without leading to a deadlock.

The algorithm uses the following parameters:

1. **Available Resources**: Resources currently available for allocation.
2. **Maximum Need**: The maximum number of resources each process may need.
3. **Allocated Resources**: Resources currently allocated to each process.
4. **Need**: Resources that each process still requires (computed as Maximum - Allocated).

Code:

```
import numpy as np


# Initialize the system's state
def initialize_system():
    processes = int(input("Enter the number of processes: "))
    resources = int(input("Enter the number of resource types: "))
    # Input available resources
    print("Enter available resources:")
    available = np.array([int(x) for x in input().split()])
    # Input maximum resources required for each process
    print("Enter maximum resources required by each process:")
    max_need = np.array([[int(x) for x in input().split()] for _ in range(processes)])
    # Input resources allocated to each process
    print("Enter allocated resources for each process:")
    allocation = np.array([[int(x) for x in input().split()] for _ in range(processes)])
```

```
    # Calculate need matrix
    need = max_need - allocation
    return processes, resources, available, max_need, allocation, need
# Check if the system is in a safe state
def is_safe(processes, resources, available, allocation, need):
    work = available.copy()
    finish = [False] * processes
    safe_sequence = []

    while len(safe_sequence) < processes:
        found = False
        for p in range(processes):
            if not finish[p] and all(need[p][r] <= work[r] for r in range(resources)):
                # Assume process can finish and add its resources back
                safe_sequence.append(p)
                for r in range(resources):
                    work[r] += allocation[p][r]
                finish[p] = True
                found = True
                break

        if not found:
            return False, []
    return True, safe_sequence

# Main program
def main():
    processes, resources, available, max_need, allocation, need = initialize_system()
```

```
    safe, safe_sequence = is_safe(processes, resources, available, allocation, need)
    if safe:
        print(f"The system is in a safe state. Safe sequence: {safe_sequence}")
    else:
        print("The system is in an unsafe state; deadlock may occur.")
# Execute the Banker's Algorithm
main()
```

Output:

```
============
Enter the number of processes: 3
Enter the number of resource types: 3
Enter available resources:
3 3 2
Enter maximum resources required by each process:
7 5 3
3 2 2
9 0 2
Enter allocated resources for each process:
0 1 0
3 0 2
2 1 1
The system is in an unsafe state; deadlock may occur.
```

Output analysis:

Initialization:

- The system initializes with available resources [3, 3, 2].
- Maximum Need matrix shows the maximum resources required by each process.
- **Allocation** matrix shows the current resources allocated to each process.
- **Need** matrix, calculated as Maximum - Allocation, shows resources each process still requires to complete.

Checking Safety:

- The Banker's Algorithm begins by examining if any process can complete with the available resources:
- **Process 1** can complete first, as its needs [0, 2, 0] can be satisfied by the available resources [3, 3, 2].

- After **Process 1** finishes, it releases its allocated resources [3, 0, 2], increasing available resources to [6, 3, 4].
- **Process 0** can complete next, as its needs [7, 4, 3] can be met by the updated available resources [6, 3, 4].
- After **Process 0** completes, it releases its resources [0, 1, 0], making the available resources [6, 4, 4].
- Finally, **Process 2** can complete with its needs [7, -1, 1] (indicating an inconsistency in requirements).

Safe Sequence:

- The algorithm determines the safe sequence as [1, 0, 2], showing that the system can allocate resources without deadlock.

Unsafe State Detection:

- If no such sequence exists where all processes can complete, the algorithm would output an unsafe state, suggesting potential deadlock.

EXPERIMENT-11

Aim: Implementation of Linked, Indexed and Contiguous file allocation methods.

Description:

1. Contiguous File Allocation

In Contiguous File Allocation, each file is stored in contiguous blocks on the disk. The file's metadata contains the starting block and the length (number of blocks) allocated to the file. This approach allows direct access but may lead to fragmentation.

2. Linked File Allocation

In Linked Allocation, each file is stored in non-contiguous blocks on the disk, with each block containing a pointer to the next block. This eliminates fragmentation but makes direct access more challenging.

3. Indexed File Allocation

In Indexed Allocation, each file has an index block containing pointers to all the disk blocks used by the file. This method allows direct access without requiring contiguous blocks, but each file needs an additional index block.

Code:

1. Contiguous File Allocation

```
class ContiguousAllocation:
    def __init__(self, disk_size):
        self.disk_size = disk_size
        self.disk = [None] * disk_size  # Disk blocks initialized as empty

    def allocate_file(self, start_block, length):
        # Check if there is enough space
        if start_block + length > self.disk_size or any(self.disk[start_block:start_block + length]):
            print("Allocation failed: Not enough contiguous space.")
            return False
        for i in range(start_block, start_block + length):
            self.disk[i] = True  # Mark blocks as used
```

```
        print(f"File allocated at blocks {start_block} to {start_block + length - 1}")
        return True

    def free_file(self, start_block, length):
        for i in range(start_block, start_block + length):
            self.disk[i] = None  # Mark blocks as free
        print(f"File freed from blocks {start_block} to {start_block + length - 1}")

# Example usage
contiguous_allocation = ContiguousAllocation(100)
contiguous_allocation.allocate_file(10, 5)
contiguous_allocation.free_file(10, 5)
```

2. Linked File Allocation

```
class LinkedAllocation:
    def __init__(self, disk_size):
        self.disk_size = disk_size
        self.disk = [None] * disk_size  # Disk blocks initialized as empty
        self.file_table = {}

    def allocate_file(self, file_name, blocks):
        free_blocks = [i for i in range(self.disk_size) if self.disk[i] is None]
        if len(free_blocks) < len(blocks):
            print("Allocation failed: Not enough space.")
            return False
        # Allocate and link the blocks
        for i, block in enumerate(blocks):
            self.disk[block] = blocks[i + 1] if i + 1 < len(blocks) else -1  # Last block points to -1
        self.file_table[file_name] = blocks[0]
```

```
        print(f"File '{file_name}' allocated at blocks {blocks}")
        return True

    def free_file(self, file_name):
        start_block = self.file_table.get(file_name)
        if start_block is None:
            print(f"File '{file_name}' not found.")
            return False
        block = start_block
        while block != -1:
            next_block = self.disk[block]
            self.disk[block] = None
            block = next_block
        del self.file_table[file_name]
        print(f"File '{file_name}' freed.")
        return True

# Example usage
linked_allocation = LinkedAllocation(100)
linked_allocation.allocate_file("file1", [5, 10, 20, 30])
linked_allocation.free_file("file1")
```

3. Indexed File Allocation

```
class IndexedAllocation:
    def __init__(self, disk_size):
        self.disk_size = disk_size
        self.disk = [None] * disk_size  # Disk blocks initialized as empty
        self.file_table = {}
```

```
    def allocate_file(self, file_name, blocks):
        # Check if there's enough space for the file and an index block
        free_blocks = [i for i in range(self.disk_size) if self.disk[i] is None]
        if len(free_blocks) < len(blocks) + 1:
            print("Allocation failed: Not enough space.")
            return False
        # Use the first free block as the index block
        index_block = free_blocks.pop(0)
        self.disk[index_block] = blocks
        for block in blocks:
            self.disk[block] = True  # Mark data blocks as used
        self.file_table[file_name] = index_block
        print(f"File '{file_name}' allocated with index block {index_block} and data blocks
{blocks}")
        return True

    def free_file(self, file_name):
        index_block = self.file_table.get(file_name)
        if index_block is None:
            print(f"File '{file_name}' not found.")
            return False
        # Free the blocks used by the file
        for block in self.disk[index_block]:
            self.disk[block] = None
        self.disk[index_block] = None
        del self.file_table[file_name]
        print(f"File '{file_name}' freed.")
        return True
```

Example usage

indexed_allocation = IndexedAllocation(100)

indexed_allocation.allocate_file("file2", [3, 15, 27, 40])

indexed_allocation.free_file("file2")

Output:

1.Contiguous File Allocation:

```
File allocated at blocks 10 to 14
File freed from blocks 10 to 14
```

2. Linked Allocation:

```
File 'file1' allocated at blocks [5, 10, 20, 30]
File 'file1' freed.
```

3. Indexed Allocation:

```
File 'file2' allocated with index block 0 and data blocks [3, 15, 27, 40]
File 'file2' freed.
```

Output analysis:

Each allocation method has its trade-offs, with Contiguous Allocation providing fast access but causing fragmentation, Linked Allocation eliminating fragmentation but slowing access, and Indexed Allocation allowing direct access without requiring contiguous blocks at the cost of additional index storage.