K M M INSTITUTE OF POSTGRADUATE STUDIES

(Affiliated to Sri Venkateswara University, Tirupati)
RAMIREDDIPALLE, TIRUPATI-517 102

DEAPARTMENT OF COMPUTER APPLICATIONS

CERTIFICATE

REGISTERED NO: _____

This is to certify that _____ *is a bonafide student of MCA (I Semester) course in our Institution. The student has done the Lab work as per this record for this practical fulfillment of the requirement of the MCA course* ***MCA108 P -OPERATING SYSTEM LAB****(****Subject Code & Subject Title****)****during the I****Semester of the Academic Year 2025-2027.*

Head-of the Department *Lecturer-In-Charge*

Attended and submitted for the University Practical Examination.

Signature of Examiners: *1.*_____
(External Examiner-1)

*2.*_____
(External Examiner-2)

CONTENTS

1. Write a python program to implement the concept of first in first out job scheduling

Aim: To implement and simulate the First in First out (FIFO) job scheduling algorithm, where jobs are executed in the order they arrive in the system.

Description:FIFO is one of the simplest CPU scheduling algorithms. It executes processes in the order they arrive in the ready queue. The process that arrives first will be executed first, and the process that arrives later will wait until the earlier process finishes. It is easy to implement but does not guarantee the best performance, especially in terms of average waiting time and turnaround time.

Algorithm:

1. Maintain a queue that stores jobs.
2. For each job, calculate the waiting time and turnaround time.
3. Execute each job in the order it arrived (FIFO).
4. For each job:
    - Waiting time = Time spent waiting in the queue before the job starts execution.
    - Turnaround time = Time from job arrival to job completion.
5. Output the average waiting time and average turnaround time

Source code:

```
# Python Program to implement FIFO Job Scheduling
class Job:
def __init__(self, name, arrival_time, burst_time):
      self.name = name
self.arrival_time = arrival_time
self.burst_time = burst_time
self.completion_time = 0
self.turnaround_time = 0
self.waiting_time = 0
deffifo_scheduling(jobs):
   # Sorting jobs by their arrival time (First In First Out)
jobs.sort(key=lambda job: job.arrival_time)
   # To keep track of the time at which the CPU will be free
current_time = 0
for job in jobs:
      # Waiting time is the difference between current time and arrival time
job.waiting_time = current_time - job.arrival_time
      # Turnaround time is the sum of waiting time and burst time
job.turnaround_time = job.waiting_time + job.burst_time
      # Completion time is when the job finishes
job.completion_time = current_time + job.burst_time
      # Update the current time (time after job execution)
current_time = job.completion_time
   # Calculate average waiting time and turnaround time
total_waiting_time = sum(job.waiting_time for job in jobs)
total_turnaround_time = sum(job.turnaround_time for job in jobs)
avg_waiting_time = total_waiting_time / len(jobs)
avg_turnaround_time = total_turnaround_time / len(jobs)
   # Print Results
print("Job Scheduling (FIFO):")
print("Job\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time")
for job in jobs:
print(f"{job.name}\t\t{job.arrival_time}\t\t{job.burst_time}\t\t{job.completion_time}\t\t{job.waiting_time}\
t\t{job.turnaround_time}"
print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")
# Example usage of the FIFO scheduling algorithm
if __name__ == "__main__":
   # List of jobs (name, arrival time, burst time)
jobs = [
Job("P1", 0, 5),
Job("P2", 1, 3),
Job("P3", 2, 8),
Job("P4", 3, 6)
   ]
fifo_scheduling(jobs)
```

Output:

Job Scheduling (FIFO):

Job	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time
P1	0	5	5	0	5
P2	1	3	8	4	7
P3	2	8	16	6	14
P4	3	6	22	10	16

Average Waiting Time: 5.00

Average Turnaround Time: 10.50

Conclusion:

The above program executed successfully

2. Write a python program to implement the concept of shortest job first scheduling

Aim: The aim of this program is to implement the Shortest Job First (SJF) CPU scheduling algorithm, which selects the process with the shortest burst time for execution first.

Description: Shortest Job First (SJF) is a non-preemptive scheduling algorithm where the process with the shortest burst time (execution time) is executed first. This algorithm is optimal in terms of minimizing the average waiting time but can suffer from the "starvation" problem, where longer processes may have to wait indefinitely if shorter processes keep arriving. SJF is ideal when the burst time of a process is known in advance.

Algorithm:

1. Input:
    - Number of processes.
    - Burst time for each process.
2. Sort the processes by their burst times in ascending order (shortest burst time first).
3. Execution Order:
    - Select the process with the shortest burst time for execution.
    - Calculate waiting time and turnaround time for each process:
        - Waiting Time: Time spent waiting in the ready queue before execution starts.
        - Turnaround Time: Time from arrival to completion (waiting time + burst time).
4. Output:
    - Completion time, waiting time, and turnaround time for each process.
    - Average waiting time and average turnaround time.

Source Code:

```
# Python Program to implement Shortest Job First (SJF) Scheduling

defsjf_scheduling(processes, burst_times):
   n = len(processes)
   # Sorting processes based on their burst times (Shortest Job First)
processes_sorted = sorted(zip(processes, burst_times), key=lambda x: x[1])

   # Unzip the sorted processes
processes, burst_times = zip(*processes_sorted)

   # Initialize time-related variables
waiting_times = [0] * n
turnaround_times = [0] * n
completion_times = [0] * n

   # Calculate Completion time for each process
completion_times[0] = burst_times[0]
for i in range(1, n):
completion_times[i] = completion_times[i - 1] + burst_times[i]

   # Calculate Waiting time and Turnaround time for each process
for i in range(n):
turnaround_times[i] = completion_times[i]  # Turnaround Time = Completion Time
waiting_times[i] = turnaround_times[i] - burst_times[i]  # Waiting Time = Turnaround Time - Burst Time

   # Calculate Average Waiting Time and Average Turnaround Time
avg_waiting_time = sum(waiting_times) / n
avg_turnaround_time = sum(turnaround_times) / n

   # Display the results
print("\nShortest Job First Scheduling Results:")
print("Process\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time")
for i in range(n):
print(f"{processes[i]}\t\t{burst_times[i]}\t\t{completion_times[i]}\t\t{waiting_times[i]}\t\t{turnaround_times
[i]}")

print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")

# Main function to take user input and invoke the SJF scheduling function
def main():
   # Take number of processes as input
   n = int(input("Enter the number of processes: "))

processes = []
```

```
burst_times = []

   # Take process names and burst times as input
for i in range(n):
process_name = input(f"Enter name of process {i+1}: ")
burst_time = int(input(f"Enter burst time for {process_name}: "))
processes.append(process_name)
burst_times.append(burst_time)

   # Call the SJF scheduling function
sjf_scheduling(processes, burst_times)

# Run the main function
if __name__ == "__main__":
main()
```

Output:

Enter the number of processes: 4
Enter name of process 1: P1
Enter burst time for P1: 6
Enter name of process 2: P2
Enter burst time for P2: 2
Enter name of process 3: P3
Enter burst time for P3: 8
Enter name of process 4: P4
Enter burst time for P4: 3

Shortest Job First Scheduling Results:

Process	Burst Time	Completion Time	Waiting Time	Turnaround Time
P2	2	2	0	2
P4	3	5	2	5
P1	6	11	5	11
P3	8	19	11	19

Average Waiting Time: 4.50
Average Turnaround Time: 9.25

Conclusion:

The above program executed successfully

3.Write a python program to implement the concept of round robin job scheduling

Aim: To implement the **Round Robin (RR)** job scheduling algorithm, where processes are executed in a cyclic order, each for a fixed time slice or quantum.

Description: Round Robin (RR) is a preemptive scheduling algorithm where each process is assigned a fixed time slice (also known as a time quantum). The CPU scheduler goes around the ready queue, giving each process in the ready queue a chance to execute for a time quantum. If a process doesn't complete within its allocated time quantum, it is placed back in the ready queue, and the CPU scheduler moves to the next process.

Algorithm:

1. **Input:**
    - Number of processes.
    - Burst times for each process.
    - Time quantum (time slice).
2. **Execution:**
    - Each process is given a time quantum (or slice) to execute.
    - If a process doesn’t finish within its time quantum, it’s added back to the ready queue with the remaining burst time.
    - If a process finishes, it’s removed from the ready queue.
    - Continue this cycle until all processes are completed.
3. **Output:**
    - Completion time, waiting time, and turnaround time for each process.
    - Average waiting time and average turnaround time.

Source code:

```
# Python Program to implement Round Robin Scheduling

defround_robin_scheduling(processes, burst_times, quantum):
   n = len(processes)
remaining_burst_times = burst_times[:]
waiting_times = [0] * n
turnaround_times = [0] * n
completion_times = [0] * n
time = 0  # Track the current time

   # Keep track of processes that are yet to be completed
remaining_processes = n

whileremaining_processes> 0:
for i in range(n):
ifremaining_burst_times[i] > 0:  # If the process has not finished
ifremaining_burst_times[i] > quantum:
            # Process gets executed for one time quantum
time += quantum
remaining_burst_times[i] -= quantum
else:
            # Process finishes execution
time += remaining_burst_times[i]
completion_times[i] = time
remaining_burst_times[i] = 0
remaining_processes -= 1

   # Calculate waiting and turnaround times
for i in range(n):
turnaround_times[i] = completion_times[i]
waiting_times[i] = turnaround_times[i] - burst_times[i]

   # Calculate average waiting time and average turnaround time
avg_waiting_time = sum(waiting_times) / n
avg_turnaround_time = sum(turnaround_times) / n

   # Print results
print("\nRound Robin Scheduling Results:")
print("Process\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time")
for i in range(n):
print(f"{processes[i]}\t\t{burst_times[i]}\t\t{completion_times[i]}\t\t{waiting_times[i]}\t\t{turnaround_times
[i]}")

print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")

# Main function to take user input and invoke the Round Robin scheduling function
def main():
   # Take number of processes as input
   n = int(input("Enter the number of processes: "))
```

```
processes = []
burst_times = []

   # Take process names and burst times as input
for i in range(n):
process_name = input(f"Enter name of process {i+1}: ")
burst_time = int(input(f"Enter burst time for {process_name}: "))
processes.append(process_name)
burst_times.append(burst_time)

   # Take time quantum as input
quantum = int(input("Enter the time quantum: "))

   # Call the Round Robin scheduling function
round_robin_scheduling(processes, burst_times, quantum)

# Run the main function
if __name__ == "__main__":
main()
```

Output:

```
Enter the number of processes: 4
Enter name of process 1: P1
Enter burst time for P1: 6
Enter name of process 2: P2
Enter burst time for P2: 2
Enter name of process 3: P3
Enter burst time for P3: 8
Enter name of process 4: P4
Enter burst time for P4: 3
Enter the time quantum: 4

Round Robin Scheduling Results:
Process  Burst Time   Completion Time    Waiting Time   Turnaround Time
P1        6          16              10             16
P2        2          4               2             4
P3        8          20              12             20
P4        3          12              9             12

Average Waiting Time: 8.25
Average Turnaround Time: 13.00
```

Conclusion:

The above program executed successfully

4.Write a python program to implement the concept of priority scheduling

Aim: The aim of this program is to implement the **Priority Scheduling** algorithm, where processes are scheduled based on their priority.

Description: Priority Scheduling is a non-preemptive scheduling algorithm where each process is assigned a priority. The process with the highest priority (lowest priority number) is executed first. If two processes have the same priority, they are scheduled based on their order of arrival. This algorithm is commonly used in systems where certain tasks require more urgent processing.

Algorithm:

1. **Input:**
    - Number of processes.
    - Burst times for each process.
    - Priorities for each process.
2. **Execution:**
    - Sort the processes by their priority (lower priority number means higher priority).
    - Once sorted, execute the processes in order of their priority.
    - Calculate waiting time and turnaround time for each process:
        - **Waiting Time:** Time spent waiting in the ready queue before execution starts.
        - **Turnaround Time:** Time from arrival to completion (waiting time + burst time).
3. **Output:**
    - Completion time, waiting time, and turnaround time for each process.
    - Average waiting time and average turnaround time.

Source Code:

```
# Python Program to implement Priority Scheduling

defpriority_scheduling(processes, burst_times, priorities):
   n = len(processes)

   # Creating a list of tuples where each tuple is (process, burst_time, priority)
process_data = [(processes[i], burst_times[i], priorities[i]) for i in range(n)]

   # Sorting processes by priority (ascending order: lower priority number means higher priority)
process_data.sort(key=lambda x: x[2])

   # Unzip the sorted process data
processes, burst_times, priorities = zip(*process_data)

waiting_times = [0] * n
turnaround_times = [0] * n
completion_times = [0] * n
time = 0  # Track the current time

   # Calculate Completion Time for each process
for i in range(n):
time += burst_times[i]
completion_times[i] = time

   # Calculate Waiting Time and Turnaround Time for each process
for i in range(n):
turnaround_times[i] = completion_times[i]  # Turnaround Time = Completion Time
waiting_times[i] = turnaround_times[i] - burst_times[i]  # Waiting Time = Turnaround Time - Burst Time

   # Calculate Average Waiting Time and Average Turnaround Time
avg_waiting_time = sum(waiting_times) / n
avg_turnaround_time = sum(turnaround_times) / n

   # Print results
print("\nPriority Scheduling Results:")
print("Process\tBurst Time\tPriority\tCompletion Time\tWaiting Time\tTurnaround Time")
for i in range(n):
print(f"{processes[i]}\t\t{burst_times[i]}\t\t{priorities[i]}\t\t{completion_times[i]}\t\t{waiting_times[i]}\t\t{tu
rnaround_times[i]}")

print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")

# Main function to take user input and invoke the Priority scheduling function
def main():
   # Take number of processes as input
   n = int(input("Enter the number of processes: "))

processes = []
burst_times = []
priorities = []
```

```
    # Take process names, burst times, and priorities as input
for i in range(n):
process_name = input(f"Enter name of process {i+1}: ")
burst_time = int(input(f"Enter burst time for {process_name}: "))
priority = int(input(f"Enter priority for {process_name} (lower number means higher priority): "))
processes.append(process_name)
burst_times.append(burst_time)
priorities.append(priority)

    # Call the Priority scheduling function
priority_scheduling(processes, burst_times, priorities)

# Run the main function
if __name__ == "__main__":
main()
```

Output:

```
Enter the number of processes: 4
Enter name of process 1: P1
Enter burst time for P1: 6
Enter priority for P1 (lower number means higher priority): 3
Enter name of process 2: P2
Enter burst time for P2: 2
Enter priority for P2 (lower number means higher priority): 1
Enter name of process 3: P3
Enter burst time for P3: 8
Enter priority for P3 (lower number means higher priority): 4
Enter name of process 4: P4
Enter burst time for P4: 3
Enter priority for P4 (lower number means higher priority): 2

Priority Scheduling Results:
Process  Burst Time   Priority    Completion Time    Waiting Time    Turnaround Time
P2          2            1           2                0               2
P4          3            2           5                2               5
P1          6            3           11               5               11
P3          8            4           19               11              19

Average Waiting Time: 4.50
Average Turnaround Time: 9.25
```

Conclusion:

The above program executed successfully

5.Write a python program implement the concept of banker's algorithm

Aim: The aim of this program is to implement the **Banker's Algorithm**, which is used to determine whether a system is in a **safe state** or not.

Description: The **Banker's Algorithm** is a deadlock avoidance algorithm that works by checking if the system can allocate resources to processes in a way that avoids a deadlock. It uses the following components:

1. **Available Resources:** The number of available resources in the system.
2. **Maximum Resources:** The maximum resources each process may need.
3. **Allocated Resources:** The resources already allocated to each process.
4. **Need Matrix:** The remaining resources needed by each process to complete.

Algorithm:

- **Input:**
  - Number of processes.
  - Number of resource types.
  - Available resources.
  - Maximum resources required by each process.
  - Resources allocated to each process.
- **Initialization:**
  - Calculate the **Need Matrix**: Need[i][j] = Max[i][j] - Allocated[i][j]
- **Safety Check:**
  - Initially, the system has all the available resources.
  - Find a process i whose **Need** is less than or equal to the available resources.
  - If such a process is found, simulate its completion by adding its allocated resources back to the available resources.
  - Repeat the above steps until all processes are finished or no such process can be found.
- **Output:**
  - If all processes are finished successfully, the system is in a **safe state**.
  - Otherwise, the system is in an **unsafe state**.

Source Code:

```
# Python Program to implement Banker's Algorithm for resource allocation

defbankers_algorithm(processes, avail, max_res, alloc_res):
   n = len(processes)  # number of processes
   m = len(avail)  # number of resources

   # Step 1: Calculate the Need Matrix
need_res = [[max_res[i][j] - alloc_res[i][j] for j in range(m)] for i in range(n)]

   # Step 2: Initialize work and finish arrays
work = avail[:]  # Available resources
finish = [False] * n  # Finish array, initially all are False

   # Step 3: Check if the system is in a safe state
safe_sequence = []
whilelen(safe_sequence) < n:
      # Find a process that can be executed
progress_made = False
for i in range(n):
if not finish[i] and all(need_res[i][j] <= work[j] for j in range(m)):
            # If the process can be executed, allocate its resources and mark as finished
for j in range(m):
work[j] += alloc_res[i][j]
finish[i] = True
safe_sequence.append(processes[i])
progress_made = True
break

      # If no progress is made, the system is in an unsafe state
if not progress_made:
return False, []

return True, safe_sequence

# Main function to take user input and invoke Banker's Algorithm
def main():
   # Take the number of processes and resources
   n = int(input("Enter the number of processes: "))
   m = int(input("Enter the number of resource types: "))

processes = []
avail = []
max_res = []
alloc_res = []

   # Input process names
for i in range(n):
processes.append(input(f"Enter name of process {i + 1}: "))

   # Input available resources
avail = list(map(int, input("Enter available resources (space-separated): ").split()))
```

```
    # Input maximum resources needed by each process
max_res = []
for i in range(n):
max_res.append(list(map(int, input(f"Enter maximum resources needed by process {processes[i]} (space-separated): ").split())))

    # Input currently allocated resources to each process
alloc_res = []
for i in range(n):
alloc_res.append(list(map(int, input(f"Enter allocated resources to process {processes[i]} (space-separated): ").split())))

    # Invoke Banker's Algorithm to check if the system is in a safe state
safe, safe_sequence = bankers_algorithm(processes, avail, max_res, alloc_res)

if safe:
print("\nThe system is in a safe state.")
print(f"Safe sequence: {' -> '.join(safe_sequence)}")
else:
print("\nThe system is in an unsafe state.")

# Run the main function
if __name__ == "__main__":
main()
```

Output:

```
Enter the number of processes: 3
Enter the number of resource types: 3
Enter name of process 1: P1
Enter name of process 2: P2
Enter name of process 3: P3
Enter available resources (space-separated): 3 3 2
Enter maximum resources needed by process P1 (space-separated): 7 5 3
Enter maximum resources needed by process P2 (space-separated): 3 2 2
Enter maximum resources needed by process P3 (space-separated): 9 0 2
Enter allocated resources to process P1 (space-separated): 2 1 1
Enter allocated resources to process P2 (space-separated): 2 1 1
Enter allocated resources to process P3 (space-separated): 3 2 2

The system is in a safe state.
Safe sequence: P2 -> P1 -> P3
```

Conclusion:

The above program executed successfully

6.Write a python program to implement the concept of dining philosopher problem

Aim: The aim of the program is to simulate the classic **Dining Philosophers Problem** using Python.

Description: In this simulation, philosophers are represented as threads that repeatedly do the following:
Think - The philosopher is thinking for some time.

1. **Become Hungry** - The philosopher attempts to acquire two forks to eat.
2. **Eat** - The philosopher eats for a random amount of time.
3. **Release Forks** - After eating, the philosopher releases both forks.

Algorithm:

- Initialize the number of philosophers and forks.
- Each philosopher will be represented by a thread.
- Each philosopher will alternate between thinking, getting hungry, eating, and releasing forks.
- Synchronization (locks) will ensure that two philosophers cannot pick the same fork at the same time.

Source code:

```
import threading
import time
import random

# Function to handle each philosopher's actions
def philosopher(i, forks, think_time_range, eat_time_range):
while True:
    # Thinking
print(f"Philosopher {i} is thinking.")
time.sleep(random.uniform(*think_time_range))  # Thinking time range

print(f"Philosopher {i} is hungry.")

    # Forks: Left fork is the philosopher's fork, right fork is the next one
left_fork = forks[i]  # Fork on the left
right_fork = forks[(i + 1) % len(forks)]  # Fork on the right

    # Try to acquire both forks
left_fork.acquire()
print(f"Philosopher {i} picked up left fork.")

    # Try to acquire the right fork
right_fork.acquire()
print(f"Philosopher {i} picked up right fork.")

    # Eating
print(f"Philosopher {i} is eating.")
time.sleep(random.uniform(*eat_time_range))  # Eating time range

    # After eating, release both forks
left_fork.release()
print(f"Philosopher {i} put down left fork.")
right_fork.release()
print(f"Philosopher {i} put down right fork.")

    # Philosopher is done and goes back to thinking

def main():
  # User inputs
num_philosophers = int(input("Enter the number of philosophers: "))
min_think_time = float(input("Enter the minimum thinking time (in seconds): "))
max_think_time = float(input("Enter the maximum thinking time (in seconds): "))
min_eat_time = float(input("Enter the minimum eating time (in seconds): "))
max_eat_time = float(input("Enter the maximum eating time (in seconds): "))

  # Define forks (locks) for each philosopher
forks = [threading.Lock() for _ in range(num_philosophers)]

  # Define the time ranges for thinking and eating
think_time_range = (min_think_time, max_think_time)
eat_time_range = (min_eat_time, max_eat_time)
```

```
   # Create and start threads for each philosopher
threads = []
for i in range(num_philosophers):
      t = threading.Thread(target=philosopher, args=(i, forks, think_time_range, eat_time_range))
threads.append(t)
t.start()

   # Join all threads (this line will block the main thread until all threads finish)
for t in threads:
t.join()

# Run the main function
if __name__ == "__main__":
main()
```

Output:

```
Enter the number of philosophers: 5
Enter the minimum thinking time (in seconds): 1
Enter the maximum thinking time (in seconds): 3
Enter the minimum eating time (in seconds): 2
Enter the maximum eating time (in seconds): 4

Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is hungry.
Philosopher 0 picked up left fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 0 put down left fork.
Philosopher 0 put down right fork.
Philosopher 1 is hungry.
...
```

Conclusion:

The above program executed successfully

7.**Write a python program to implement the concept of producer and consumer problem**

Aim: The goal of the program is to simulate a shared buffer between a producer and a consumer

Description: In this problem, there is a shared buffer that can hold a limited number of items. The producer produces items and puts them in the buffer. The consumer takes items from the buffer and consumes them. The producer should wait if the buffer is full, and the consumer should wait if the bufferis empty.

Algorithm:

1. Initialize a **buffer** with a defined maximum size.
2. The **producer** produces items and adds them to the buffer.
3. The **consumer** consumes items from the buffer.
4. If the buffer is full, the producer waits. If the buffer is empty, the consumer waits.
5. Synchronization is managed using **semaphores** or **locks** to prevent race conditions.

Source Code:

```
import threading
import time
import random

# Define the buffer and synchronization mechanisms
class Buffer:
def __init__(self, size):
self.size = size
self.buffer = []
self.lock = threading.Lock()  # To prevent race conditions
self.not_full = threading.Condition(self.lock)  # Condition variable to check buffer not full
self.not_empty = threading.Condition(self.lock)  # Condition variable to check buffer not empty

   # Produce an item and put it in the buffer
def produce(self, item):
withself.lock:
        # Wait until there is space in the buffer
whilelen(self.buffer) == self.size:
self.not_full.wait()
        # Produce the item and add it to the buffer
self.buffer.append(item)
print(f"Produced: {item}. Buffer: {self.buffer}")
self.not_empty.notify()  # Notify consumer that there is data available

   # Consume an item from the buffer
def consume(self):
withself.lock:
        # Wait until there is an item in the buffer
whilelen(self.buffer) == 0:
self.not_empty.wait()
        # Consume the item from the buffer
item = self.buffer.pop(0)
print(f"Consumed: {item}. Buffer: {self.buffer}")
self.not_full.notify()  # Notify producer that there is space available
return item

# Producer function
def producer(buffer, producer_id):
while True:
      # Produce a random item (in this case, a random integer)
item = random.randint(1, 100)
buffer.produce(item)
time.sleep(random.uniform(0.5, 2))  # Random sleep to simulate time taken to produce an item

# Consumer function
def consumer(buffer, consumer_id):
while True:
      # Consume an item from the buffer
buffer.consume()
time.sleep(random.uniform(0.5, 2))  # Random sleep to simulate time taken to consume an item
```

```
def main():
   # User inputs
buffer_size = int(input("Enter the size of the buffer: "))
num_producers = int(input("Enter the number of producers: "))
num_consumers = int(input("Enter the number of consumers: "))

   # Create the shared buffer
buffer = Buffer(buffer_size)

   # Create and start producer threads
producers = []
for i in range(num_producers):
producer_thread = threading.Thread(target=producer, args=(buffer, i))
producers.append(producer_thread)
producer_thread.start()

   # Create and start consumer threads
consumers = []
for i in range(num_consumers):
consumer_thread = threading.Thread(target=consumer, args=(buffer, i))
consumers.append(consumer_thread)
consumer_thread.start()

   # Join all threads (this line will block the main thread until all threads finish)
for t in producers + consumers:
t.join()

# Run the main function
if __name__ == "__main__":
main()
```

Output:

```
Enter the size of the buffer: 5
Enter the number of producers: 2
Enter the number of consumers: 2

Produced: 43. Buffer: [43]
Produced: 82. Buffer: [43, 82]
Consumed: 43. Buffer: [82]
Produced: 18. Buffer: [82, 18]
Produced: 91. Buffer: [82, 18, 91]
Consumed: 82. Buffer: [18, 91]
Consumed: 18. Buffer: [91]
Produced: 65. Buffer: [91, 65]
...
```

Conclusion:

The above program executed successfully

8.Write a java program to implement best fit algorithm

Aim: To implement the **Best Fit memory allocation algorithm** using Python, which allocates a process to the smallest available memory block that is sufficient for its size, thereby reducing memory wastage.

Description: The **Best Fit algorithm** is a memory management technique used by operating systems. When a process requests memory, the algorithm searches through all free memory blocks and selects the **smallest block that is large enough** to satisfy the request.

Algorithm :

1. Read number of memory blocks and their sizes.
2. Read number of processes and their memory requirements.
3. For each process:
    - Find the smallest memory block that is ≥ process size.
    - Allocate the process to that block.
    - Reduce the block size accordingly.
4. Display allocation results.

Source Code:

```
# Best Fit memory allocation algorithm
defbest_fit(blocks, processes):
allocation = [-1] * len(processes)

   # Iterate through each process and find the best block
for i in range(len(processes)):
best_index = -1

      # Search for the best fit block
for j in range(len(blocks)):
if blocks[j] >= processes[i]:
ifbest_index == -1 or blocks[j] < blocks[best_index]:
best_index = j

      # If a suitable block is found, allocate it
ifbest_index != -1:
allocation[i] = best_index
blocks[best_index] -= processes[i]  # Reduce the block size after allocation

return allocation

# Main program
def main():
   # User input for memory blocks
   n = int(input("Enter number of memory blocks: "))
blocks = []

print("Enter size of each block:")
for i in range(n):
blocks.append(int(input(f"Block {i+1}: ")))

   # User input for processes
   m = int(input("\nEnter number of processes: "))
processes = []

print("Enter memory required for each process:")
for i in range(m):
processes.append(int(input(f"Process {i+1}: ")))

   # Get the allocation result using Best Fit
allocation = best_fit(blocks, processes)

   # Display the results
print("\nProcess No.\tProcess Size\tBlock No.")
for i in range(m):
```

```
if allocation[i] != -1:
print(f"{i+1}\t\t{processes[i]}\t\t{allocation[i]+1}")
else:
print(f"{i+1}\t\t{processes[i]}\t\tNot Allocated")

if __name__ == "__main__":
main()
```

Output:

Enter number of memory blocks: 5
Enter size of each block:
Block 1: 500
Block 2: 300
Block 3: 400
Block 4: 200
Block 5: 600

Enter number of processes: 4
Enter memory required for each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Conclusion:

The above program executed successfully

9.**Write a java program to implement the concept of memory fragmentation**

Aim: The aim of this program is to simulate **memory fragmentation**, a situation where free memory is divided into small, non-contiguous blocks due to allocation and deallocation of memory spaces over time.

Description: In a simple memory management system, memory is divided into fixed-size blocks. As processes request and release memory, the memory space becomes fragmented. There are two types of fragmentation.

Algorithm:

- **Initialization**:
  - Define a memory pool (array of blocks).
  - Set up a list of processes that require memory.
- **Memory Allocation**:
  - Try to allocate memory for a process in a free block.
  - If no free block is large enough, fragmentation occurs.
- **MemoryDeallocation**:
  - Free the memory when a process finishes, making that block available for future allocations.
- **Simulation of Fragmentation**:
  - Track allocated and free blocks and display memory state to observe fragmentation

Source code:

```
# Memory Block class
classMemoryBlock:
def __init__(self, size):
self.size = size
self.free = True
self.process_id = None

# Memory Manager class
classMemoryManager:
def __init__(self, total_size, block_size):
self.block_size = block_size
self.blocks = [MemoryBlock(block_size) for _ in range(total_size // block_size)]

defdisplay_memory(self):
print("\nMemory Status:")
for i, block in enumerate(self.blocks):
ifblock.free:
print(f"Block {i}: Free")
else:
print(f"Block {i}: Allocated to Process {block.process_id}")
print()

def allocate(self, process_id, required_blocks):
count = 0
start_index = -1

for i, block in enumerate(self.blocks):
ifblock.free:
if count == 0:
start_index = i
count += 1
if count == required_blocks:
for j in range(start_index, start_index + required_blocks):
self.blocks[j].free = False
self.blocks[j].process_id = process_id
print(f"Process {process_id} allocated {required_blocks} blocks")
return
else:
count = 0

print(f"Process {process_id} allocation failed due to fragmentation")

defdeallocate(self, process_id):
found = False
for block in self.blocks:
```

```
ifblock.process_id == process_id:
block.free = True
block.process_id = None
found = True

if found:
print(f"Process {process_id} deallocated successfully")
else:
print(f"Process {process_id} not found in memory")

# Main program
def main():
total_memory = int(input("Enter total memory size: "))
block_size = int(input("Enter block size: "))

manager = MemoryManager(total_memory, block_size)
while True:
print("\n1. Allocate Memory")
print("2. Deallocate Memory")
print("3. Display Memory")
print("4. Exit")
choice = int(input("Enter your choice: "))
if choice == 1:
pid = int(input("Enter process ID: "))
memory_required = int(input("Enter memory required: "))
blocks_needed = memory_required // block_size
ifmemory_required % block_size != 0:
blocks_needed += 1
manager.allocate(pid, blocks_needed)
elif choice == 2:
pid = int(input("Enter process ID to deallocate: "))
manager.deallocate(pid)
elif choice == 3:
manager.display_memory()

elif choice == 4:
print("Exiting program")
break
else:
print("Invalid choice")

if __name__ == "__main__":
main()
```

Output:

```
Enter total memory size: 500
Enter block size: 50

1. Allocate Memory
2. Deallocate Memory
3. Display Memory
4. Exit
Enter your choice: 1
Enter process ID: 1
Enter memory required: 120
Process 1 allocated 3 blocks
```

Conclusion:

The above program executed successfully

10. Write a python program to implement the concept of FIFO page replacement.

Aim: To implement the **FIFO (First In, First Out)** page replacement algorithm using Python, where pages are loaded into memory, and when memory is full, the oldest page (the one loaded first) is replaced with the new page.

Description: The FIFO page replacement algorithm works by maintaining a queue of pages currently in memory. When a new page needs to be loaded into memory and the memory is full, the oldest page (the one that was first loaded) is replaced by the new page.

Algorithm:

1. Initialize an empty queue for the page frames.
2. Start reading page references one by one.
3. If the page is not in memory (page fault), add the page to the memory.
    - If the memory is full, remove the page at the front of the queue and add the new page at the back.
4. Keep track of the pages and display memory status at each step.

Source Code:

```
# FIFO Page Replacement Algorithm in Python

deffifo_page_replacement(pages, num_frames):
memory = []  # This will hold the pages currently in memory
page_faults = 0  # To count the number of page faults

for page in pages:
if page not in memory:
iflen(memory) <num_frames:
        # If there's space in memory, just add the page
memory.append(page)
else:
        # Memory is full, remove the first page (FIFO) and add the new page
memory.pop(0)  # Remove the oldest page
memory.append(page)
page_faults += 1
    # Display the current state of memory (for visualization)
print(f"Page {page} referenced -> Memory: {memory}")

print(f"\nTotal Page Faults: {page_faults}")


# Taking input for pages and frame size
if __name__ == "__main__":
print("FIFO Page Replacement Algorithm\n")

  # Input the page reference string
pages = list(map(int, input("Enter the page reference string (space separated): ").split()))

  # Input the number of frames
num_frames = int(input("Enter the number of frames: "))

  # Call the FIFO page replacement function
fifo_page_replacement(pages, num_frames)
```

Output:

```
Enter the page reference string (space separated): 7 0 1 2 0 3 0 4 2 3 0 3 2 3
Enter the number of frames: 4

FIFO Page Replacement Algorithm

Page 7 referenced -> Memory: [7]
Page 0 referenced -> Memory: [7, 0]
Page 1 referenced -> Memory: [7, 0, 1]
Page 2 referenced -> Memory: [7, 0, 1, 2]
Page 0 referenced -> Memory: [7, 0, 1, 2]
Page 3 referenced -> Memory: [0, 1, 2, 3]
Page 0 referenced -> Memory: [1, 2, 3, 0]
Page 4 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]
Page 0 referenced -> Memory: [4, 2, 3, 0]
Page 3 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]

Total Page Faults: 12
```

Conclusion:

The above program executed successfully

11. **Write a python program to implement the concept of LRU page replacement**

Aim:
To implement the **LRU (Least Recently Used)** page replacement algorithm in Python, where pages are loaded into memory, and when memory is full, the least recently used page (the page that has not been referenced for the longest time) is replaced with the new page.

Description:
The **LRU (Least Recently Used)** algorithm replaces the page that has not been used for the longest period of time. It keeps track of the most recently used pages and, when the memory is full and a page needs to be loaded, it removes the page that was least recently used.

The basic idea is to maintain an ordered list (or use a data structure like a queue or a dictionary) that stores the pages in the order of their use. When a page is accessed, it becomes the most recently used. If the page is not in memory, a page fault occurs, and if memory is full, the least recently used page is removed.

Algorithm:

1. Maintain a list or queue to represent the memory and track the most recent and least recent pages.
2. For each page reference:
    - If the page is already in memory (hit), move it to the most recent position.
    - If the page is not in memory (miss), and there is space in memory, add the page.
    - If memory is full, remove the least recently used page and add the new page.
3. Keep track of page faults (when a page is loaded into memory).
4. Display the current state of memory after each reference.

Source code:

```
from collections import deque

# LRU Page Replacement Algorithm in Python
deflru_page_replacement(pages, num_frames):
memory = deque()  # This will hold the pages currently in memory
page_set = set()  # Set for O(1) lookup of pages in memory
page_faults = 0  # To count the number of page faults

for page in pages:
      # If the page is not in memory
if page not in page_set:
page_faults += 1

         # If there's space in memory, add the new page
iflen(memory) <num_frames:
memory.append(page)
            page_set.add(page)
else:
            # If memory is full, remove the least recently used page
lru_page = memory.popleft()  # Remove the least recently used page
page_set.remove(lru_page)  # Remove from the set
memory.append(page)  # Add the new page
            page_set.add(page)  # Add the new page to the set
else:
         # If the page is already in memory, move it to the end (most recent)
memory.remove(page)
memory.append(page)

      # Display the current state of memory (for visualization)
print(f"Page {page} referenced -> Memory: {list(memory)}")

print(f"\nTotal Page Faults: {page_faults}")

# Taking input for pages and frame size
if __name__ == "__main__":
print("LRU Page Replacement Algorithm\n")

   # Input the page reference string
pages = list(map(int, input("Enter the page reference string (space separated): ").split()))

   # Input the number of frames
num_frames = int(input("Enter the number of frames: "))

   # Call the LRU page replacement function
lru_page_replacement(pages, num_frames)
```

Output:

```
Enter the page reference string (space separated): 7 0 1 2 0 3 0 4 2 3 0 3 2 3
Enter the number of frames: 4

LRU Page Replacement Algorithm

Page 7 referenced -> Memory: [7]
Page 0 referenced -> Memory: [7, 0]
Page 1 referenced -> Memory: [7, 0, 1]
Page 2 referenced -> Memory: [7, 0, 1, 2]
Page 0 referenced -> Memory: [7, 1, 2, 0]
Page 3 referenced -> Memory: [1, 2, 0, 3]
Page 0 referenced -> Memory: [1, 2, 3, 0]
Page 4 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]
Page 0 referenced -> Memory: [4, 2, 3, 0]
Page 3 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]

Total Page Faults: 12
```

Conclusion:

The above program executed successfully

12.**Write a java program to implement the concept of optimal page replacement**

Aim:
To implement the **Optimal Page Replacement** algorithm in Python, where pages are loaded into memory, and when memory is full, the page that will not be used for the longest period in the future is replaced with the new page.

Description:
The **Optimal Page Replacement** algorithm works by selecting the page that is **farthest in the future** for replacement. This means that when the memory is full and a page fault occurs, the algorithm checks which page in memory will not be needed for the longest time in the future and replaces that page.

While this algorithm guarantees the minimum number of page faults, it is not practical in real systems because it requires knowledge of future page requests, which is not available in practice. However, it is useful as a benchmark for comparing other algorithms like FIFO, LRU, etc.

Algorithm:

1. Maintain a list to represent the pages in memory.
2. For each page reference:
    - If the page is already in memory (a hit), do nothing.
    - If the page is not in memory (a miss), check if there is space in memory:
        - If there is space, simply add the page.
        - If there is no space, find the page that will not be used for the longest period in the future and replace it.
3. Keep track of the number of page faults (when a page is added to memory).
4. Display the current state of memory after each reference.

Source code:

```
# Optimal Page Replacement Algorithm in Python

defoptimal_page_replacement(pages, num_frames):
memory = []  # This will hold the pages currently in memory
page_faults = 0  # To count the number of page faults

forcurrent_time, page in enumerate(pages):
    # If the page is not in memory
if page not in memory:
page_faults += 1
        # If there's space in memory, add the page
iflen(memory) <num_frames:
memory.append(page)
else:
            # If memory is full, replace the optimal page
            # Find the page that is used farthest in the future (or never used again)
farthest_use = -1
page_to_replace = None

for p in memory:
try:
next_use = pages[current_time+1:].index(p)  # Index of next use of the page
exceptValueError:
next_use = float('inf')  # If the page is not used again, it's farthest

                # Find the page with the farthest next use (or never used again)
ifnext_use>farthest_use:
farthest_use = next_use
page_to_replace = p
            # Remove the page that will be used farthest in the future and add the new page
memory.remove(page_to_replace)
memory.append(page)

    # Display the current state of memory (for visualization)
print(f"Page {page} referenced -> Memory: {memory}")
print(f"\nTotal Page Faults: {page_faults}")
# Taking input for pages and frame size
if __name__ == "__main__":
print("Optimal Page Replacement Algorithm\n")
    # Input the page reference string
pages = list(map(int, input("Enter the page reference string (space separated): ").split()))
    # Input the number of frames
num_frames = int(input("Enter the number of frames: "))
    # Call the Optimal page replacement function
optimal_page_replacement(pages, num_frames)
```

Output:

Enter the page reference string (space separated): 7 0 1 2 0 3 0 4 2 3 0 3 2 3
Enter the number of frames: 4

Optimal Page Replacement Algorithm

Page 7 referenced -> Memory: [7]
Page 0 referenced -> Memory: [7, 0]
Page 1 referenced -> Memory: [7, 0, 1]
Page 2 referenced -> Memory: [7, 0, 1, 2]
Page 0 referenced -> Memory: [7, 1, 2, 0]
Page 3 referenced -> Memory: [1, 2, 0, 3]
Page 0 referenced -> Memory: [1, 2, 3, 0]
Page 4 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]
Page 0 referenced -> Memory: [4, 2, 3, 0]
Page 3 referenced -> Memory: [2, 3, 0, 4]
Page 2 referenced -> Memory: [3, 0, 4, 2]
Page 3 referenced -> Memory: [0, 4, 2, 3]

Total Page Faults: 12

Conclusion:

The above program executed successfully

13.Write a python program to implement the concept of SCAN disk scheduling.

Aim: To write a Python program that implements the **SCAN Disk Scheduling Algorithm** and calculates the **total head movement** for a given set of disk requests.

Description: The **SCAN Disk Scheduling Algorithm** (also called the *Elevator Algorithm*) works as follows:

- The disk arm moves in **one direction** servicing all requests until it reaches the **end of the disk**.
- Then it **reverses direction** and services the remaining requests.
- This approach reduces starvation and provides better performance than FCFS.

Algorithm:

- Take disk requests and initial head position.
- Decide the direction of movement (left or right).
- Sort requests based on the direction.
- Calculate total head movement while servicing requests.

Source code:

```
defscan_disk_scheduling(requests, head, disk_size, direction):
total_movement = 0
seek_sequence = []

left = []
right = []

   # Separate requests to left and right of head
for request in requests:
if request < head:
left.append(request)
elif request > head:
right.append(request)

left.sort()
right.sort()

if direction == "left":
      # Move left first
for request in reversed(left):
total_movement += abs(head - request)
head = request
seek_sequence.append(request)

      # Go to start of disk
total_movement += abs(head - 0)
head = 0

      # Move right
for request in right:
total_movement += abs(head - request)
head = request
seek_sequence.append(request)

elif direction == "right":
      # Move right first
for request in right:
total_movement += abs(head - request)
head = request
seek_sequence.append(request)

      # Go to end of disk
total_movement += abs(head - (disk_size - 1))
head = disk_size - 1

      # Move left
```

```
for request in reversed(left):
total_movement += abs(head - request)
head = request
seek_sequence.append(request)

returnseek_sequence, total_movement


# ----------- User Input Section -----------
n = int(input("Enter number of disk requests: "))
requests = list(map(int, input("Enter disk requests: ").split()))
head = int(input("Enter initial head position: "))
disk_size = int(input("Enter disk size: "))
direction = input("Enter direction (left/right): ").lower()

sequence, movement = scan_disk_scheduling(requests, head, disk_size, direction)

print("\nSeek Sequence:", sequence)
print("Total Head Movement:", movement)
```

Output:

Enter number of disk requests: 8
Enter disk requests: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Enter disk size: 200
Enter direction (right)

Seek Sequence: [65, 67, 98, 122, 124, 183, 37, 14]
Total Head Movement: 331

Conclusion:

The above program executed successfully

14. Write a python program to implement the concept of CLOOK disk scheduling.

Aim: To write a Python program to implement the C-LOOK (Circular LOOK) Disk Scheduling Algorithm

Description: In the C-LOOK disk scheduling algorithm, the disk head moves in one direction only and services all the requests in that direction until the last request is reached.
After servicing the last request, the head jumps to the first request and continues servicing the remaining requests.

Algorithm:

•Start

- Read the **number of disk requests**.
- Read the **disk request queue** in the order they arrive.
- Read the **initial head position**.
- Set`total_head_movement = 0`.
- Set`current_position = initial head position`.
- For each request in the request queue:
    - Calculate the head movement as `|request – current_position|`
    - Add this movement to `total_head_movement`.
    - Update `current_position = request`.
- Repeat step 7 until all requests are serviced.
- Display the **total head movement**.

•Stop

Source Code

```
# C-LOOK Disk Scheduling Algorithm using User Input
defclook_disk_scheduling(requests, head):
requests.sort()
total_head_movement = 0
current_position = head
print("\nDisk movement:")
   # Service requests greater than or equal to head
forreq in requests:
ifreq>= head:
print(f"{current_position} -> {req}")
total_head_movement += abs(req - current_position)
current_position = req
   # Jump to the first request
if requests[0] < head:
print(f"{current_position} -> {requests[0]}")
total_head_movement += abs(current_position - requests[0])
current_position = requests[0]
   # Service remaining requests
forreq in requests:
ifreq< head:
print(f"{current_position} -> {req}")
total_head_movement += abs(req - current_position)
current_position = req
print("\nTotal Head Movement =", total_head_movement)

# Main Program
n = int(input("Enter number of disk requests: "))
requests = []
print("Enter disk request sequence:")
for _ in range(n):
requests.append(int(input()))
head = int(input("Enter initial head position: "))
clook_disk_scheduling(requests, head)
```

Output

```
Enter number of disk requests: 5
Enter disk request sequence:
82
170
43
140
24
Enter initial head position: 50
Disk movement:
50 -> 82
82 -> 140
140 -> 170
170 -> 24
24 -> 43
Total Head Movement = 341
```

Conclusion:

The above program executed successfully

15. Write a python program to implement the concept of FCFS disk scheduling.

Aim: To write a Python program to implement the First Come First Serve (FCFS) Disk Scheduling Algorithm.

Description: FCFS (First Come First Serve) is the simplest disk scheduling algorithm.
In this algorithm, disk requests are serviced **in the exact order in which they arrive**, without any prioritization or reordering.

The disk head starts at an initial position and moves sequentially to each requested track.
The **total head movement** is calculated as the sum of the absolute differences between consecutive track positions.

Algorithm:

- Start
- Read the number of disk requests `n`.
- Read the disk request queue in the order of arrival.
- Read the initial head position.
- Initialize`total_head_movement = 0`.
- Set`current_position = initial head position`.
- For each request in the queue:

a. Calculate head movement = `|request - current_position|`.
b. Add the movement to `total_head_movement`.
c. Update `current_position = request`.

- Display the total head movement.
- Stop.

Source Code:

```
deffcfs_disk_scheduling(requests, head):
total_head_movement = 0
current_position = head
print("\nDisk movement:")
for request in requests:
print(f"{current_position} -> {request}")
total_head_movement += abs(request - current_position)
current_position = request
print("\nTotal Head Movement =", total_head_movement)
# Main Program
n = int(input("Enter number of disk requests: "))
requests = []
print("Enter disk request sequence:")
for i in range(n):
requests.append(int(input()))

head = int(input("Enter initial head position: "))
fcfs_disk_scheduling(requests, head)
```

Output:

```
Enter number of disk requests: 5
Enter disk request sequence:
82
170
43
140
24
Enter initial head position: 50
Disk movement:
50 -> 82
82 -> 170
170 -> 43
43 -> 140
140 -> 24
Total Head Movement = 642
```

Conclusion:

The above program executed successfully