OS CA-2

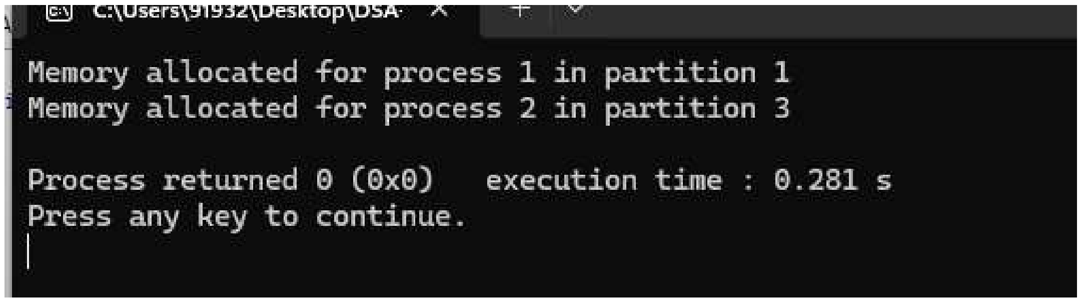
Name - Deepa Behrani
Roll no - 7
Division - D10B

Q1) To write a C program for implementation memory allocation methods for fixed partition using first fit.

Code:

```
#include <stdio.h>
#include <stdbool.h>

#define NUM_PARTITIONS 5

// Structure to represent a partition
typedef struct {
    int size;           // Partition size
    bool allocated;     // Flag to indicate if the partition is allocated
} Partition;

// Function to allocate memory using First Fit strategy
int allocateMemoryFirstFit(Partition partitions[], int size) {
    for (int i = 0; i < NUM_PARTITIONS; i++) {
        if (!partitions[i].allocated && partitions[i].size >= size) {
            partitions[i].allocated = true;
            return i;
        }
    }
    return -1; // Memory allocation failed
}

int main() {
    Partition partitions[NUM_PARTITIONS] = {{100, false}, {200, false}, {150, false}, {300,
false}, {250, false}};
    int allocatedPartitionId;

    // Memory allocation requests
    allocatedPartitionId = allocateMemoryFirstFit(partitions, 180);
    if (allocatedPartitionId != -1) {
        printf("Memory allocated for process 1 in partition %d\n", allocatedPartitionId);
    } else {
        printf("Memory allocation failed for process 1\n");
    }

    allocatedPartitionId = allocateMemoryFirstFit(partitions, 250);
    if (allocatedPartitionId != -1) {
        printf("Memory allocated for process 2 in partition %d\n", allocatedPartitionId);
    } else {
        printf("Memory allocation failed for process 2\n");
    }

    return 0;
}
```

OUTPUT:



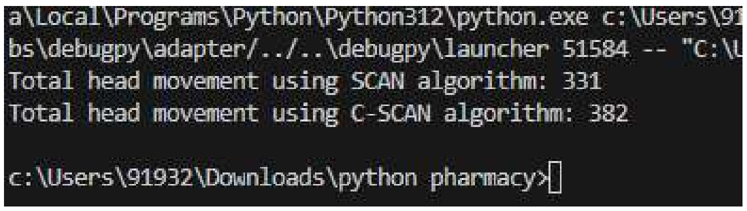
```
C:\Users\91932\Desktop\DSA·

Memory allocated for process 1 in partition 1
Memory allocated for process 2 in partition 3

Process returned 0 (0x0)   execution time : 0.281 s
Press any key to continue.
```

Q2) Implement various disk scheduling algorithms like SCAN, and C-SCAN in C/Python/Java

Code:

```
def SCAN(requests, start, direction, max_track):
    total_head_movement = 0
    current_track = start

    # Sort the requests based on their track numbers
    sorted_requests = sorted(requests)

    # Determine the direction of head movement
    if direction == "left":
        head_movement_direction = -1
    else:
        head_movement_direction = 1

    # Move towards the end of the disk
    while sorted_requests:
        next_track = None
        for request in sorted_requests:
            if current_track <= request <= max_track and request > current_track:
                next_track = request
                break
        if next_track is not None:
            total_head_movement += abs(next_track - current_track)
            current_track = next_track
            sorted_requests.remove(next_track)
        else:
            total_head_movement += abs(max_track - current_track)
            current_track = max_track
            break

    # Move towards the beginning of the disk
    while sorted_requests:
        next_track = None
        for request in sorted_requests:
            if 0 <= request < current_track:
                next_track = request
                break

        if next_track is not None:
            total_head_movement += abs(next_track - current_track)
            current_track = next_track
            sorted_requests.remove(next_track)
        else:
            break

    return total_head_movement


def CSCAN(requests, start, max_track):
    total_head_movement = 0
    current_track = start

    # Sort the requests based on their track numbers
    sorted_requests = sorted(requests)

    # Move towards the end of the disk
    while sorted_requests:
        next_track = None
        for request in sorted_requests:
            if current_track <= request <= max_track and request > current_track:
                next_track = request
                break

        if next_track is not None:
            total_head_movement += abs(next_track - current_track)
            current_track = next_track
            sorted_requests.remove(next_track)
        else:
            total_head_movement += abs(max_track - current_track)
            current_track = max_track
            break

    # Move towards the beginning of the disk (wrap around)
    total_head_movement += abs(0 - current_track)
    current_track = 0

    # Continue moving towards the end of the disk
    while sorted_requests:
        next_track = sorted_requests.pop(0)
        total_head_movement += abs(next_track - current_track)
        current_track = next_track

    return total_head_movement


# Example usage:

requests = [98, 183, 37, 122, 14, 124, 65, 67]  # Sample disk requests
start = 53  # Starting track
max_track = 199  # Maximum track number

# SCAN algorithm
total_movement_scan = SCAN(requests, start, direction="left", max_track=max_track)
print("Total head movement using SCAN algorithm:", total_movement_scan)

# C-SCAN algorithm
total_movement_cscan = CSCAN(requests, start, max_track=max_track)
print("Total head movement using C-SCAN algorithm:", total_movement_cscan)
```

OUTPUT :



```
a\Local\Programs\Python\Python312\python.exe c:\Users\91
bs\debugpy\adapter/../..\debugpy\launcher 51584 -- "C:\U
Total head movement using SCAN algorithm: 331
Total head movement using C-SCAN algorithm: 382

c:\Users\91932\Downloads\python pharmacy>
```

Q3) Case Study on Distributed Operating System.

Ans:

*Case Study: Edge Computing in Distributed Operating Systems

Introduction:

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed, improving response times and saving bandwidth. In the context of a distributed operating system, edge computing plays a crucial role in enabling efficient and scalable processing of data and applications across a distributed network of devices.

Scenario:
Consider a smart city deployment where various IoT devices are deployed across different locations to monitor and manage urban infrastructure such as traffic, energy usage, and environmental conditions. These IoT devices generate a vast amount of data that needs to be processed in real-time to enable timely decision-making and resource optimization.

Challenges:
- Latency: Traditional cloud computing solutions may introduce significant latency due to the distance between the data source (IoT devices) and the cloud data centre.
- Bandwidth Constraints: Transmitting large volumes of data to the cloud can strain network bandwidth and incur high data transfer costs**.**
- Reliability: Dependence on a centralised cloud infrastructure introduces a single point of failure and may result in service disruptions.

Solution:
A distributed operating system incorporating edge computing principles can address these challenges effectively. Here's how:

1. Edge Node Deployment: Deploy edge nodes (e.g., edge servers, gateways) closer to the IoT devices within the smart city. These edge nodes act as intermediate computing and storage hubs, processing data locally before transmitting relevant insights to the cloud.

2.Distributed Data Processing: Implement distributed data processing algorithms and middleware on edge nodes to perform real-time analytics and decision-making at the edge. This reduces the need to transmit raw data to the cloud, thereby minimising latency and bandwidth usage.

3. Decentralised Control: Distribute control and management functions across edge nodes to ensure fault tolerance and scalability. Each edge node operates autonomously while coordinating with neighbouring nodes to optimise resource utilisation and workload distribution.

4. Dynamic Resource Allocation: Utilise dynamic resource allocation techniques to efficiently allocate computational resources based on workload demands and priorities. This ensures optimal performance and responsiveness across the distributed system.

5. Edge-Cloud Integration: Establish seamless integration between edge nodes and the cloud infrastructure to facilitate data sharing, synchronisation, and offloading of computationally intensive tasks to the cloud when necessary. This hybrid approach leverages the strengths of both edge and cloud computing paradigms.

Benefits:

- Low Latency: Edge computing reduces data processing and response times by processing data closer to its source, enabling real-time decision-making and faster insights.
- Bandwidth: Efficiency: By filtering and aggregating data at the edge, bandwidth usage and data transfer costs are minimised, particularly for large-scale IoT deployments.
- Scalability and Reliability: The distributed nature of edge computing enhances system scalability and fault tolerance, ensuring high availability and resilience to network disruptions.

Conclusion:

Incorporating edge computing principles into a distributed operating system architecture offers a compelling solution for addressing the unique challenges of processing and managing data in distributed IoT environments, such as smart cities. By leveraging edge nodes strategically deployed throughout the network, organizations can achieve superior performance, efficiency, and reliability in their operations while unlocking new opportunities for innovation and value creation.