Operating System Lab: Scheduling Algorithm

Question 1: First Come First Serve Process Scheduling

The FCFS algorithm schedules processes in the order they arrive, with no preemption. The process that arrives first is executed first.

Python Code:

```
class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.completion_time = 0
        self.waiting_time = 0
        self.turnaround_time = 0

def fcfs_scheduling(processes):
    processes.sort(key=lambda x: x.arrival_time)

    current_time = 0
    for process in processes:
        if current_time < process.arrival_time:
            current_time = process.arrival_time

        process.completion_time = current_time + process.burst_time
        process.turnaround_time = process.completion_time - process.arrival_time
        process.waiting_time = process.turnaround_time - process.burst_time
        current_time = process.completion_time
```

```
    total_waiting_time = sum(p.waiting_time for p in processes)
    total_turnaround_time = sum(p.turnaround_time for p in processes)
    print("\nFCFS Scheduling Results:")
    for process in processes:
        print(f"Process {process.pid}: Waiting Time: {process.waiting_time}, Turnaround Time: {process.turnaround_time}")

    print(f"Average Waiting Time: {total_waiting_time / len(processes):.2f}")
    print(f"Average Turnaround Time: {total_turnaround_time / len(processes):.2f}")

num_processes = int(input("Enter the number of processes: "))
burst_times = list(map(int, input("Enter the CPU times: ").split()))
arrival_times = list(map(int, input("Enter the arrival times: ").split()))

processes = [Process(i + 1, arrival_times[i], burst_times[i]) for i in range(num_processes)]
fcfs_scheduling(processes)
```

Question 2: Job First (Non-Preemptive) Scheduling

In non-preemptive SJF, the process with the shortest burst time is executed next. The CPU is not preempted, and processes are executed until completion.

Python Code:

```
def sjf_non_preemptive_scheduling(processes):
    processes.sort(key=lambda x: (x.arrival_time, x.burst_time))
    current_time = 0

    for process in processes:
        if current_time < process.arrival_time:
```

```
            current_time = process.arrival_time
        process.completion_time = current_time + process.burst_time
        process.turnaround_time = process.completion_time - process.arrival_time
        process.waiting_time = process.turnaround_time - process.burst_time
        current_time = process.completion_time

    total_waiting_time = sum(p.waiting_time for p in processes)
    total_turnaround_time = sum(p.turnaround_time for p in processes)
    print("\nSJF Non-Preemptive Scheduling Results:")
    for process in processes:
        print(f"Process {process.pid}: Waiting Time: {process.waiting_time}, Turnaround Time: {process.turnaround_time}")

    print(f"Average Waiting Time: {total_waiting_time / len(processes):.2f}")
    print(f"Average Turnaround Time: {total_turnaround_time / len(processes):.2f}")
sjf_non_preemptive_scheduling(processes)
```

Question 3: Shortest Job First (Preemptive) Scheduling

In preemptive SJF (also called Shortest Remaining Time First), the scheduler always picks the process with the smallest remaining burst time.

Python Code:

```
def sjf_preemptive_scheduling(processes):
    n = len(processes)
    completed = 0
    current_time = 0
    while completed < n:
        # Select the process with the smallest remaining time that has arrived
        available_processes = [p for p in processes if p.arrival_time <= current_time and p.burst_time > 0]
        if not available_processes:
```

```
            current_time += 1
            continue

        available_processes.sort(key=lambda x: x.burst_time)
        current_process = available_processes[0]

        current_process.burst_time -= 1
        current_time += 1

        if current_process.burst_time == 0:
            completed += 1
            current_process.completion_time = current_time
            current_process.turnaround_time = current_process.completion_time -
current_process.arrival_time
            current_process.waiting_time = current_process.turnaround_time - (current_process.burst_time
+ 1)

    total_waiting_time = sum(p.waiting_time for p in processes)
    total_turnaround_time = sum(p.turnaround_time for p in processes)
    print("\nSJF Preemptive Scheduling Results:")
    for process in processes:
        print(f"Process {process.pid}: Waiting Time: {process.waiting_time}, Turnaround Time:
{process.turnaround_time}")

    print(f"Average Waiting Time: {total_waiting_time / len(processes):.2f}")
    print(f"Average Turnaround Time: {total_turnaround_time / len(processes):.2f}")

processes = [Process(i + 1, arrival_times[i], burst_times[i]) for i in range(num_processes)]
sjf_preemptive_scheduling(processes)
```

Question 4: Analysis and Findings

1. **FCFS**: It's straightforward and fair but suffers from the "convoy effect," where a long process can delay others, leading to high average waiting times.
2. **Non-Preemptive SJF**: This is more efficient in reducing waiting time but suffers if a shorter job arrives after a longer one starts execution. It can lead to **starvation** if short processes keep arriving, delaying longer ones indefinitely.
3. **Preemptive SJF**: This offers the lowest possible waiting time and reduces the likelihood of starvation by immediately addressing new short jobs. However, it's more complex to implement, requiring more frequent context switching, which can add overhead.

Question 5 : Challenges Faced

1. **Context Switching in SJF Preemptive**: Managing the continuous switching between processes based on burst times led to complexities in managing arrival times.
2. **Dynamic Sorting**: Sorting processes in SJF and updating frequently made the implementation of the preemptive version a bit challenging, especially when processes arrived midway during the scheduling.