Vansh Jalora
231070074
SY CE BTech
Batch-D

Operating Systems — Lab 3

AIM:

To implement all 4 Job Scheduling Algorithms:

- First Come First Serve Scheduling
- Shortest Job First Scheduling
- Round Robin Scheduling
- Priority Scheduling

Theory

First Come First Serve Scheduling

- In FCFS scheduling, the CPU is allocated to the process that enters the ready queue first. If two processes arrive at the same time, the one with the smaller process ID is executed first.
- This scheduling is always non-preemptive, meaning a process, once started, cannot be interrupted until completion.
- Processes are handled in the order of their arrival, making it a simple, easy-to-implement algorithm based on a FIFO (First In, First Out) queue. However, its performance is generally poor due to the high average waiting time.

Advantages:

- Simple and easy to understand.
- Easily implemented using a queue data structure.
- Prevents starvation, as every process gets its turn.

Disadvantages:

- Does not take into account process priority or burst time.
- Suffers from the convoy effect, where processes with shorter burst times are delayed by longer ones that arrive first.

Shortest Job First Scheduling

- In SJF scheduling, the CPU is assigned to the process with the shortest burst time (execution time) first. If two processes have the same burst time, the one with the smaller process ID is chosen.
- This scheduling can be either preemptive or non-preemptive. Preemptive SJF, also called Shortest Remaining Time First (SRTF), allows a process to be interrupted if a new process with a shorter burst time arrives.
- SJF is known for minimizing the average waiting time, making it more efficient than FCFS, but it can be more complex to implement due to the need for burst time estimation.

Advantages:

- Minimizes average waiting time.
- More efficient than FCFS, especially in reducing turnaround time.

Disadvantages:

- Difficult to implement as it requires accurate knowledge or estimation of burst times.
- Can lead to starvation, as longer processes might get delayed indefinitely if shorter ones keep arriving.
- Not suitable for interactive systems, where process execution times are unpredictable.

Round Robin Scheduling

- In Round Robin scheduling, each process is assigned a fixed time slice or quantum during which it can execute.
- After its time slice expires, the process is moved to the back of the ready queue, and the next process gets the CPU. This preemptive scheduling ensures that all processes are treated fairly and that no process dominates the CPU.
- The time quantum plays a critical role in performance: a small quantum leads to more context switching, while a large quantum makes it resemble FCFS. It is widely used in time-sharing systems.

Advantages:

- Ensures fairness by giving each process an equal share of CPU time.
- Suitable for time-sharing and interactive systems, as it prevents any single process from monopolizing the CPU.
- Reduces the risk of starvation.

Disadvantages:

- Performance depends on the choice of the time quantum; too small a quantum increases overhead due to frequent context switching, while too large a quantum can increase average waiting time.
- Not optimal for processes with varying burst times, as it doesn't prioritize shorter tasks.
- Higher overhead due to frequent context switching compared to non-preemptive algorithms.

Priority Scheduling

- In Priority Scheduling, each process is assigned a priority, and the CPU is allocated to the process with the highest priority (smallest numerical value).
- If two processes have the same priority, other criteria like arrival time may be used to break the tie.
- This scheduling can be either preemptive, where a running process can be interrupted by a higher-priority process, or non-preemptive, where the CPU is only allocated to the highest-priority process after the current one completes.
- Priority Scheduling ensures that important tasks are processed first but can lead to issues like starvation.

Advantages:

- Allows prioritization of critical tasks, improving responsiveness for high-priority processes.
- Efficient in handling processes with varying levels of urgency.
- Can be fine-tuned by dynamically adjusting process priorities.

Disadvantages:

- Can lead to starvation, where low-priority processes may never get executed if higher-priority tasks keep arriving (this can be mitigated using techniques like aging, where the priority of a process increases the longer it waits).
- Requires careful priority assignment, which can be complex in systems with many processes.
- Not ideal for environments where all processes are equally important.

Algorithms

First Come First Serve

```
FUNCTION fcfs(processes):
    initialize an empty queue
    initialize wait_times and turnaround_times as empty lists
    set current_time to 0

    // Add all processes to the queue
    for each process in processes:
        enqueue process

    while queue is not empty:
        process = dequeue queue
        process_id = process.first
        burst_time = process.second

        add current_time to wait_times
        add (current_time + burst_time) to turnaround_times
        current_time += burst_time

    total_wait_time = sum of wait_times
    average_wait_time = total_wait_time / number of processes

    total_turnaround_time = sum of turnaround_times
    average_turnaround_time = total_turnaround_time / number of processes
```

Shortest Job First

```
FUNCTION sjf_algorithm(processes):
    initialize an empty priority queue (min-heap)
    initialize wait_times and turnaround_times as empty lists
    set current_time to 0

    // Add all processes to the priority queue (sorted by burst time)
    for each process in processes:
        push process to priority queue

    while priority queue is not empty:
        process = pop from priority queue
        process_id = process.first
        burst_time = process.second

        add current_time to wait_times
        add (current_time + burst_time) to turnaround_times
        current_time += burst_time


    total_wait_time = sum of wait_times
    average_wait_time = total_wait_time / number of processes

    total_turnaround_time = sum of turnaround_times
    average_turnaround_time = total_turnaround_time / number of processes
```

Round Robin Scheduling

```
FUNCTION round_robin(processes, time_slice):
    initialize an empty queue
    initialize remaining_burst_time, wait_times, and turnaround_times as
lists
    set total_time to 0

    // Add all processes to the queue and set initial burst times
    for each process in processes:
        enqueue process
        remaining_burst_time[process.id] = process.burst_time
```

```
    while queue is not empty:
        process = dequeue queue
        process_id = process.first
        burst_time = remaining_burst_time[process_id]

        if burst_time <= time_slice then
            total_time += burst_time
            turnaround_times[process_id] = total_time

        else
            total_time += time_slice
            remaining_burst_time[process_id] -= time_slice
            enqueue(process_id, remaining_burst_time[process_id])

        // Update wait times for all other processes
        for each process in processes:
            if remaining_burst_time[process.id] > 0 and process.id !=
process_id:
                wait_times[process.id] += min(time_slice, burst_time)

    total_wait_time = sum of wait_times
    average_wait_time = total_wait_time / number of processes

    total_turnaround_time = sum of turnaround_times
    average_turnaround_time = total_turnaround_time / number of processes
```

Priority Scheduling

```
FUNCTION priority_scheduling(processes):
    initialize an empty priority queue (min-heap)
    initialize wait_times and turnaround_times as empty lists
    set current_time to 0

    // Add all processes to the priority queue (sorted by priority)
    for each process in processes:
        push process to priority queue

    while priority queue is not empty:
        process = pop from priority queue
        process_id = process.first
        burst_time = process.second
```

```
        add current_time to wait_times
        add (current_time + burst_time) to turnaround_times
        current_time += burst_time

    total_wait_time = sum of wait_times
    average_wait_time = total_wait_time / number of processes


    total_turnaround_time = sum of turnaround_times
    average_turnaround_time = total_turnaround_time / number of processes
```

COMPLETE PROGRAM

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>
#include <numeric>
#include <tuple>
using namespace std;

struct CompareByBurstTime {
   // For SJF scheduling
    bool operator()(const pair<int, int>& p1, const pair<int, int>& p2) {
        return p1.second > p2.second;
    }
};

struct CompareByPriority {
   // For priority scheduling
    bool operator()(const tuple<int, int, int>& p1, const tuple<int, int,
int>& p2) {
        return get<2>(p1) > get<2>(p2);
    }
};
```

```
void fcfs_algorithm(vector<pair<int, int>> processes) {
      queue<pair<int, int>> q;
      vector<int> wait_times, turnaround_times;
      int current_time = 0;

      for (int i = 0; i < processes.size(); i++) q.push(processes[i]);

      while (!q.empty()) {
            pair<int, int> process = q.front();
            q.pop();

            int process_id = process.first;
            int burst_time = process.second;

            wait_times.push_back(current_time);
            turnaround_times.push_back(current_time + burst_time);
            current_time += burst_time;

            cout << "Process " << process_id << " completed." << endl;
      }

      int total_wait_time = accumulate(wait_times.begin(),
wait_times.end(), 0);
      double average_wait_time = static_cast<double>(total_wait_time) /
wait_times.size();
      cout << "Average wait time: " << average_wait_time << endl;

      int total_turnaround_time = accumulate(turnaround_times.begin(),
turnaround_times.end(), 0);
      double average_turnaround_time =
static_cast<double>(total_turnaround_time) / turnaround_times.size();
      cout << "Average turnaround time: " << average_turnaround_time <<
endl << endl;
}

void sjf_algorithm(vector<pair<int, int>> process) {
      priority_queue< pair<int, int>, vector<pair<int, int>>,
CompareByBurstTime > pq;
      vector<int> wait_times, turnaround_times;
      int current_time = 0;

      for (const auto& p : process) pq.push(p);
```

```
      while (!pq.empty()) {
            pair<int, int> process = pq.top();
            pq.pop();

            int process_id = process.first;
            int burst_time = process.second;

            wait_times.push_back(current_time);
            turnaround_times.push_back(current_time + burst_time);
            current_time += burst_time;

            cout << "Process " << process_id << " completed." << endl;
      }

      int total_wait_time = accumulate(wait_times.begin(),
wait_times.end(), 0);
      double average_wait_time = static_cast<double>(total_wait_time) /
wait_times.size();
      cout << "Average wait time: " << average_wait_time << endl;

      int total_turnaround_time = accumulate(turnaround_times.begin(),
turnaround_times.end(), 0);
      double average_turnaround_time =
static_cast<double>(total_turnaround_time) / turnaround_times.size();
      cout << "Average turnaround time: " << average_turnaround_time <<
endl << endl;
}


void round_robin(vector<pair<int, int>> processes, int time_slice) {
    queue<pair<int, int>> q;
    vector<int> remaining_burst_time(processes.size()),
                wait_times(processes.size(), 0),
                turnaround_times(processes.size());
    int total_time = 0;

    // Keep track of original index to match process_id and array indices
    for (int i = 0; i < processes.size(); i++) {
        q.push({i, processes[i].second});  // Enqueue index and burst time
        remaining_burst_time[i] = processes[i].second;
    }
```

```
    while (!q.empty()) {
        pair<int, int> p = q.front(); q.pop();
        int index = p.first;                // Index of process in the
original array
        int burst_time = remaining_burst_time[index];

        if (burst_time <= time_slice) {
            total_time += burst_time;
            turnaround_times[index] = total_time;
            cout << "Process " << processes[index].first << " completed at
time " << total_time << endl;
        } else {
            total_time += time_slice;
            remaining_burst_time[index] -= time_slice;
            q.push({index, remaining_burst_time[index]});
        }

        for (int i = 0; i < processes.size(); i++) {
            if (remaining_burst_time[i] > 0 && i != index) {
                wait_times[i] += min(time_slice, burst_time);
            }
        }
    }

    int total_wait_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < processes.size(); i++) {
        total_wait_time += wait_times[i];
        total_turnaround_time += turnaround_times[i];
    }

    double average_wait_time = static_cast<double>(total_wait_time) /
processes.size();
    double average_turnaround_time =
static_cast<double>(total_turnaround_time) / processes.size();

    cout << "Average wait time: " << average_wait_time << endl;
    cout << "Average turnaround time: " << average_turnaround_time << endl
<< endl;
}


void priority_scheduling(vector<tuple<int, int, int>> process) {
    priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>,
```

```
CompareByPriority> pq;
    vector<int> wait_times, turnaround_times;
    int current_time = 0;

    for (const auto& p : process) pq.push(p);

    while (!pq.empty()) {
        tuple<int, int, int> p = pq.top();
        pq.pop();

        int process_id = get<0>(p);
        int burst_time = get<1>(p);

        wait_times.push_back(current_time);
        turnaround_times.push_back(current_time + burst_time);
        current_time += burst_time;

        cout << "Process " << process_id << " completed." << endl;
    }

    int total_wait_time = accumulate(wait_times.begin(), wait_times.end(),
0);
    double average_wait_time = static_cast<double>(total_wait_time) /
wait_times.size();
    cout << "Average wait time: " << average_wait_time << endl;

    int total_turnaround_time = accumulate(turnaround_times.begin(),
turnaround_times.end(), 0);
    double average_turnaround_time =
static_cast<double>(total_turnaround_time) / turnaround_times.size();
    cout << "Average turnaround time: " << average_turnaround_time << endl;
}


int main() {
   vector<pair<int, int>> v;
   v.push_back(make_pair(1, 4));
   v.push_back(make_pair(2, 3));
   v.push_back(make_pair(3, 2));
   v.push_back(make_pair(4, 7));
   v.push_back(make_pair(5, 6));

   cout<<"FCFS Algorithm:"<<endl;
```

```
    fcfs_algorithm(v);

    cout<<"SJF Algorithm:"<<endl;
    sjf_algorithm(v);

    cout<<"Round Robin algorithm:"<<endl;
    round_robin(v, 2);

    cout<<"Priority algorithm:"<<endl;
    vector<tuple<int, int, int>> processes = { {1, 6, 2}, {2, 8, 1}, {3, 7,
3}, {4, 3, 2} };
    priority_scheduling(processes);
    return 0;
}
```

OUTPUT

FCFS

```
c:\Users\Vansh_Prac\SY_Prac\Python\OS-La
omplete && "c:\Users\Vansh_Prac\SY_Prac\
FCFS Algorithm:
Process 1 completed.
Process 2 completed.
Process 3 completed.
Process 4 completed.
Process 5 completed.
Average wait time: 6.2
Average turnaround time: 10.2
```

SJF

```
SJF Algorithm:
Process 1 completed.
Process 2 completed.
Process 4 completed.
Process 3 completed.
Process 5 completed.
Average wait time: 6
Average turnaround time: 10
```

Round Robin

```
Round Robin algorithm:
Process 1 completed at time 2
Process 2 completed at time 11
Process 4 completed at time 15
Process 3 completed at time 18
Process 5 completed at time 20
Average wait time: 16
Average turnaround time: 13.2
```

Priority

```
Priority algorithm:
Process 2 completed.
Process 1 completed.
Process 4 completed.
Process 3 completed.
Average wait time: 9.75
Average turnaround time: 15.75

c:\Users\Vansh_Prac\SY_Prac\Python\OS-Lab\Lab3>
```

Conclusion:

We implemented four essential CPU scheduling algorithms: *FCFS (First-Come, First-Served), SJF (Shortest Job First), Round Robin, and Priority Scheduling.* These algorithms are foundational in operating system design and manage how processes are allocated CPU time, optimizing performance based on different criteria. From a practical standpoint, we observed how these algorithms manage process execution, influencing system efficiency and user experience. Selecting the appropriate algorithm depends on the system's goals, such as minimizing wait time or ensuring fairness.

- *FCFS* is simple but can lead to high waiting times due to its non-preemptive nature.
- *SJF* minimizes waiting time but requires precise knowledge of burst times.
- *Round Robin* ensures fairness by assigning time slices, making it ideal for time-sharing systems.
- *Priority Scheduling* prioritizes processes but can cause starvation without proper handling.