CPU Scheduler:

It is a system software component in an operating system that is responsible for managing process execution by CPU. Its function is to decide which process in the ready queue should be executed by CPU, allocation of that process to CPU acc. to specific algorithm. It also ensures that processes get appropriate CPU time for execution, especially in time shared systems.

CPU Scheduling is very important because,

- By scheduling, we can ensure the CPU has minimal idle time and being maximum utilized.
- It is used to increase the number of processes completed per unit time. Higher the throughput, the more the system can perform/handle the tasks.
- Scheduling ensures quick responses to the user inputs, enhancing the user experience by reducing the response time.
- It enables multiple processes to run concurrently, making multitasking possible.

CPU scheduling decisions only being taken under the following 4 circumstances:

- When a process switches from running state to waiting state. [Non Preemptive Scheduling]
- When a process switches from running state to ready state. [Preemptive Scheduling] – for e.g., when an interrupt occurs.
- When a process switches from waiting state to ready state. [Preemptive Scheduling] – for e.g., at completion of I/O.
- When a process terminates.

Scheduling Criteria:

Scheduling criteria is crucial for selecting or designing an appropriate scheduling algorithm that meets the specific needs of a computing environment.

- Maximize CPU utilization to ensure the CPU is not idle.
- Maximize throughput to handle more processes in less time.
- Minimize turnaround time [i.e. time taken from the submission of a process to its completion] to ensure processes are completed quickly.
- Minimize waiting time [i.e. total time a process spends waiting in the ready queue] to reduce delays in process execution.

- Minimize response time [i.e. time from the submission of a request until the first response is produced] to improve the interactivity of the system, especially important for time-sharing systems.

Scheduling Algorithms:

1. First Come First Served Scheduling:

   - Simplest Scheduling Algorithm
   - In this, the CPU will be allocated to the process that requests the CPU first. The CPU picks the process at the front of the ready queue and executes it until completion or it moves to the waiting state.
   - Non Preemptive type algorithm i.e., once a process starts executing, it runs to completion without being preempted.

   Implementation:

```
void FCFS_scheduling(vector<Task> &tasks, const int &num_tasks)
{
   // Sort tasks by arrival time
   sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b)
       { return a.arrival_time < b.arrival_time; });
```

This line sorts the tasks vector based on the arrival times of the tasks in ascending order. This ensures that tasks are executed in the order they arrive.

```
   int execution_lower[num_tasks], execution_higher[num_tasks],
waiting[num_tasks], turnaround[num_tasks];
```

Arrays are initialized to store the start and end times of execution (execution_lower and execution_higher), waiting times (waiting), and turnaround times (turnaround) for each task.

```
   cout << "\nFCFS Scheduling:\n\n";
   int current_time = 0;
   int total_turnaround_time = 0;
```

```
    int total_waiting_time = 0;

    for (const Task &task : tasks)
    {
        // Wait if the task hasn't arrived yet
        if (current_time < task.arrival_time)
        {
            current_time = task.arrival_time;
        }

        // Execute the task
        execution_lower[task.id - 1] = current_time;
        current_time += task.burst_time;
        execution_higher[task.id - 1] = current_time;
        turnaround[task.id - 1] = current_time - task.arrival_time;
        waiting[task.id - 1] = turnaround[task.id - 1] - task.burst_time;
        total_turnaround_time += turnaround[task.id - 1];
        total_waiting_time += waiting[task.id - 1];
    }
```

The current time is tracked with current_time.
For each task:

- *If the current time is less than the task's arrival time, the current time is updated to the task's arrival time (simulating waiting for the task to arrive).*
- *The task is then executed, with execution_lower marking the start and execution_higher marking the end of execution.*
- *Turnaround time (total time from arrival to completion) and waiting time (time spent waiting before execution) are calculated and stored.*
- *Total turnaround and waiting times are accumulated.*

```
    sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b)
         { return a.id < b.id; });
```

After processing, tasks are sorted back by their ID to display the results in the order of task IDs.

```
    for (const Task &task : tasks)
    {
        cout << "Task " << task.id << " executed during time " <<
execution_lower[task.id - 1];
```

```
        cout << " to " << execution_higher[task.id - 1] << " unit" << endl;
        cout << "Turn Around Time for Task " << task.id << " is " <<
turnaround[task.id - 1] << " unit" << endl;
        cout << "Waiting Time for Task " << task.id << " is " << waiting[task.id - 1] <<
" unit\n"
             << endl;
    }

    double average_waiting_time = static_cast<double>(total_waiting_time) /
num_tasks;
    double average_turnaround_time =
static_cast<double>(total_turnaround_time) / num_tasks;

    cout << "Average Turn Around Time: " << average_turnaround_time << " unit"
<< endl;
    cout << "Average Waiting Time: " << average_waiting_time << " unit" << endl;
}
```

The function prints the execution times, turnaround times, and waiting times for each task. It also calculates and prints the average turnaround and waiting times.

This function implements the FCFS scheduling algorithm by first sorting the tasks by their arrival times. It then processes each task in sequence, calculating the execution times, waiting times, and turnaround times. Finally, it outputs these values along with the average turnaround and waiting times for all tasks. The tasks are re-sorted by their IDs for organized output display.

One of the advantage of using this algorithm is its simplicity i.e., it is easy to understand and implement.

Limitations:

- Convoy Effect: Short processes may have to wait for long processes to complete, leading to the "convoy effect," where shorter tasks are delayed excessively.
- The average waiting under the FCFS algorithm is quite long which makes it troublesome to use in a time-shared system.

FCFS is ideal for environments that prioritize simplicity and predictability over performance optimization.

2. <u>Shortest Job First Scheduling</u>:

    - It selects the process with the smallest amount of time remaining until completion. If a new process arrives with a shorter remaining time than the current process, the current process is interrupted and the new process is executed.
    - Preemptive/Non-Preemptive type algorithm. I had consider it to be Preemptive i.e., processes can be interrupted and moved to the ready queue if a new process arrives with a shorter remaining time.

<u>Implementation</u>:

```
void SJF_scheduling(vector<Task> &tasks, const int &num_tasks) {
   int bursttime[num_tasks];
   for (int i = 0; i < num_tasks; i++)
   {
      bursttime[tasks[i].id - 1] = tasks[i].burst_time;
   }
```

A bursttime array is initialized to store the original burst times of each task.

```
   // Sort tasks primarily by arrival time, secondarily by burst time
   sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) {
      if (a.arrival_time == b.arrival_time) {
         return a.burst_time < b.burst_time;
      }
      return a.arrival_time < b.arrival_time;
   });
```

Tasks are sorted primarily by their arrival time and secondarily by their burst time (if arrival times are the same). This ensures that when multiple tasks arrive at the same time, the task with the shorter burst time is considered first.

```
vector<vector<int>> execution(num_tasks);
 cout << "\nSJF Scheduling:\n"<<endl;
 int current_time = 0;
 int total_turnaround_time = 0;
 int total_waiting_time = 0;
 int tasks_completed = 0;
```

execution: A 2D vector to store the execution timeline for each task.
current_time: Keeps track of the current time in the schedule.
total_turnaround_time and total_waiting_time: Accumulators for calculating the average times.
tasks_completed: Counter to track the number of completed tasks.

```
 while (tasks_completed < num_tasks) {
     int shortest_job_index = -1;
     for (int i = 0; i < num_tasks; ++i) {
         if (tasks[i].arrival_time <= current_time && tasks[i].burst_time > 0) {
        if (shortest_job_index == -1 || tasks[i].burst_time <
        tasks[shortest_job_index].burst_time) {
                shortest_job_index = i;
             }
         }
     }

     if (shortest_job_index == -1) {
        // No task is ready to execute, move to the next available task's
        arrival time
         current_time++;
         continue;
     }

     Task &task = tasks[shortest_job_index];
     task.burst_time--;
     execution[task.id - 1].push_back(current_time);

     if (task.burst_time == 0) {
         tasks_completed++;
     }
```

```
        current_time++;
    }
```

- *The main loop runs until all tasks are completed.*
- *It finds the shortest job that is ready to execute by iterating through tasks and selecting the one with the shortest burst time that has arrived (arrival_time <= current_time).*
- *If no task is ready, it increments the current_time.*
- *If a task is selected, it decrements its burst time, logs its execution time, and if it completes (burst_time == 0), increments the tasks_completed counter.*

```
// Sort tasks back by their IDs for output consistency
        sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) {
        return a.id < b.id; });
```

Tasks are sorted back by their original IDs to ensure the output is consistent and easy to read.

```
    // Print the execution times for each task
    for (const Task &task : tasks) {
        cout << "Task " << task.id << " executed during time ";
        if (execution[task.id - 1].empty()) {
            cout << "no time";
        } else {
            int start = execution[task.id - 1][0], end = execution[task.id - 1][0];
            for (size_t i = 1; i < execution[task.id - 1].size(); ++i) {
                if (execution[task.id - 1][i] == execution[task.id - 1][i - 1] + 1) {
                    end = execution[task.id - 1][i];
                } else {
                    if (start == end) {
                        cout << start<<" to "<<start+1;
                    } else {
                        cout << start << " to " << end+1;
                    }
                    cout << ", ";
                    start = end = execution[task.id - 1][i];
                }
            }
            int completion_time=0;
```

```
          if (start == end) {
             cout << start<<" to "<<start+1<<" unit";
             completion_time=start+1;
          } else {
             cout << start << " to " << end+1<<" unit";
             completion_time=end+1;
          }
          int turnaroundtime=completion_time-task.arrival_time;
          int waitingtime=turnaroundtime-bursttime[task.id-1];
         cout << "\nTurn Around Time for Task " << task.id << " is " <<
         turnaroundtime << " unit" << endl;
         cout << "Waiting Time for Task " << task.id << " is " << waitingtime
         << " unit"
           << endl;
          total_waiting_time+=waitingtime;
          total_turnaround_time+=turnaroundtime;
       }
       cout << endl;
    }
```

- *For each task, the function prints the time intervals during which it executed.*
- *It calculates the completion time, turnaround time (completion time - arrival time), and waiting time (turnaround time - original burst time).*
- *It accumulates the total turnaround and waiting times for all tasks.*

```
double average_waiting_time = static_cast<double>(total_waiting_time) /
num_tasks;
double average_turnaround_time =
static_cast<double>(total_turnaround_time) / num_tasks;

cout << "Average Turnaround Time: " << average_turnaround_time << "
units" << endl;
cout << "Average Waiting Time: " << average_waiting_time << " units" <<
endl;
}
```

The function calculates and prints the average turnaround and waiting times.

This function effectively implements the SJF scheduling algorithm. It handles the scheduling by continuously selecting the shortest job that has arrived and is ready to execute. It logs the execution times, calculates the turnaround and waiting times for each task, and finally prints these values along with the average turnaround and waiting times for the entire set of tasks.

Advantages:

- SJF is optimal for minimizing the average waiting time among all scheduling algorithms.
- Prioritizes shorter processes, making it highly responsive to processes with shorter burst times.

Disadvantages:

- Longer processes may experience quite long waiting time if short processes keep arriving that lead to situation of Starvation.
- Although the SJF algorithm is optimal, it can not be implemented at the level of short-term CPU scheduling.
- Needs Prediction: Requires accurate knowledge or prediction of the remaining burst time for each process, which is not feasible.

SJF is best for environments needing optimal average waiting times and quick responses for short tasks but may involve complex implementation and potential starvation of longer tasks.

3. Priority Scheduling:

   - It selects the process based on their priority. Each process is assigned a priority, and the CPU is allocated to the process with the highest priority. If two processes have the same priority, they are scheduled according to their arrival order.
   - Preemptive/Non-Preemptive type algorithm. I had consider it to be Preemptive i.e., if a new process arrives with a higher priority than the currently running process, the current process is preempted.

   Implementation:

```
void P_scheduling(vector<Task> &tasks, const int &num_tasks) {
        int bursttime[num_tasks];
        for (int i = 0; i < num_tasks; i++)
        {
                bursttime[tasks[i].id - 1] = tasks[i].burst_time;
         }
         // Sort tasks primarily by arrival time, secondarily by burst time
   sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) {
      if (a.arrival_time == b.arrival_time) {
         return a.priority > b.priority;
      }
      return a.arrival_time < b.arrival_time;
   });
```

After initializing bursttime, the function sorts the tasks vector. It uses std::sort with a custom comparator function to sort tasks primarily by arrival time and secondarily by priority.

```
   vector<vector<int>> execution(num_tasks);
   cout << "\nPriority Scheduling:\n"<<endl;
   int current_time = 0;
   int total_turnaround_time = 0;
   int total_waiting_time = 0;
   int tasks_completed = 0;

   while (tasks_completed < num_tasks) {
      int highest_priority_index = -1;
      for (int i = 0; i < num_tasks; ++i) {
         if (tasks[i].arrival_time <= current_time && tasks[i].burst_time > 0) {
        if (highest_priority_index == -1 || tasks[i].priority >
        tasks[highest_priority_index].priority) {
               highest_priority_index = i;
            }
         }
      }
```

Inside the main loop, the function implements the scheduling logic. It iterates until all tasks are completed. Within the loop, it continuously selects the *task with the highest priority among those that have arrived and are ready to execute.*

```
        if (highest_priority_index == -1) {
          // No task is ready to execute, move to the next available task's
          arrival time
           current_time++;
           continue;
        }
```

If no task is ready to execute at the current time, it increments the current time and continues to the next iteration of the loop.

```
        Task &task = tasks[highest_priority_index];
        task.burst_time--;
        execution[task.id - 1].push_back(current_time);
```

Once a task is selected, its burst time is decremented, and the execution time is logged in the execution vector.

```
        if (task.burst_time == 0) {
           tasks_completed++;
        }

        current_time++;
     }
```

The function keeps track of completed tasks by incrementing the tasks_completed counter and updating the current time.

```
     // Sort tasks back by their IDs for output consistency
     sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) { return
  a.id < b.id; });
```

After completing the scheduling, tasks are sorted back by their IDs for output consistency.

```
// Print the execution times for each task
for (const Task &task : tasks) {
    cout << "Task " << task.id << " executed during time ";
    if (execution[task.id - 1].empty()) {
        cout << "no time";
    } else {
        int start = execution[task.id - 1][0], end = execution[task.id - 1][0];
        for (size_t i = 1; i < execution[task.id - 1].size(); ++i) {
            if (execution[task.id - 1][i] == execution[task.id - 1][i - 1] + 1) {
                end = execution[task.id - 1][i];
            } else {
                if (start == end) {
                    cout << start<<" to "<<start+1;
                } else {
                    cout << start << " to " << end+1;
                }
                cout << ", ";
                start = end = execution[task.id - 1][i];
            }
        }
        int completion_time=0;
        if (start == end) {
            cout << start<<" to "<<start+1<<" unit";
            completion_time=start+1;
        } else {
            cout << start << " to " << end+1<<" unit";
            completion_time=end+1;
        }
        int turnaroundtime=completion_time-task.arrival_time;
        int waitingtime=turnaroundtime-bursttime[task.id-1];
       cout << "\nTurn Around Time for Task " << task.id << " is " <<
       turnaroundtime << " unit" << endl;
       cout << "Waiting Time for Task " << task.id << " is " << waitingtime
       << " unit"
         << endl;
        total_waiting_time+=waitingtime;
        total_turnaround_time+=turnaroundtime;
    }
    cout << endl;
```

```
   }
```

For each task, the function prints the execution time intervals and calculates turnaround and waiting times.

```
double average_waiting_time = static_cast<double>(total_waiting_time)
/ num_tasks;
double average_turnaround_time =
static_cast<double>(total_turnaround_time) / num_tasks;

   cout << "Average Turnaround Time: " << average_turnaround_time << "
units" << endl;
   cout << "Average Waiting Time: " << average_waiting_time << " units"
<< endl;
}
```

Finally, the function calculates and prints the average turnaround and waiting times for all tasks.

The P_scheduling function efficiently implements Priority Scheduling by continuously selecting and executing tasks based on their priority. It handles task execution, tracks execution times, calculates turnaround and waiting times, and provides detailed output and average metrics.

Advantages:

- Can assign priorities based on importance, urgency, or other criteria.
- Efficient for Critical Tasks: Ensures that high-priority tasks are executed quickly.

Disadvantages:

- Starvation: Low-priority processes may suffer indefinite postponement if higher-priority processes keep arriving.
- Aging Required: To prevent starvation, aging techniques may need to be implemented, where the priority of a process increases the longer it waits.

Priority scheduling is best for systems needing to handle tasks with varying importance or urgency but requires careful management to avoid starvation of low-priority tasks.

4. Round Robin Scheduling:

   - Simple preemptive CPU scheduling algorithm
   - Each process is assigned a fixed time slice or time quantum. The CPU scheduler cyclically switches between processes, allocating each process a time quantum to execute. If a process completes its execution within the time quantum, it is removed from the queue. If not, it is moved to the end of the queue to await its next turn.

Implementation:

```
void RR_scheduling(vector<Task> &tasks, const int &num_tasks, const int
&time_quantum) {
   queue<Task*> task_queue;
   int current_time = 0;
   int total_turnaround_time = 0;
   int total_waiting_time = 0;
   int tasks_completed = 0;
   cout << "\nRound Robin Scheduling:\n"<<endl;

   vector<pair<int, int>> execution[num_tasks];
```

The function starts by initializing various variables and data structures:

- *A queue named task_queue is created to hold pointers to tasks.*
- *current_time is initialized to track the current time during scheduling.*
- *Variables total_turnaround_time and total_waiting_time are initialized to accumulate the total turnaround and waiting times, respectively.*
- *tasks_completed is initialized to keep track of the number of tasks that have completed execution.*
- *A vector named execution is initialized to store the execution intervals for each task.*

```
// Sort tasks by arrival time
sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) {
    return a.arrival_time < b.arrival_time;
});

// Add initial tasks to the queue
auto task_it = tasks.begin();
while (task_it != tasks.end() && task_it->arrival_time <= current_time) {
    task_queue.push(&(*task_it));
    task_it++;
}
```

The function sorts the tasks by their arrival time using std::sort. This ensures that tasks are processed in the order they arrive.
The initial tasks that have already arrived (i.e., their arrival time is less than or equal to the current time) are added to the task queue.

```
while (tasks_completed < num_tasks) {
    if (!task_queue.empty()) {
        Task *task = task_queue.front();
        task_queue.pop();

        int exec_time = min(time_quantum, task->remaining_time);

       execution[task->id - 1].emplace_back(current_time, current_time +
       exec_time);
        task->remaining_time -= exec_time;
        current_time += exec_time;

        // Add new tasks to the queue that have arrived
       while (task_it != tasks.end() && task_it->arrival_time <=
       current_time) {
            task_queue.push(&(*task_it));
            task_it++;
        }

        if (task->remaining_time > 0) {
            task_queue.push(task);
            } else {
                int turnaround_time = current_time - task->arrival_time;
                int waiting_time = turnaround_time - task->burst_time;

                total_turnaround_time += turnaround_time;
                total_waiting_time += waiting_time;
                tasks_completed++;
            }
        } else {
            current_time++;
        }
    }
        sort(tasks.begin(), tasks.end(), [](const Task &a, const Task &b) {
        return a.id < b.id; });
```

The main scheduling logic is implemented within a while loop that continues until all tasks are completed.
Within this loop:

- *If the task queue is not empty, a task is dequeued and executed for a fixed time quantum or until its remaining time is exhausted.*
- *The execution interval for the task is recorded in the execution vector.*
- *Any new tasks that have arrived during this time quantum are added to the queue.*
- *If the task's remaining time is still greater than 0 after execution, it is enqueued again.*
- *If the task has completed its execution, its turnaround and waiting times are calculated, and it is considered completed.*

```
for (const Task &task : tasks) {
        cout << "Task " << task.id << " executed during time ";
        for (const auto &period : execution[task.id - 1]) {
          cout << period.first << " to " << period.second << (period ==
          execution[task.id - 1].back() ? " unit" : ", ");
        }
        cout << endl;
          int turnaround_time = execution[task.id - 1].back().second -
          task.arrival_time;
        int waiting_time = turnaround_time - task.burst_time;
        cout << "Turn Around Time for Task " << task.id << " is " <<
        turnaround_time << " unit" << endl;
        cout << "Waiting Time for Task " << task.id << " is " << waiting_time
        << " unit\n" << endl;
    }
```

After completing the scheduling, tasks are sorted back by their IDs for output consistency.
For each task, the function prints the execution time intervals and calculates turnaround and waiting times based on the recorded execution intervals.

```
double average_waiting_time = static_cast<double>(total_waiting_time) /
num_tasks;
    double average_turnaround_time =
static_cast<double>(total_turnaround_time) / num_tasks;
    cout << "\nAverage Turnaround Time: " << average_turnaround_time
<< " unit" << endl;
    cout << "Average Waiting Time: " << average_waiting_time << " unit" <<
endl;
}
```

Finally, the function calculates and prints the average turnaround and waiting times for all tasks.

Advantages:

- Fairness: Provides fair allocation of CPU time among all processes.
- Efficient for Time-Sharing Systems: Suitable for time-sharing systems where multiple users require quick responses.

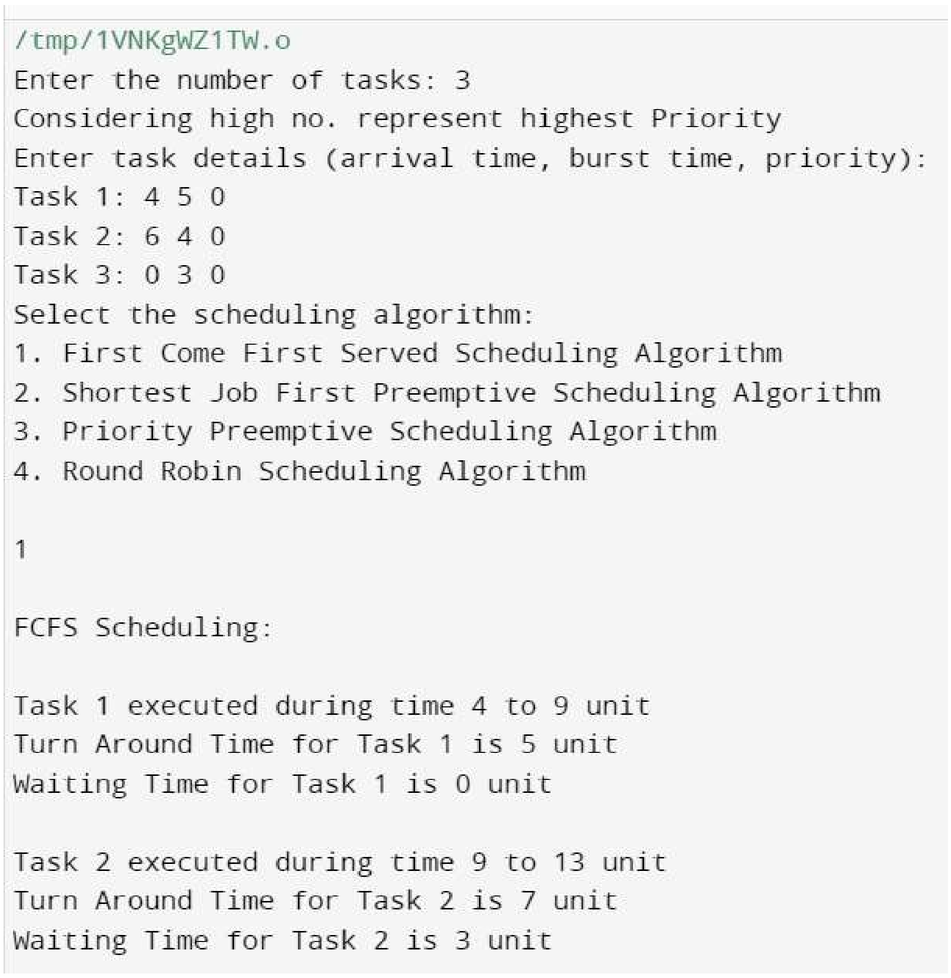
Disadvantages:

- Higher Turnaround Time for Longer Processes: Longer processes may need to wait for multiple time slices before completing, leading to higher turnaround time.
- Poor Performance with Uneven Time Quantums: If the time quantum is too large, RR can resemble FCFS, and if it's too small, there's increased overhead due to frequent context switching.

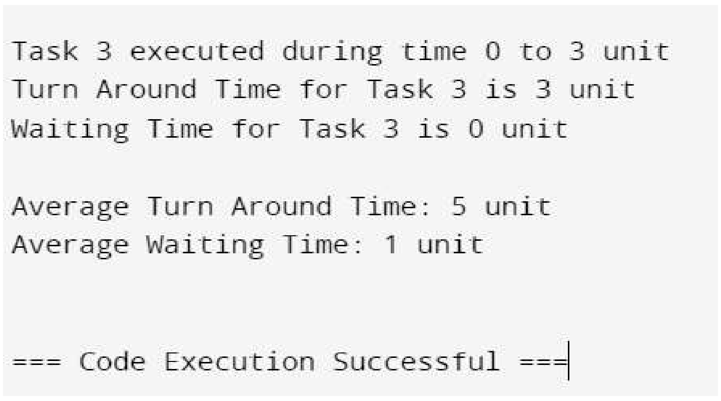
Round Robin is best for time-sharing and interactive systems where fairness and responsiveness are essential, but requires careful selection of the time quantum to balance efficiency and performance.

Test Run of Main Code:

For FCFS scheduling:

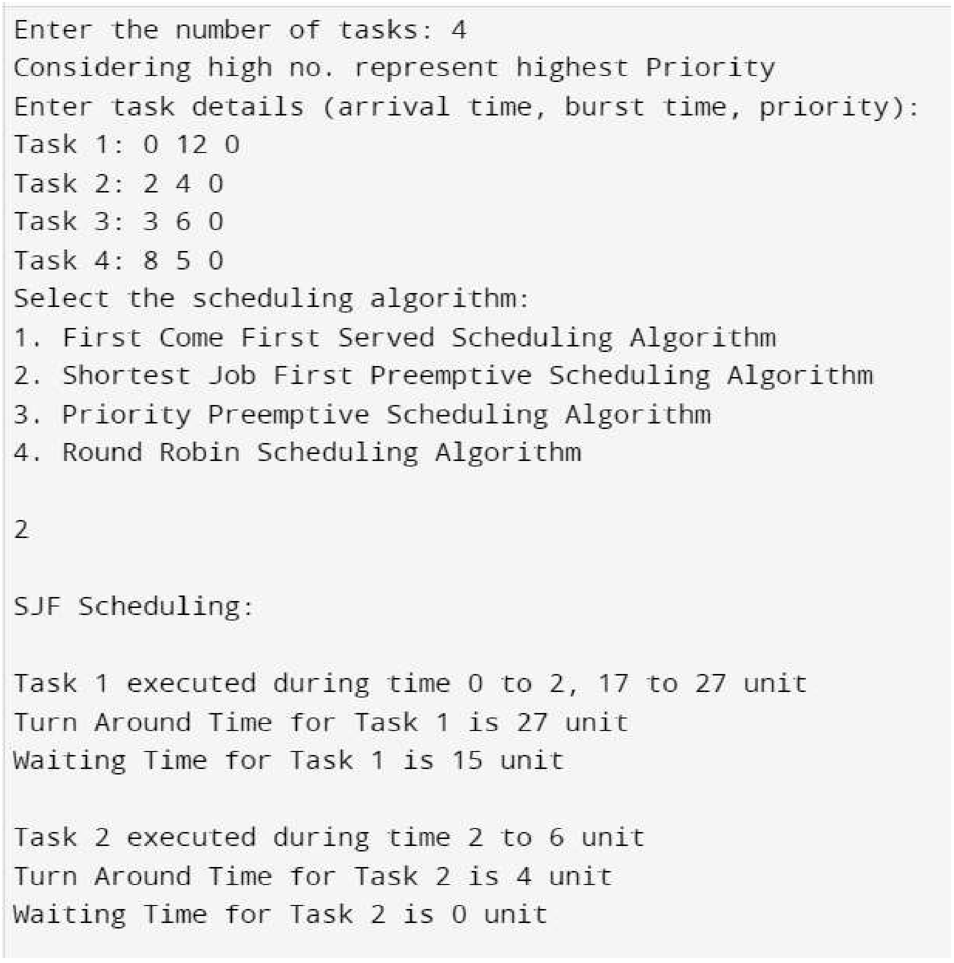


```
/tmp/1VNKgWZ1TW.o
Enter the number of tasks: 3
Considering high no. represent highest Priority
Enter task details (arrival time, burst time, priority):
Task 1: 4 5 0
Task 2: 6 4 0
Task 3: 0 3 0
Select the scheduling algorithm:
1. First Come First Served Scheduling Algorithm
2. Shortest Job First Preemptive Scheduling Algorithm
3. Priority Preemptive Scheduling Algorithm
4. Round Robin Scheduling Algorithm

1

FCFS Scheduling:

Task 1 executed during time 4 to 9 unit
Turn Around Time for Task 1 is 5 unit
Waiting Time for Task 1 is 0 unit

Task 2 executed during time 9 to 13 unit
Turn Around Time for Task 2 is 7 unit
Waiting Time for Task 2 is 3 unit
```

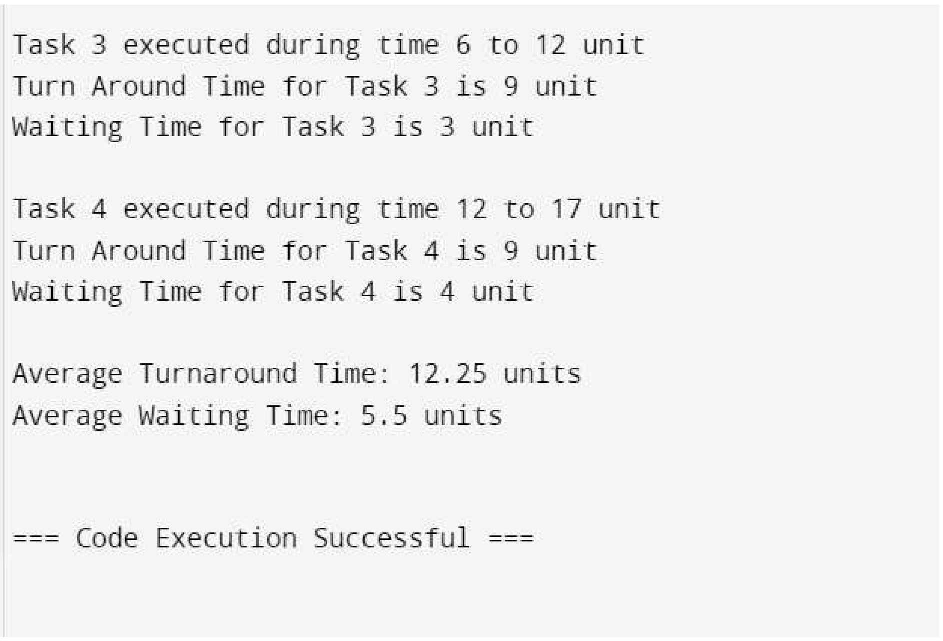


```
Task 3 executed during time 0 to 3 unit
Turn Around Time for Task 3 is 3 unit
Waiting Time for Task 3 is 0 unit

Average Turn Around Time: 5 unit
Average Waiting Time: 1 unit


=== Code Execution Successful ===
```

For Preemptive SJF scheduling:

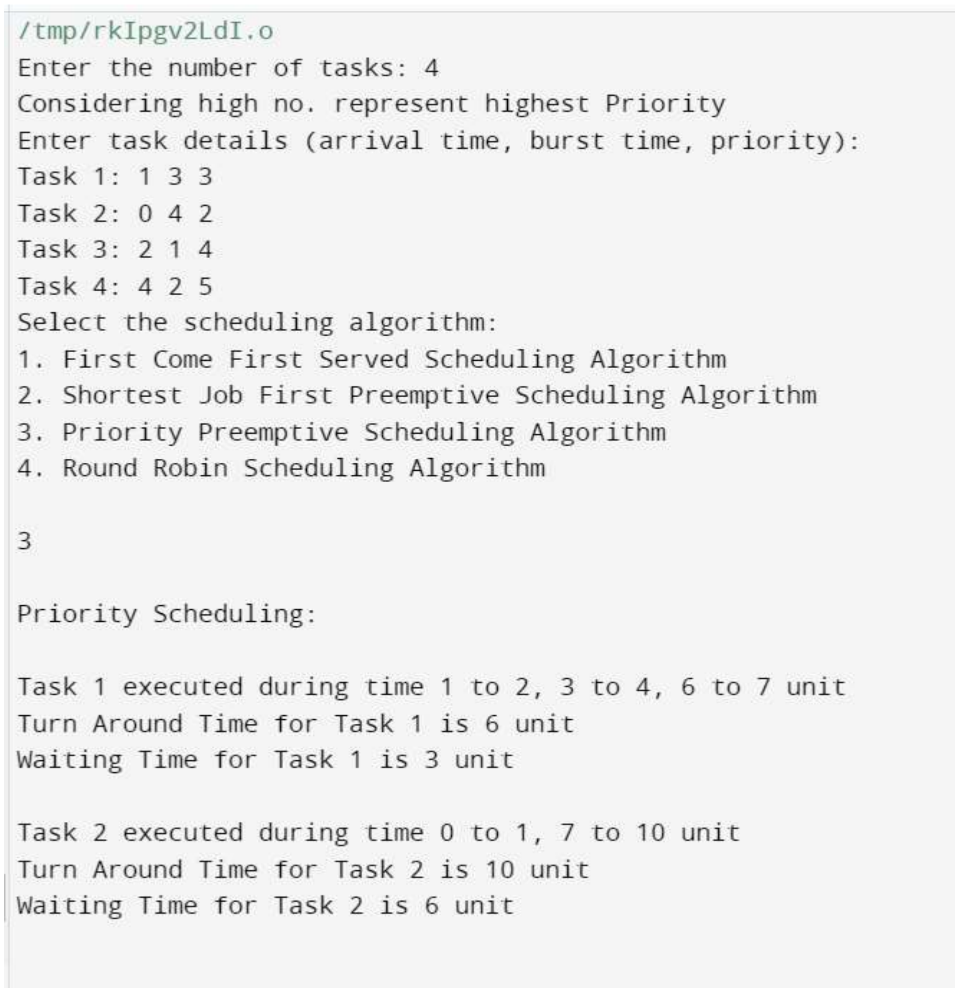


```
Enter the number of tasks: 4
Considering high no. represent highest Priority
Enter task details (arrival time, burst time, priority):
Task 1: 0 12 0
Task 2: 2 4 0
Task 3: 3 6 0
Task 4: 8 5 0
Select the scheduling algorithm:
1. First Come First Served Scheduling Algorithm
2. Shortest Job First Preemptive Scheduling Algorithm
3. Priority Preemptive Scheduling Algorithm
4. Round Robin Scheduling Algorithm

2

SJF Scheduling:

Task 1 executed during time 0 to 2, 17 to 27 unit
Turn Around Time for Task 1 is 27 unit
Waiting Time for Task 1 is 15 unit

Task 2 executed during time 2 to 6 unit
Turn Around Time for Task 2 is 4 unit
Waiting Time for Task 2 is 0 unit
```

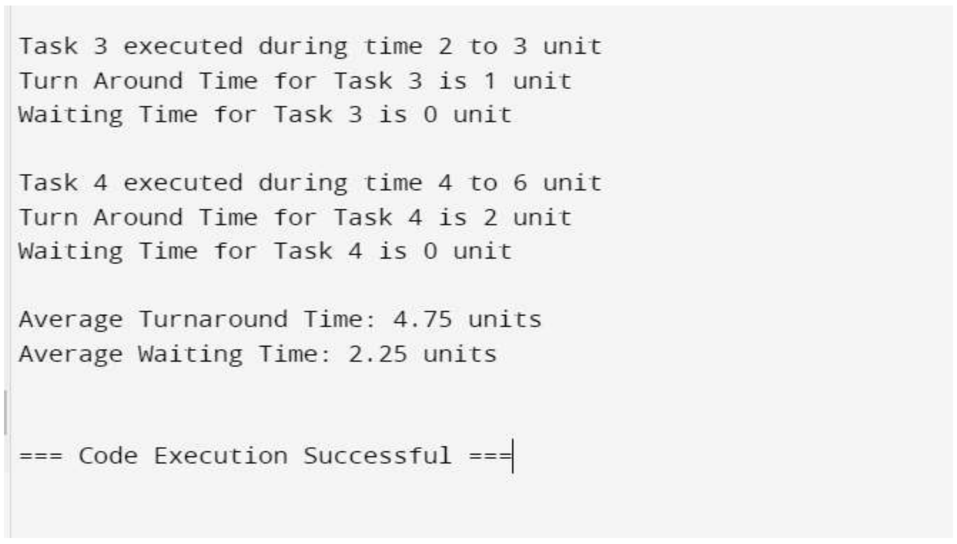


```
Task 3 executed during time 6 to 12 unit
Turn Around Time for Task 3 is 9 unit
Waiting Time for Task 3 is 3 unit

Task 4 executed during time 12 to 17 unit
Turn Around Time for Task 4 is 9 unit
Waiting Time for Task 4 is 4 unit

Average Turnaround Time: 12.25 units
Average Waiting Time: 5.5 units


=== Code Execution Successful ===
```

For Preemptive Priority scheduling:

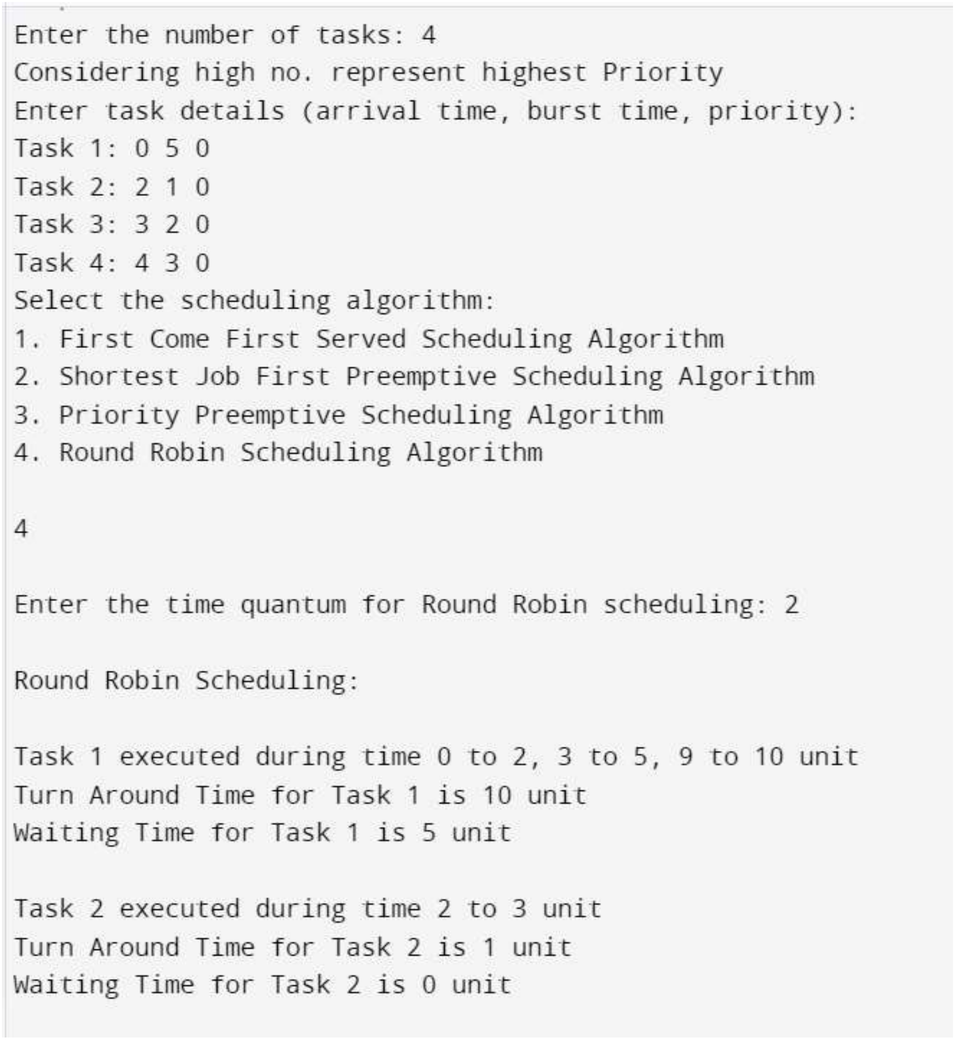


```
/tmp/rkIpgv2LdI.o
Enter the number of tasks: 4
Considering high no. represent highest Priority
Enter task details (arrival time, burst time, priority):
Task 1: 1 3 3
Task 2: 0 4 2
Task 3: 2 1 4
Task 4: 4 2 5
Select the scheduling algorithm:
1. First Come First Served Scheduling Algorithm
2. Shortest Job First Preemptive Scheduling Algorithm
3. Priority Preemptive Scheduling Algorithm
4. Round Robin Scheduling Algorithm

3

Priority Scheduling:

Task 1 executed during time 1 to 2, 3 to 4, 6 to 7 unit
Turn Around Time for Task 1 is 6 unit
Waiting Time for Task 1 is 3 unit

Task 2 executed during time 0 to 1, 7 to 10 unit
Turn Around Time for Task 2 is 10 unit
Waiting Time for Task 2 is 6 unit
```

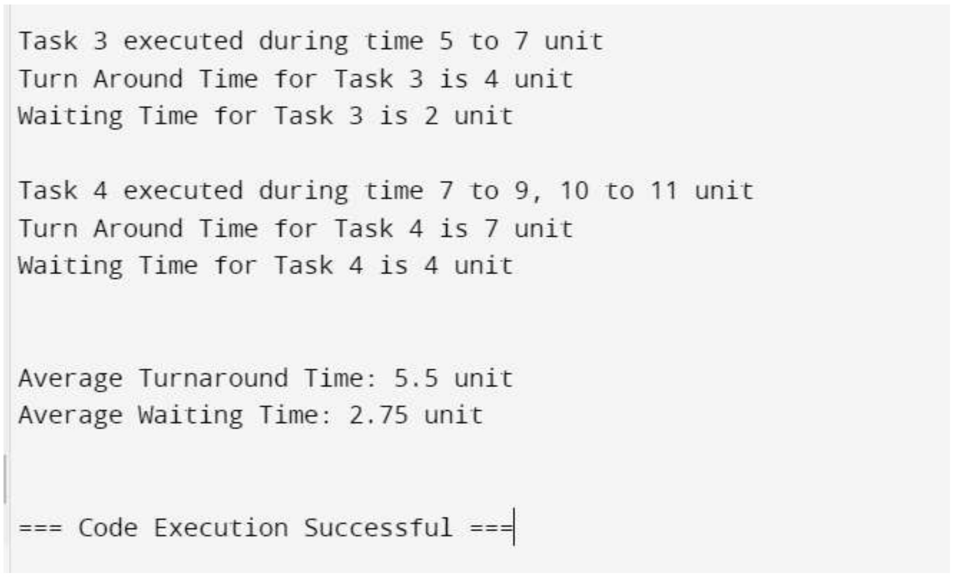


```
Task 3 executed during time 2 to 3 unit
Turn Around Time for Task 3 is 1 unit
Waiting Time for Task 3 is 0 unit

Task 4 executed during time 4 to 6 unit
Turn Around Time for Task 4 is 2 unit
Waiting Time for Task 4 is 0 unit

Average Turnaround Time: 4.75 units
Average Waiting Time: 2.25 units


=== Code Execution Successful ===
```

For Round-Robin scheduling:



```
Enter the number of tasks: 4
Considering high no. represent highest Priority
Enter task details (arrival time, burst time, priority):
Task 1: 0 5 0
Task 2: 2 1 0
Task 3: 3 2 0
Task 4: 4 3 0
Select the scheduling algorithm:
1. First Come First Served Scheduling Algorithm
2. Shortest Job First Preemptive Scheduling Algorithm
3. Priority Preemptive Scheduling Algorithm
4. Round Robin Scheduling Algorithm

4

Enter the time quantum for Round Robin scheduling: 2

Round Robin Scheduling:

Task 1 executed during time 0 to 2, 3 to 5, 9 to 10 unit
Turn Around Time for Task 1 is 10 unit
Waiting Time for Task 1 is 5 unit

Task 2 executed during time 2 to 3 unit
Turn Around Time for Task 2 is 1 unit
Waiting Time for Task 2 is 0 unit
```



```
Task 3 executed during time 5 to 7 unit
Turn Around Time for Task 3 is 4 unit
Waiting Time for Task 3 is 2 unit

Task 4 executed during time 7 to 9, 10 to 11 unit
Turn Around Time for Task 4 is 7 unit
Waiting Time for Task 4 is 4 unit


Average Turnaround Time: 5.5 unit
Average Waiting Time: 2.75 unit


=== Code Execution Successful ===
```

Conclusion:

In this project, I explored various CPU scheduling algorithms, including First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin (RR). Each algorithm offers unique advantages and challenges, making them suitable for different system requirements and workloads.

FCFS is straightforward but can lead to inefficiencies for shorter tasks. SJF optimizes waiting time but can starve longer processes. Priority Scheduling allows flexibility in process handling but requires careful management to prevent starvation. Round Robin ensures fairness and responsiveness, ideal for interactive systems, but requires an appropriately chosen time quantum.

Understanding these algorithms and their trade-offs is crucial for designing effective CPU schedulers that balance performance, fairness, and complexity. This project provides a comprehensive overview of CPU scheduling, enabling more informed decisions in process management and system optimization.