

Course Name: Operating Systems

Course Code: CS2006

Operating Systems Project Title:

CPU Process Scheduling Methods

# Group Members:

**Muhammad Anas Hassan 20K-1726**

**Muhammad Warzan 20K-1649**

**TEACHERS NAME: MISS SAFIA BBALOCH**

**DATE OF SUBMISSION: 27th MAY 2022**

**INTRODUCTION:**

CPU process scheduling is a technique that allows one process to use the CPU while another's execution is halted (in a waiting state) due to the lack of a resource such as I/O, allowing the CPU to be fully utilized. CPU scheduling is to make the system more efficient, quick, and fair. Pre-emptive and non-pre-emptive scheduling are the two forms of scheduling. There are two types of scheduling. The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called preemptive scheduling. The scheduling in which a running process cannot be interrupted by any other process is called non-preemptive scheduling. First come first serve scheduling (FCFS), round-robin scheduling (RR), priority scheduling (preemptive), priority scheduling (non-preemptive), SJF scheduling (non-preemptive), SJF scheduling (preemptive) also known as SRTF (shortest remaining time first) scheduling are some of the CPU scheduling algorithms that are implemented.

**EXPLANATION:**

There are mainly four types of process scheduling algorithms which we covered in our OS Project.

1. **First Come First Serve (FCFS) Scheduling:**

Operating systems and networks employ the First Come, First Served (FCFS) scheduling method to effectively and automatically execute queued tasks, processes, and requests in the order of their arrival. A FIFO queue can be used to manage this scheduling strategy.

1. **Shortest-Job-First (SJF) Scheduling:**

The algorithm Shortest Job First (SJF) selects the process with the shortest execution time for the next execution. This style of scheduling can be either preemptive or non-preemptive. It significantly reduces the average time spent waiting for other processes to complete. SJF stands for Shortest Job First in its full form.

**Preemptive:**

Jobs are placed in the ready queue as they arrive in Preemptive SJF Scheduling. The process with the smallest burst time is the first to run. If a process with a shorter burst time comes, the current one is terminated or preempted, and the shorter job is given a CPU cycle.

**Non-preemptive:**

When a CPU cycle is allotted to a process under non-preemptive scheduling, the process keeps it until it reaches a waiting state or is terminated.

1. **Priority Scheduling:**

Priority Scheduling is a priority-based strategy for scheduling processes. The scheduler uses this method to choose which tasks to work on based on their priority. Jobs with higher priorities should be completed first, whereas jobs with equal priorities should be completed in a round-robin or FCFS fashion. Prioritization is determined by memory limitations, time constraints, and other factors.

**Preemptive:**

Tasks are generally allocated with their priority in Preemptive Scheduling. Even though the lower priority activity is still running, it is sometimes necessary to run a higher priority task before a lower priority task. The lower priority work is put on hold for a while and then continues when the higher priority activity is completed.

**Non-preemptive:**

The CPU has been assigned to a certain process in this scheduling mechanism. The process that keeps the CPU occupied will either switch context or terminate to relieve the CPU. It's the only approach that works across a variety of hardware platforms. That's because, unlike preemptive scheduling, it doesn't require any particular hardware (such as a timer).

1. **Round Robin (RR) Scheduling:**

The round-robin concept, in which each individual gets an equal amount of something in turn, inspired the name of this method. It is the simplest and oldest scheduling method, and it is mostly used for multitasking. Round-robin scheduling allows each ready job to run in a cyclic queue for a certain amount of time. This technique also allows for process execution without hunger.

**METHODOLOGY:**

1. **First Come First Serve (FCFS) Scheduling:**

The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time of each process can also be entered.

Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

* Stable sort the processes in order of arrival time in ascending order.
* Calculate start time and completion time for each process
* Start time = (i == 0)?p[i].arrival\_time:max(p[i].arrival\_time,p[i-1].completion\_time)
* completion\_time = start\_time + p[i].burst\_time
* Calculate TAT, WT and RT using formulas

1. **Shortest Job First (SJF) Scheduling:**

**Preemptive:**

The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time of each process can also be entered.

Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

completed = 0

current\_time = 0

while(completed != n) {

find process with minimum burst time among process that are in ready queue at current\_time

if(process found) {

if(process is getting CPU for the first time) {

start\_time = current\_time

}

burst\_time = burst\_time - 1

current\_time = current\_time + 1

if(burst\_time == 0) {

completion\_time = current\_time

turnaround\_time = completion\_time - arrival\_time

waiting\_time = turnaround\_time - burst\_time

response\_time = start\_time - arrival\_time

mark process as completed

completed++

}

}

else {

current\_time++

}

}

**Non-preemptive:**

The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time of each process can also be entered.

Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

completed = 0

current\_time = 0

while(completed != n) {

find process with minimum burst time among process that are in ready queue at current\_time

if(process found) {

start\_time = current\_time

completion\_time = start\_time + burst\_time

turnaround\_time = completion\_time - arrival\_time

waiting\_time = turnaround\_time - burst\_time

response\_time = start\_time - arrival\_time

mark process as completed

completed++

current\_time = completion\_time

}

else {

current\_time++

}

1. **Priority Scheduling:**

**Preemptive:**

The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time and the priority of each process can also be entered.

Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

completed = 0

current\_time = 0

while(completed != n) {

find process with maximum priority time among process that are in ready queue at current\_time

if(process found) {

if(process is getting CPU for the first time) {

start\_time = current\_time

}

burst\_time = burst\_time - 1

current\_time = current\_time + 1

if(burst\_time == 0) {

completion\_time = current\_time

turnaround\_time = completion\_time - arrival\_time

waiting\_time = turnaround\_time - burst\_time

response\_time = start\_time - arrival\_time

mark process as completed

completed++

}

}

else {

current\_time++

}

**Non-preemptive:**

The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time and the priority of each process can also be entered.

Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

completed = 0

current\_time = 0

while(completed != n) {

find process with maximum priority among process that are in ready queue at current\_time

if(process found) {

start\_time = current\_time

completion\_time = start\_time + burst\_time

turnaround\_time = completion\_time - arrival\_time

waiting\_time = turnaround\_time - burst\_time

response\_time = start\_time - arrival\_time

mark process as completed

completed++

current\_time = completion\_time

}

else {

current\_time++

}

}

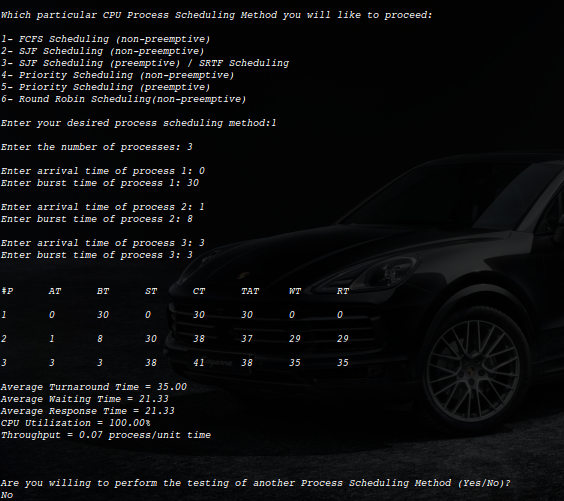
1. **Round Robin (RR) Scheduling:**

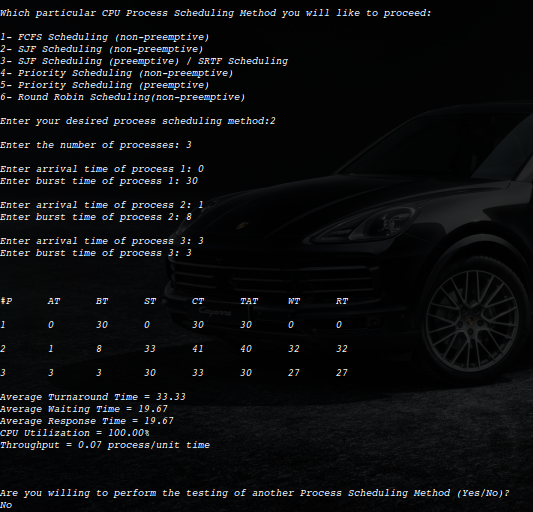
The number of processes and the arrival time of each process are the input variables. If all processes arrive at the same time, this may be set to 0, and the burst time and the priority of each process can also be entered.

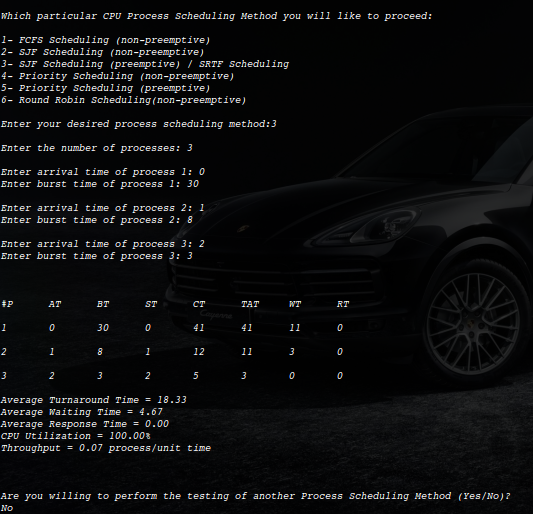
Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT) for each process, as well as Average Turnaround Time, Average Waiting Time, and Average Response Time, as well as throughput and CPU usage are all displayed in the output. The algorithm is;

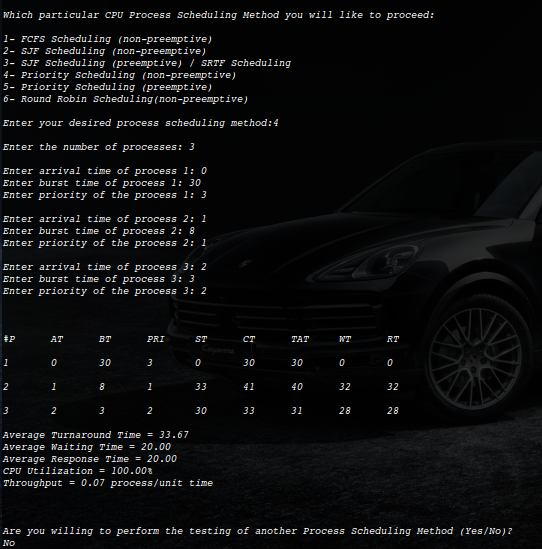
1. Stable sort the processes in order of arrival time in ascending order.
2. We will be using a FIFO queue to implement this algorithm
3. We will first push the first process from the sorted list into queue.
4. We will use a array to check if the process is in the queue or not.
5. Keep track of the time using a variable - current\_time
6. If the process is getting CPU for the first time, record its start time as current\_time.
7. Give quantum unit of time to the process that is at front in the queue and pop this process from the queue.
8. If the burst time of this process becomes 0, calculate CT, TAT, WT and RT for it.
9. If some process has arrived when this process was executing, insert them into the queue.
10. If the current process has burst time remianing, push the process into queue again.
11. If the queue is empty, pick the first process from the list that is not completed.
12. Keep doing this till all processes are completed.

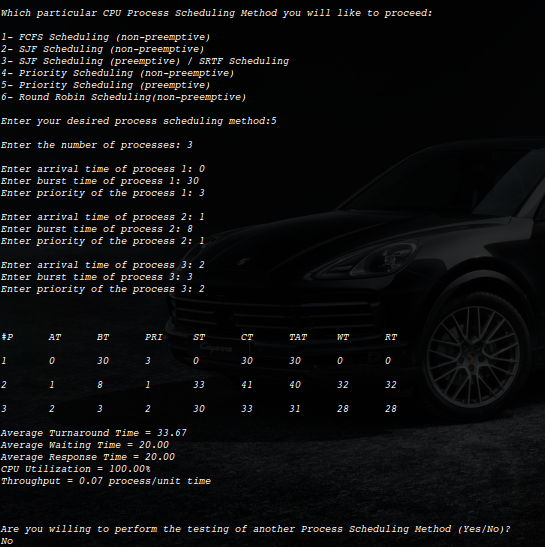
**RESULT/OUTPUT SCREEN SHOOT:**

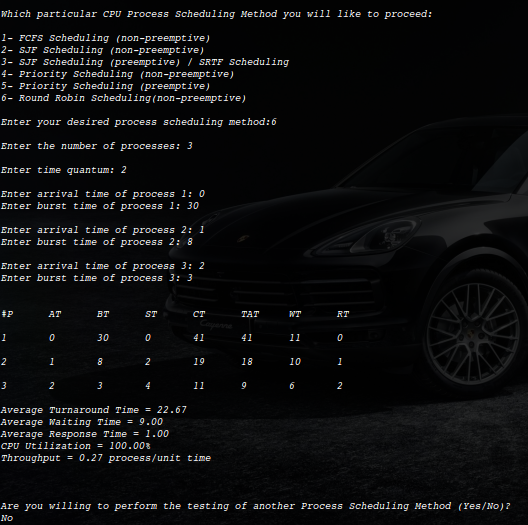
****

****

****

****

****

****

**Settings:**

The settings required to run this scheduling algorithm are as follows

* GNU compiler GCC collection
* Development tools
* Development libraries
* IDE or text editor to write programs

Visual studio code is our IDE. First, we download the GNU compiler after it we need all supported extensions. There is no kernel-level or any Api involved in this user-level scheduling algorithm.

**Reference:**

<https://www.guru99.com/cpu-scheduling-algorithms.html>

<https://www.studytonight.com/operating-system/cpu-scheduling>

<https://ecomputernotes.com/fundamental/disk-operating-system/cpu-scheduling-algorithms>