Programming Assignment-3 Report
CS3523: Operating Systems-2
Implementing Rate-Monotonic Scheduling & Earliest Deadline First Scheduling through Discrete Event Simulation

Name: Vikhyath Sai Kothamasu
Roll Number: CS20BTECH11056

Goal: The goal of this assignment is to implement a program in C++ using discrete event simulation to simulate the Rate-Monotonic & Earliest Deadline First (EDF) scheduling algorithms. Then compare the average waiting time and deadlines missed of both algorithms.

Low-Level Design: For either of the source codes to run as expected, there needs to be an inp-params.txt file in the same directory which contains the necessary input information required by the source code. Once we have the required file and the compiled source code is executed, it first inputs the number of processes (say n). Then, it inputs n lines each containing a process ID, its corresponding processing time, period, and the number of times it is repeated. Every instance of all the processes is stored in `all_processes, sorted_processes` initially. Then the processes in `sorted_processes` are sorted based on their arrival time and if it is the same, then sorted based on their priority. Without loss of generality, let's assume that this explanation uses the RMS scheduling algorithm. For the EDF scheduling algorithm, the only difference is that the priority of each process is calculated differently (1/deadline).

Then a `ready_queue` vector is initialized to keep track of the processes which are ready to be executed. A variable named current_time is also initialized which keeps track of simulated time. Since we sorted the processes based on arrival time and priority, the first process to be executed will be first in the list `sorted_processes.` Now a while loop runs until all the processes are either completed or terminated. This is achieved only when both the `ready_queue` and `sorted_processes` completely empty.

If none of them are empty, we extract the process with the highest priority from the `ready_queue` and check if it can meet its deadline. At any point of the simulation, if we know beforehand that a process will not be able to meet its deadline, the process is terminated immediately giving way to further processes. If the process can meet its deadline, first we extract all processes from `sorted_processes` which are supposed to arrive on or before the `current_time`. Since new processes have been added, we extract the process with the highest priority again from the `ready_queue` and compare the priority values. We choose the process with the higher priority and push the other process back into the `ready_queue`. We check to see if this process is running for the first time or if it has been preempted earlier and is running again to complete processing. Once this check is done, we keep adding processes that arrive at the `ready_queue` before the current process is done executing. If we encounter a process that has a higher priority than that of the current process, we preempt the current process, update the remaining processing time of the current process, update `current_time`

value and break out of the loop so that the new process can start running. Now if encounter a situation where `sorted_processes` is empty while `ready_queue` has only one process, then we have reached the last process. Once the last process is done, the program breaks out of the loop to end the simulation. If no process preempted the current process, then the process has completed its processing and the `current_time` value is updated accordingly.

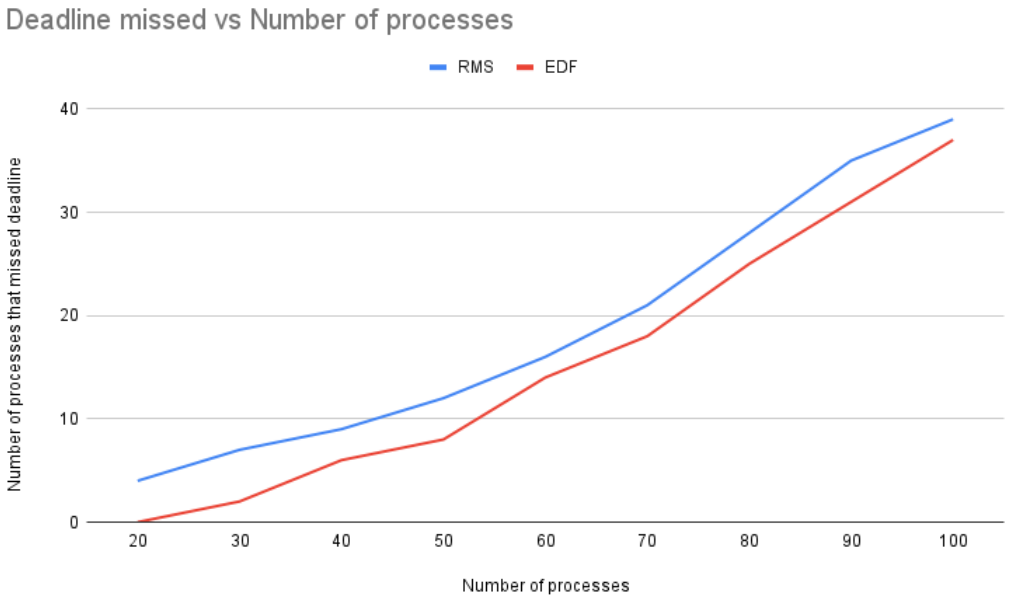
Alternatively, if `sorted_processes` is not empty yet but `ready_queue` is empty, it means that the CPU stays idle for a while until another process from `sorted_process` arrives. Since it is sorted, we extract the first process from it and update `current_time` accordingly. Now, if the `sorted_process` is empty, it means that it had only one process earlier and now we extracted it so we run that process and exit the loop as all processes have been executed.

For the last alternative, `sorted_processes` can be empty while the `ready_queue` is not empty. This means that all the processes have entered the ready queue and if a process is running, no process will be able to preempt it because there is no addition of new processes and the existing processes run in a manner based on their priorities. So we keep extracting the process with the highest priority and execute the process. This continues until the `ready_queue` is completely empty signifying that all processes have been executed/terminated successfully.

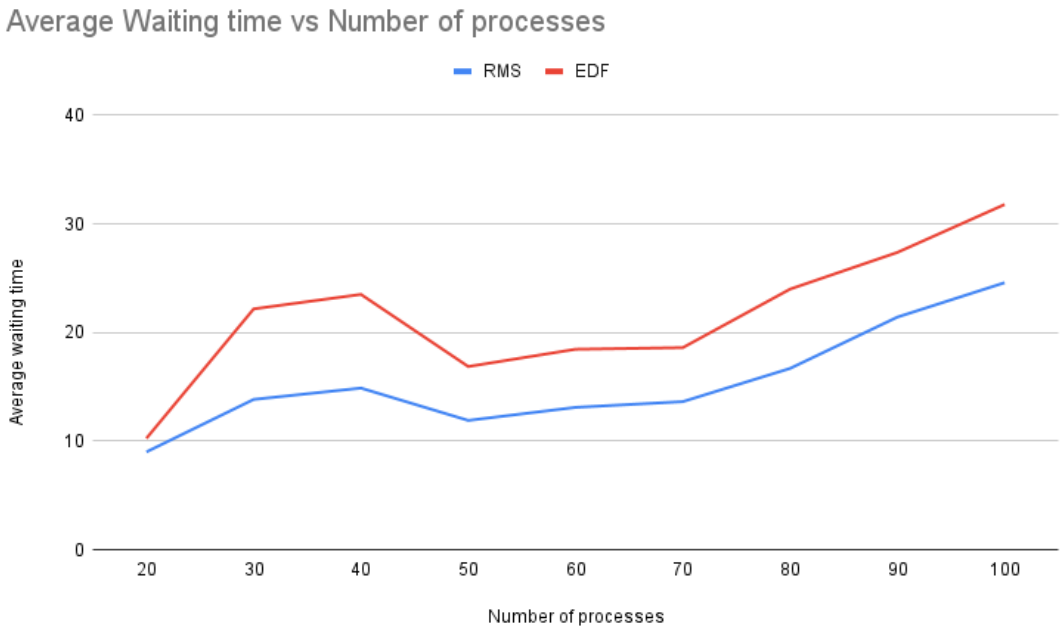
Finally, the code prints necessary stats namely the number of processes that entered the system, how many of those processes were successfully completed or terminated as they missed their deadlines, and also the average waiting time for each process and overall average waiting time. This marks the end of the program.

I would like to make note that while calculating the waiting time for a terminated process, I have considered the waiting time up until the point whereafter if the process starts running, it will not be able to meet its deadline at any cost.

Graph1: Deadline missed vs Number of processes



Graph2: Average Waiting time vs Number of processes



Sample Input:
2
1 20 50 3
2 35 60 3

Corresponding Sample Output:

1. On running RMS scheduling algorithm:
    a. RM-Log
    Process P1: processing time=20; deadline:50; period:50 joined the system at time 0
    Process P2: processing time=35; deadline:60; period:60 joined the system at time 0
    Process P1 starts execution at time 1.
    Process P1 finishes execution at time 20.
    Process P2 starts execution at time 21.
    Process P2 is preempted by Process P1 at time 50. Remaining process time:5
    Process P1 starts execution at time 51.
    Process P1 finishes execution at time 70.
    Process P2 could not meet its deadline at time 60 and thus got terminated
    Process P2 starts execution at time 71.
    Process P2 is preempted by Process P1 at time 100. Remaining process time:5
    Process P1 starts execution at time 101.
    Process P1 finishes execution at time 120.
    Process P2 could not meet its deadline at time 120 and thus got terminated

Process P2 starts execution at time 121.
Process P2 finishes execution at time 155.

b. RM-Stats
Number of processes that came into the system: 6
Number of processes that successfully completed: 4
Number of processes that missed their deadlines: 2
Average waiting time for each process is given below
Process P1 Average waiting time: 0
Process P2 Average waiting time: 16.6667
Average waiting time: 8.33333

2. On running EDF scheduling algorithm:
   a. EDF-Log
   Process P1: processing time=20; deadline:50; period:50 joined the system at time 0
   Process P2: processing time=35; deadline:60; period:60 joined the system at time 0
   Process P1 starts execution at time 1.
   Process P1 finishes execution at time 20.
   Process P2 starts execution at time 21.
   Process P2 finishes execution at time 55.
   Process P1 starts execution at time 56.
   Process P1 finishes execution at time 75.
   Process P2 starts execution at time 76.
   Process P2 finishes execution at time 110.
   Process P1 starts execution at time 111.
   Process P1 finishes execution at time 130.
   Process P2 starts execution at time 131.
   Process P2 finishes execution at time 165.
   b. EDF-Stats
   Number of processes that came into the system: 6
   Number of processes that successfully completed: 6
   Number of processes that missed their deadlines: 0
   Average waiting time for each process is given below
   Process P1 Average waiting time: 5
   Process P2 Average waiting time: 15
   Average waiting time: 10