Operating Systems II : CS3523
Spring 2019

Programming Assignment 1 : Implementing Rate-Monotonic Scheduling & Earliest Deadline First Scheduling through Discrete Event Simulation

Instructor : Dr. Sathya Peri
Author : Puneet Mangla (CS17BTECH11029)

Design of Program :

We implemented **Rate – Monotonic** & **Earliest Deadline First** scheduling algorithms and compared / tested them using **Discrete Event Stimulation**.

`class Process`

A process in the program is defined by an object of class Process. The class contains the following parameters as process attributes to make the flow easy :

`int pid` - process ID

`int process_time` – time take to complete execution

`int period` – period of arrival of process

`int deadline` – timestamp at which its deadline is completed

`int arrival_time` – arrival time in the system

`int interuppted` – whether it was preempted or not

`int nowaiting_time` – time it should finish execution when there is no waiting

`nowaiting_time` and **`deadline`** are calculated as follows :

```
nowaiting_time =  arrival_time +  process_time
deadline  =  arrival_time +  period
```

`time and next_arrival`

`time` is the current stimulated clock time whereas `next_arrival` is the least time when a new process arrives in a system , this help us to hop directly to `next_arrival` ensuring discrete time stimulation.

`priority_queue<Process> processes`

We maintained a priority queue that stores all the future coming processes **prioritized according to their arrival time**.

`map<Process> ready_queue`

A map is maintained for the processes currently in ready queue for fast indexing . Processes are scheduled from this ready queue only.

`Process schedule(map<int,Process>* ready_queue)`

This functions takes out a process from the ready queue with the highest priority and returns it .

The highest priority process is the process with least **period** and least **deadline** in case of **RMS and EDF** respectively.

Comparison metrics :

1. **Missed Deadlines :** Number of processes that missed their deadlines i.e before completing execution they expired.

2. **Average waiting time :** Average time spent by a process in ready queue waiting to be dispatched.

Algorithm :

1. Initially at `time = 0` , `ready_queue` is empty and algorithms starts .

2. At current clock `time` (intially 0) processes whose `arrival_time` is current time are poped from `processes` queue and placed in the `ready_queue`.

3. The `next_arrival` time is set to the arrival time of top process of `processes` queue . This will be the time when new process(es) arrive in the system.

4. A process is scheduled from `ready_queue` by calling schedule function and assigned as `running` process.

5. We keep on scheduling processes from `ready_queue` till the a **process A' s** finishing time exceeds `next_arrival` time. That **process A** is now assigned as `prev_running` process and current `time` is hopped to `next_arrival` time.

6. When time is hopped to `next_arrival` time , we check if any of the processes in `ready queue` are expired . If yes we remove them form `ready_queue` .

7. New processes are filled from `processes` queue in `ready_queue` and `next_arrival` is assigned again as in **step 3**

8. A **process P** is scheduled from modified `ready_queue` and that process is compared with `prev_running` process. If the `prev_running` process still has high priority we make it as `running` process again and add **process P** to `ready_queue` . Else the `prev_running` process was preempted by **P** and we set **P** as `running` process and `prev_running.interupted = true` .

9. Now we again have a `running` process . We check it was preempted or not by its `running.interupted` flag. If yes we mark its execution as **resumes or starts** otherwise.

10. **Step 5** is repeated again till the `processes` queue is empty . Once it is empty we assigned `next_arrival` as `INT_MAX` so that all proceess currently in `ready queue` are executed without any preemption.

Complications while programming:

1. Finding and verifiying edge cases was a challenging task .
2. Forgot to consider the deadlines for last processes ,so they were finishing execution even their deadline was expired.
3. If two processes have same priority , sheduling different processes results in slightly different logs and statistics . This sometimes makes it difficult to check program correctness

---

Graphs :

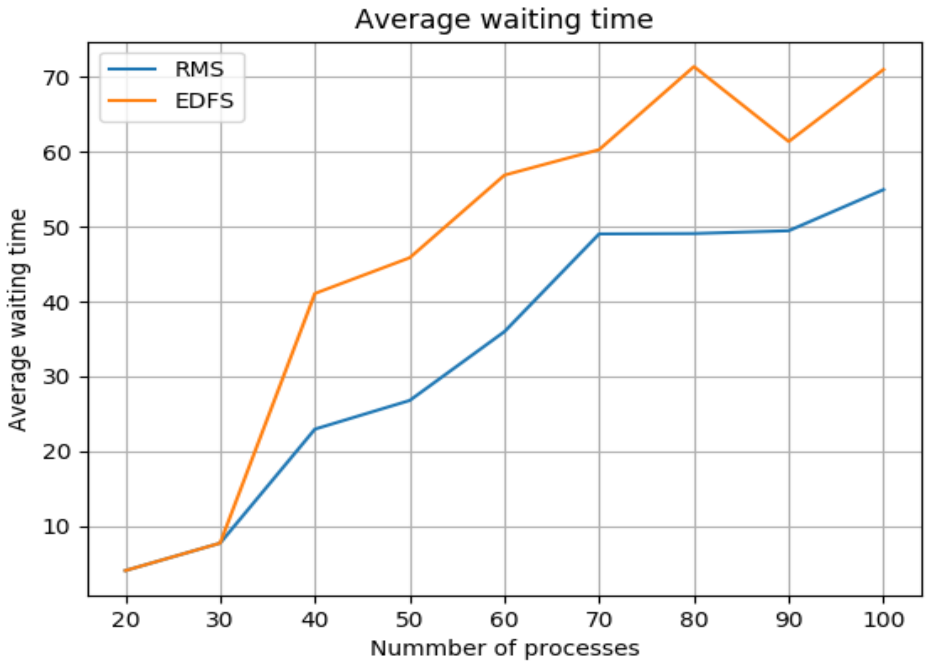
C++ script is written to generate random input file with number of process ranging from 20 (2 processes each repeating 10 times) to 100 (10 processes each repeating 10 times) with a difference of 10 processes .

For CPU utilization > 1 :

Period and processing time of each process is randomly chosen in the following ranges respectively :

$$60 <= \textbf{period} <= 110$$

$$10 <= \textbf{process\_time} <= 30$$



As evidient from the graphs **RM scheduling outperform EDF Scheduling** for large number of processes both in terms of average waiting time and number of deadlines missed. This outperforming of RMS over EDF is due to **dommino effect .**

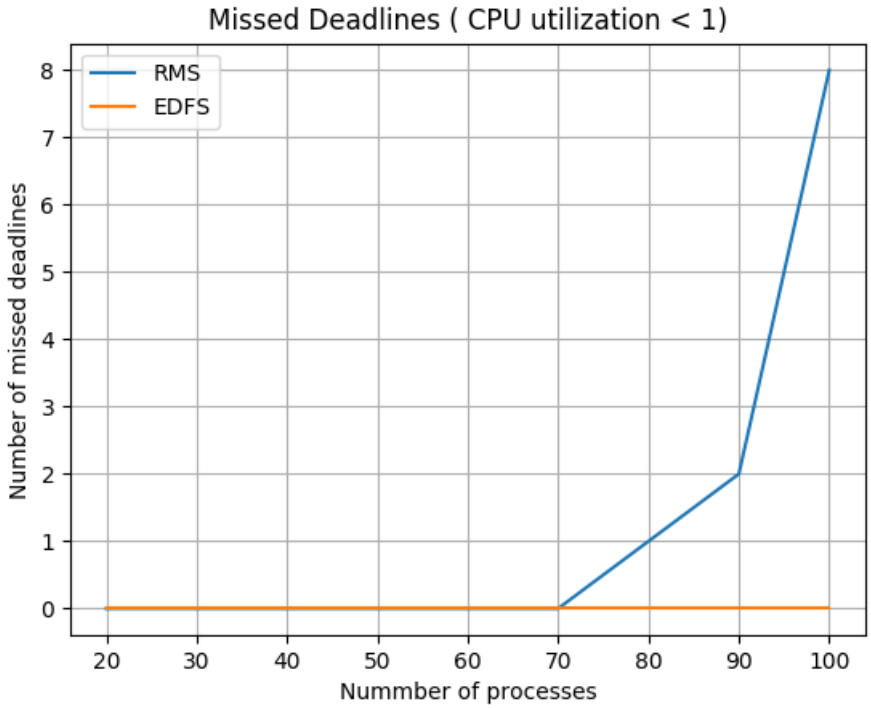
The increase in the average waiting time decreases gradually as number of processes increases whereas the number of deadlines missed rises almost monotonically with processes.

For CPU utilization < 1 :

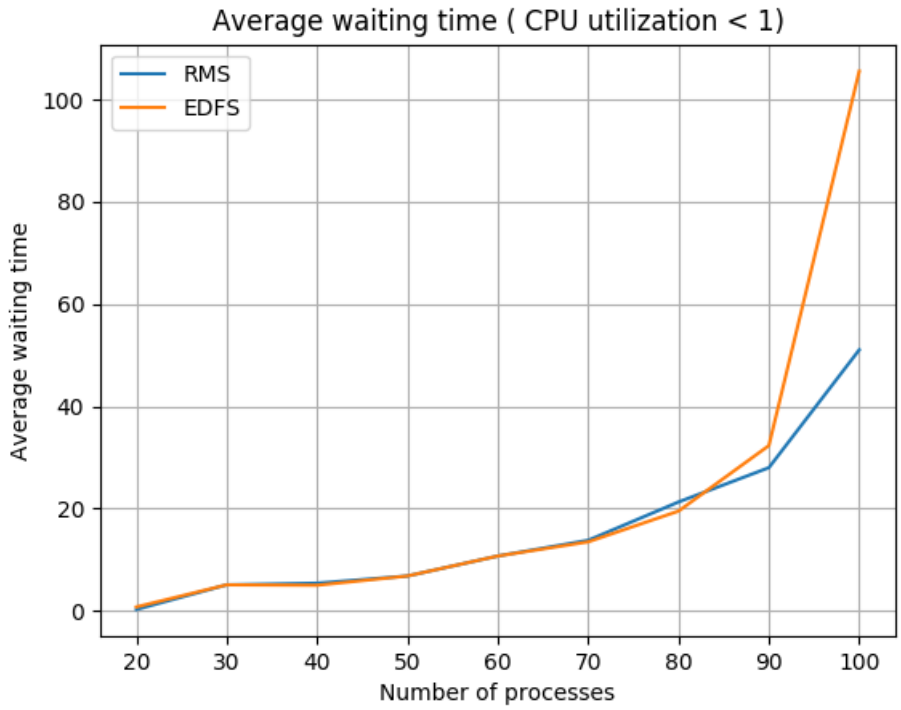
Period and processing time of each process is randomly chosen in the following ranges respectively :

$$150 <= \textbf{period} <= 250$$

$$10 <= \textbf{process\_time} <= 30$$



When CPU utilization doesn’ t exceed 1 we can see that **EDF** has proven to be a good scheduling algorithm than **RMS** in terms of number of missed deadlines . The deadlines misses in RMS is because the CPU utilization exceeds the upper bound given by



When the **RMS** process starts missing the deadlines , the proceeses are not completed and discraded from ready queue due to which that process doesn’ t have to wait anymore in the ready queue and leads to decrease in average waiting time . Whereas **EDF** is trying to schedule those processes and hence avergae waiting time increases.

Conclusion :

From the graphs we can conclude that **CPU utilization is exceeding 1 then RMS** is the best scheduling algorithm in both terms (Average waiting time and deadline miss)

If CPU utilization is less than 1 then **EDF** is better than **RMS** in terms of **number of missed deadlines** whereas **RMS** outperfroms **EDF** in terms of **average waiting time.**
