Operating Systems - 2
Spring 2019

Instructor - Dr. Sathya Peri
Jatin Chauhan - CS17BTECH11019

Design of Program:

The program has been written to simulate a scheduling dispatcher. The scheduler does real-time process scheduling using -> **Rate Monotonic Scheduling** and **Earliest Deadline First Scheduling.**

A struct called **ProcessInfo** is created which contains multiple variables, which are -> **process_id**, **processing_time**, next **deadline** of process (updated dynamically once process finishes its role in ready queue), **period**, **time left for current period**, **number of times** its has to run, **in_ready_queue** (whether it is currently in ready_queue), **expected_next_incoming_time** (the time at which it will come next in the ready queue), **running** to signify current process is running on cpu, **ready_queue_incoming_time** (when does the process enters ready_queue) and **dispatch_time** (time when process comes out of ready_queue, when -> either its deadline finishes or has to run on cpu).

The total number of processes which should run are -> ***∑ P[k] * number_of_times_it_has_to_run*** , where P[k] is kth process.

A **ready queue** is created which stores which processes should run on CPU core. The processes in ready queue are sorted based on different metrics for **RMS** and **EDFS**.

Rate Monotonic ->

This Scheduling is characterised by giving priority to processes which have the least period. This priority of a process is constant throughout execution.

Earliest Deadline First ->

This Scheduling is characterised by giving priority to processes which have the earliest deadline. This is a **dynamic-priority** scheduling as the deadline of a processes is increased by its period after every repetition.

Both the simulated programs are compared based on two metrics ->

1) **Average Waiting Time** ->

   Waiting time is defined as the time the process spends in ready queue (before executing on CPU). Average waiting time is the ->

$$\frac{\textbf{Sum of waiting times of all processes}}{\textbf{total number of processes}}$$

2) **Number of Processes missing their Deadline ->**

   This is the total number of processes which missed their deadlines, Assuming all instances of a process are different.

The better strategy will have both the metrics **low**.

Execution of Program ->

Three global variables are kept -> **total number of processes** , **number_of_failures (**which is the number of processes missing their deadlines **)** and average_waiting_time_sum -> sum of waiting times of all values(which is divided by number of processes to get final answer).

A pointer to current running process named **current_process** is created to keep track of current executing process.

All the processes are put in ready queue initially. The queue is sorted based on the specific property -> **period or earliest deadline.**

The Program loops over all processes(as stated, all instances of a process are processes themselves). A global counter is maintained which jumps over to value of time, at which the current process stops. When the current process stops, it is checked whether there was any process that should have preempted it, if so, then the global timer jumps back to the preemption time. All the processes outside ready queue, which have **expected_next_incoming_time** value less than current global timer are put in ready queue, and all process in ready queue, which have **deadline** less than current timer are removed from it and brought back, with each of them having missed their deadlines. The **number of failure processes** is increased by number of processes missing their deadlines.The process which was executing is put back in the ready queue, with its time_left parameter set to time units it should execute in current run .The ready queue is sorted and highest priority process is run. Average waiting time_sum variable ss updated to difference of dispatch time and ready queue incoming of current process.The ready_queue_incoming_time variable is then set to current value of global_timer.

If the current executing process was not preempted, then its deadline is checked. If it has crossed its deadline, then **number of failure processes** increases by one and current process is removed from ready queue and put back with its deadline and **expected_next_incoming_time** set to next values. The average_waiting_time is updated to difference of current process's deadline and its ready_queue_incoming time.

If the process has run successfully, then its **in_ready_queue** variable is set to false and its **in_ready_queue and time_left** attributes are set to their normal value and its deadline and expected_incoming_time are increased to next appropriate values. The **current_process->running** parameter is set to false. The average waiting time is again updated to difference of current process dispatch time and its ready queue incoming time.The loop runs again with the condition that **total number of processes** is greater than 0.

The comparison between both algorithms is done using Graphs.

Graphs ->

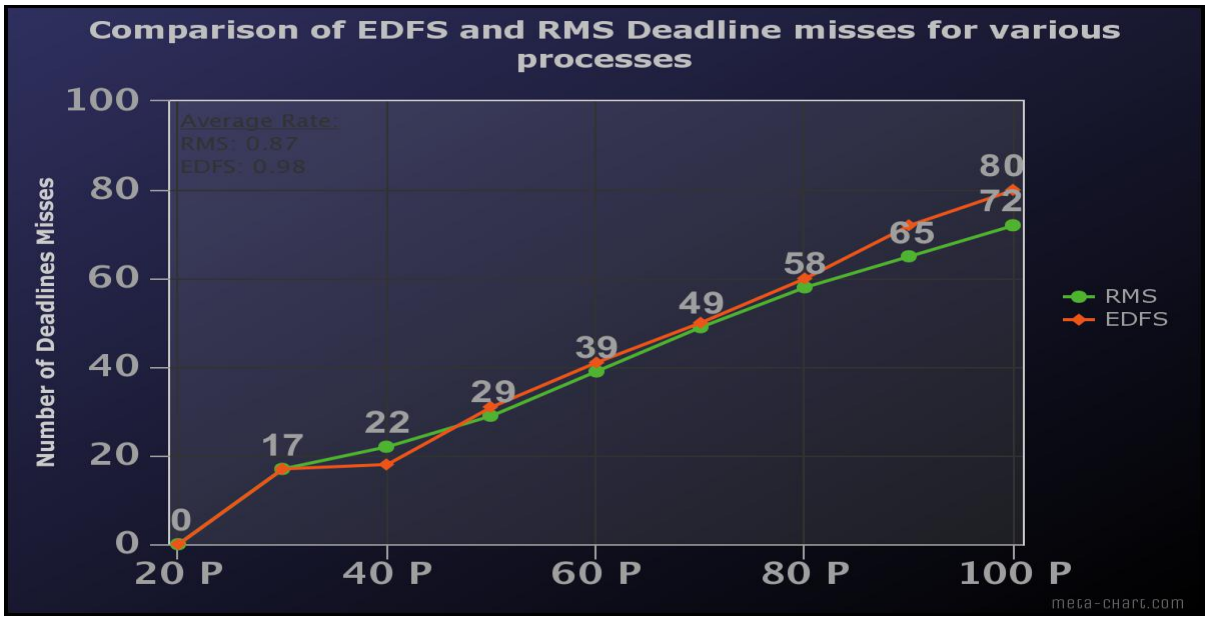
Graphs for both processes are drawn on same set of values->
Values are drawn at random with constraints->
1 <= Processing Time <=100 ; k=10 ; 10 <= Period <= 100

*1) **Number of Deadline Misses***

*Graph of **<u>number of processes missing deadlines</u>** vs **<u>number of processes run.</u>***

Here P is number of processes. (The value shown on graph in grey are for RMS, due to space constraints).

Number of Processes	Misses in RMS	Misses in EDFS
20	0	0
30	17	17
40	22	18
50	29	31
60	39	41
70	49	50
80	58	60
90	65	72
100	72	80

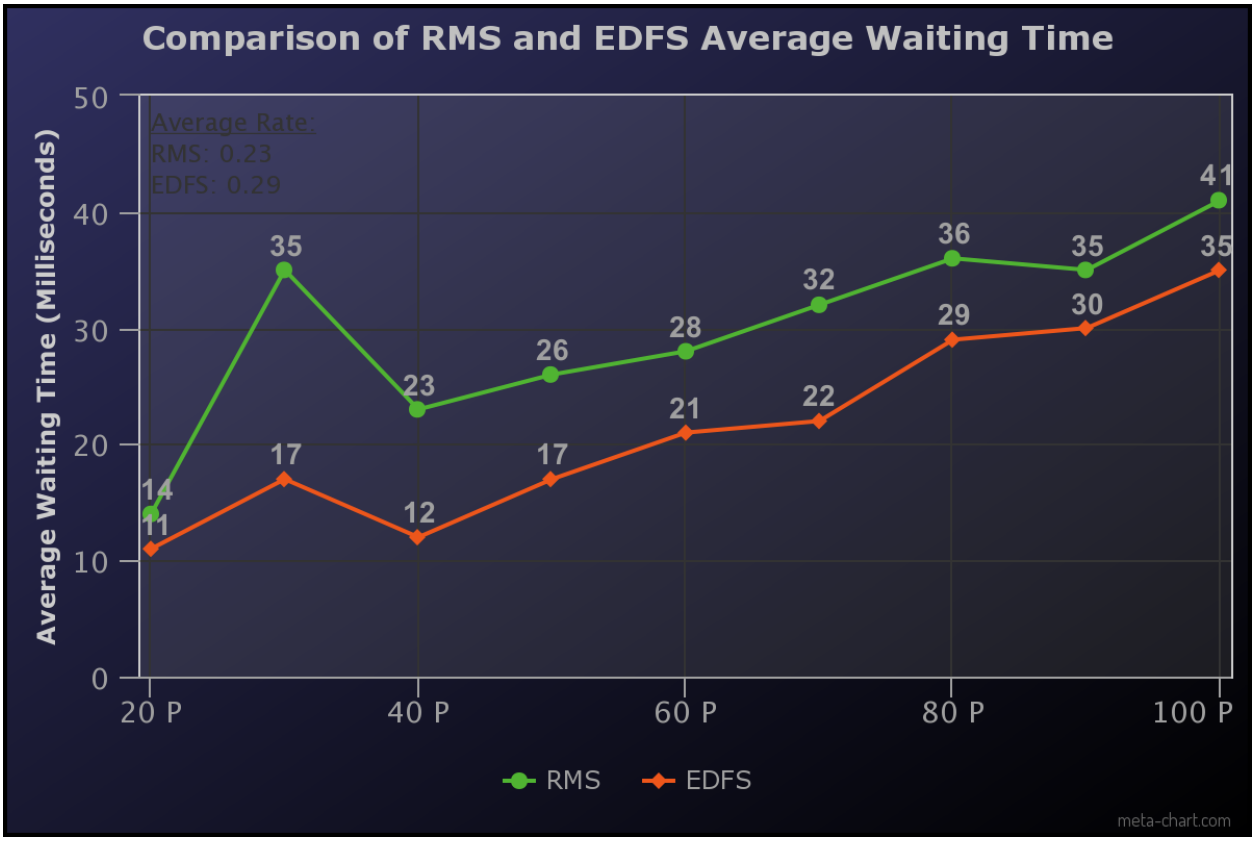


2) **Average Waiting Time**

Graph of ***<u>average waiting time vs number of processes</u>***

Here P is number of processes.

Number of Processes	Misses in RMS	Misses in EDFS
20	14	11
30	35	17
40	23	12
50	26	17
60	28	21
70	32	22
80	36	29
90	35	30
100	41	35



We can see that number of **deadline misses** in **RMS** are better.
Also as **number of Processes** approach infinity the average waiting time value for both approaches to each other (not shown here) and becomes nearly constant.
As guranteed by **RMS** algorithm through the equation -> **N * (2^(1/N) - 1),** the CPU utilization is bounded, as can be seen here and RMS can schedule better in when processes have **static priority and periodic processes**.