REPORT

DESIGN OF RATE-MONOTONIC SCHEDULING CODE:

We maintain a waiting queue of the processes that enter at a time instant *t*. We carry a variable to store the current process running. The waiting queue maintains a priority order, sorted every time a new process enters the queue at a time instant *t*. The criteria we use here to sort is least period has higher priority (the lesser the period, the higher the priority). If two processes have the same deadline, we use the criteria of the least remaining time. The priorities do not change all the time the processes run.

We send the top of this waiting queue to the processor and update the variable "current Process." We check at every point; if a new process starts requesting to enter, we update that to the waiting queue and match this with the current Process variable. If the top of the waiting queue has higher priority, we pre-empt the running process, save its state, and add it to the waiting queue. Then we send the higher priority process to the processor and update the current Process variable.

We maintain an array of deadline checkpoints of added processes. We check at this every checkpoint if a process crosses its deadline. If the process crosses its deadline, we terminate it and send the following process from the waiting queue to the processor by updating the current process variable. We even terminate the processes in the waiting queue if they miss their deadline.

This process selection does not take much time because we select from the well-maintained priority queue. The time taken to exchange a process during any pre-emption, termination of a deadline missed process, or adding a process to the pre-processor, is constant.

This update of the waiting queue is done at every point when a new process is added. And we frequently check about the missing deadline and update the current process depending on the checkpoints we stored.

DESIGN OF EARLIEST-DEADLINE FIRST CODE:

We maintain a waiting queue of the processes that enter at a time instant *t*. We carry a variable to store the current process running. The waiting queue preserves a priority order, sorted every time a new process enters the queue at a time instant *t*. The criteria we use here to sort is the earliest deadline has higher priority (nearer the deadline, higher the priority). If two processes have the same deadline, we use the criteria of the least remaining time. The priorities change every time the new process enters, depending on the earliest deadline.

We send the top of this waiting queue to the processor and update the variable "current Process." We check at every point; if a new process starts requesting to enter, we update that to the waiting queue and match this with the current Process variable. If the top of the waiting queue has higher priority, we pre-empt the running process, save its state, and add it to the waiting queue. Then we send the higher priority process to the processor and update the current Process variable.

We maintain an array of deadline checkpoints of added processes. We check at this every checkpoint if a process crosses its deadline. If the process crosses its deadline, we terminate it and send the following process from the waiting queue to the processor by updating the current process variable. We even terminate the processes in the waiting queue if they miss their deadline.

This process selection does not take much time because we select from the well-maintained priority queue. The time taken to exchange a process during any pre-emption, termination of a deadline missed process, or adding a process to the pre-processor, is constant.

We do these updates on the waiting queue when a new process comes in. And we frequently check about the missing deadline and update the current process depending on the checkpoints we stored.

The difference between the two codes is the priority function used to sort the waiting queue.

The complications occurred while pre-empting. When a new process enters, we have two options: to pre-empt the current process if the new process entered has higher priority, and two is not to pre-empt if the process entered is less prior. Handling these two cases was a bit confusing.

One more while calculating the waiting times. This made me create a new struct to store every process's waiting time.

We will compare the performance of these algorithms using the following data:

Process 1: processing time=20; deadline=90; period=90 joined the system at time 0

Process 2: processing time=30; deadline=250; period=250 joined the system at time 0

Process 3: processing time=70; deadline=370; period=370 joined the system at time 0

Process 4: processing time=50; deadline=330; period=330 joined the system at time 0

Process 5: processing time=125; deadline=2000; period=2000 joined the system at time 0

Process 6: processing time=35; deadline=80; period=80 joined the system at time 0

Process 7: processing time=25; deadline=50; period=50 joined the system at time 0

Process 8: processing time=20; deadline=100; period=100 joined the system at time 0

Process 9: processing time=25; deadline=40; period=40 joined the system at time 0

Process 10: processing time=30; deadline=75; period=75 joined the system at time

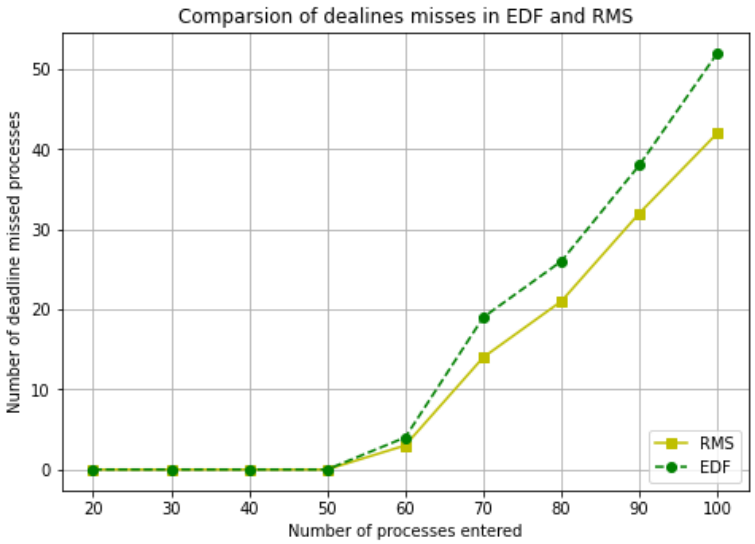
Each process above repeats at every ten periods and we terminate a process if it misses a deadline.

COMPARISON OF PERFORMANCE:

(a) Varying the number of threads:

x-axis: number of processes ranging from 20 (2 processes each repeating ten times) to 100 (10 processes each repeating ten times) in the increments of 10.

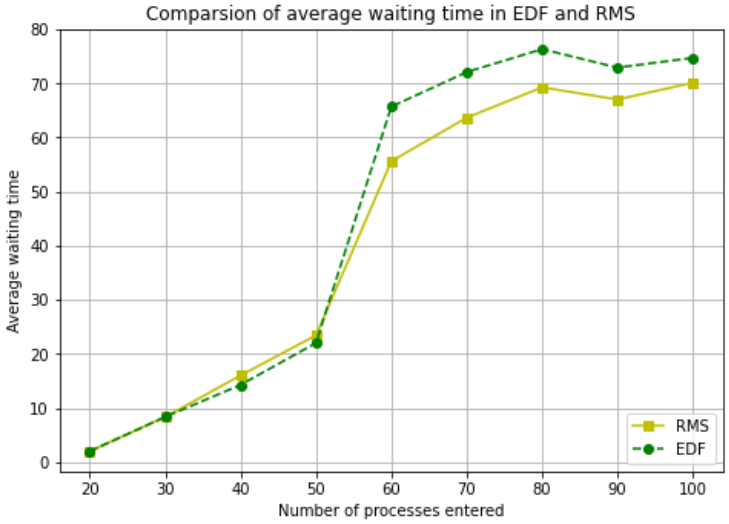
y-axis: the number of processes that miss the deadline



(b) Varying the input array size:

x-axis: number of processes ranging from 20 (2 processes each repeating ten times) to 100 (10 processes each repeating ten times) in the increments of 10.

y-axis: the average waiting time



The EDF has higher deadline misses than RMS because of the context switching and selection time, which was wasted.