REPORT

SRI HARI. M
CS19BTECH 11039

Aim: Simulation of Rate monotonic and Earliest deadline first scheduling algorithms.

Code Design:

In common, the first and foremost thing is to include the header files in the source code. All the useful header files are included in both the source codes. Next, we need to have an object for holding the process and all its details in one place. For that, I created a class named **process_details** in which all the properties related to the process are stored. It has members like process id, processing time, period, deadline misses, the number of times a process should execute and other useful variables like waiting time and is_prempted which helps in scheduling the processes or for further analysis of waiting times or deadline misses. Using the fstream in CPP, the program takes all the required inputs. After taking input for a process, I made a container for all the processes to be held in a single place using vector notation. The process carried out till fetching input is the same for both the source codes. Hereby, calling upon the respective schedulers, the set of processes gets scheduled respectively to the scheduler called. Let's see the implementation of both the scheduling algorithms. The input file is hardcoded in the source codes with the default file name "inp-params.txt". The process of checking deadlines and executing the process carries the same way in both the schedulers, so these are written after clearly explaining the process of adding into the queue as this document is getting extremely large.

Rate monotonic scheduler:

The rate monotonic scheduler schedules in a way that the periods of processes are considered for the priority. The lesser the period, the higher the priority and hence it executes accordingly. For that, we need a processing queue in which the processes are added accordingly and then execute correspondingly to the condition. For that, I've created a class named **schedule_processes** in which it has members like a queue, a method to add processes in the queue, a method to check if the present queue has processes that have deadline misses and a method to schedule the processes in the queue. First, the main function calls the scheduler and then it sorts the processes according to the priority and then calls the method to add processes for every real-time unit. This is done using a loop.

- **Adding process in the queue:**

In this, for every instance of time, a loop adds the process if the time is considered to be the multiple of the respective process's period. Now upon calling the add process function, a queue that is already created comes into existence. This function takes the input of a process detail and a file to print if in case any process gets preempted. Conditions for the ordering of processes:

1. If the queue is empty, push the process into the queue.
2. If the queue is not empty, then using a loop, iterate over the present existing queue and then compare accordingly to the periods and decide the priority.
3. Here in this loop, it has 3 situations, the process's period is lesser, equal to and greater than the iterator's process's period.
4. If the present iterator process has a period greater than the coming process's period, then note the entry time and push it into the queue according to the priority.

5. Else if, we face the situation, where the periods are equal which results in a tie, check the process id of both the processes and insert it into the queue accordingly as the lesser the process id more the priority.
6. Else we come to the condition where the process's period is greater than the iterator's process's period. Here we don't have to do anything, just iterating the queue makes the process compare to the next iterated process. But if the iterator comes to the last, then add the process at the back of the queue.
7. Here in this, we get to know if a process is going to be preempted. If a process gets preempted, then change the process's is_prempted status, which will be helpful further.
8. By the end of this function, a process gets into the queue according to the priority.

Earliest deadline first scheduler:

The earliest deadline first scheduler, schedules in a way that the next deadlines are considered for the priority in each process. The lesser the next deadline, the higher the priority and hence it executes accordingly. For that, we need a processing queue in which the processes are added accordingly and then execute correspondingly to the condition. For that, I've created a class named **schedule_processes** in which it has members like a queue, a method to add processes in the queue, a method to check if the processes in the queue have deadline misses and a method to execute the processes in the queue. First, the main function calls the scheduler and then it sorts the processes according to the priority and then calls the method to add processes for every real-time unit. This is done using a loop.

- **Adding process in the queue:**

In this, for every instance of time, a loop adds the process if the time is considered to be the multiple of the respective process's next deadline. Now upon calling the add process function, a queue that is already created comes into existence. Conditions for the ordering of processes:

1. If the queue is empty, push the process into the queue.
2. If the queue is not empty, then using a loop, iterate over the present existing queue and then compare accordingly to the next deadlines and decide the priority.
3. Here in this loop, it has 3 situations, the process's next deadline is lesser, equal to and greater than the iterator's process's period.
4. If the present iterator process has a deadline greater than the coming process's deadline, then note the entry time and push it into the queue according to the priority.
5. Else if, we face the situation, where the next deadlines are equal which results in a tie, check the process id of both the processes and insert it into the queue accordingly as the lesser the process id more the priority.
6. Else we come to the condition where the process's deadline is greater than the iterator's process's deadline. Here we don't have to do anything, just iterating the queue makes the process compare to the next iterated process. But if the iterator comes to the last, then add the process at the back of the queue.
7. Here in this, we get to know if a process is going to be preempted. If a process gets preempted, then change the process's is_prempted status, which will be helpful further.
8. By the end of this function, a process gets into the queue according to the priority.

Now, we've seen the add process function which adds a process into the processing queue for both the schedulers. The following functions are common for both schedulers.

- **Checking the deadline misses:**

As the queue contains all the processes, now we need to track the processes which miss their deadlines and remove the process from the process queue as soon as it's status of deadline miss is identified.

1. Iterate over the queue, if the sum of the process's remaining time to execute and real-time is greater than the respective period, then the process is going to miss its deadline.
2. We are not sure that, if a process is going to miss its deadline, the next process also misses its deadline. For that, the check deadline function is called recursively and a check is made for every process in the processing queue.
3. If a deadline miss is encountered, then update the count of deadline misses in the original details. Also, make sure that the waiting count for a process with a deadline miss is updated with the sum of its period and removed from the queue.

- **Executing the processes in the processing queue:**

Apart from loading the processes into a process queue, checking for deadline misses, the next job to be done is to execute the process according to the scheduling principle. In this section of code, as the processes are already loaded into the queue, the execution proceeds in the following way with the conditions.

1. If the queue is empty, note the times in a specific way which denotes the time in which the CPU is idle.
2. If the queue is not empty, then pull out the first process in the processing queue, but don't pop it from the queue as we don't

know if it’s going to be preempted or going to be executed completely by then.

3. If a process’s remaining time to be executed is 0, and if the sum of the process's entry time and it’s period is greater than the real-time, then the process is going to be completely executed.
4. After successfully noting the properties, then pop the process from the queue and return the control to the scheduler which further calls the add process function again.

And then after successfully executing all the processes, the remaining counts for the processes approach 0 and the scheduling of the process completes. Next, a function is used to print all the statistics in the log file and statistics file.

Code Execution for RMS:

- Fetching the inputs of all the processes and subsequently pushing it in a container.
- Calling the RMS scheduler bypassing these processes as parameters.
- The scheduler then calls the methods
    1. Add process

    Adds all the eligible processes into the processing queue
    2. Check for deadline misses

    Checks all the process and removes the process from the processing queue if a deadline miss is encountered.
    3. Execute/schedule

    Executes the processes one by one.

    And in a loop, these execute one after another till all the processes are executed for their respective repeat counts.
- Calling the print stats function to print all the logs and statistics sequentially for scheduling the processes.
- End of the program.

Code Execution for EDF:

- Fetching the inputs of all the processes and subsequently pushing it in a container.
- Calling the EDF scheduler bypassing these processes as parameters.
- The scheduler then calls the methods
    1. Add process
       Adds all the eligible processes into the processing queue
    2. Check deadline misses
    3. execute/schedule
       Executes the processes one by one.

  And in a loop, these execute one after another till all the processes are executed for their respective repeat counts.
- Calling the print stats function to print all the logs and statistics sequentially for scheduling the processes.
- End of the program.

For programs with context switch:

The implementation of the programs involving the context switch is the same as that of normal except at switching the processes. A context switch is a time that the scheduler takes to switch the processes of execution. It involves taking off the process and adding a new process. It happens when a process is newly added into the execution unit, i.e either if a process has started execution or a process resumed its execution after a preemption. So, at the block of code determining the resumption or starting execution of a process, a context switch is added to the real time. Now everything works fine. For judging the CPU idle status extra measures are taken as the context switches add a float or double values where the real-time is increased by a unit at every instance.

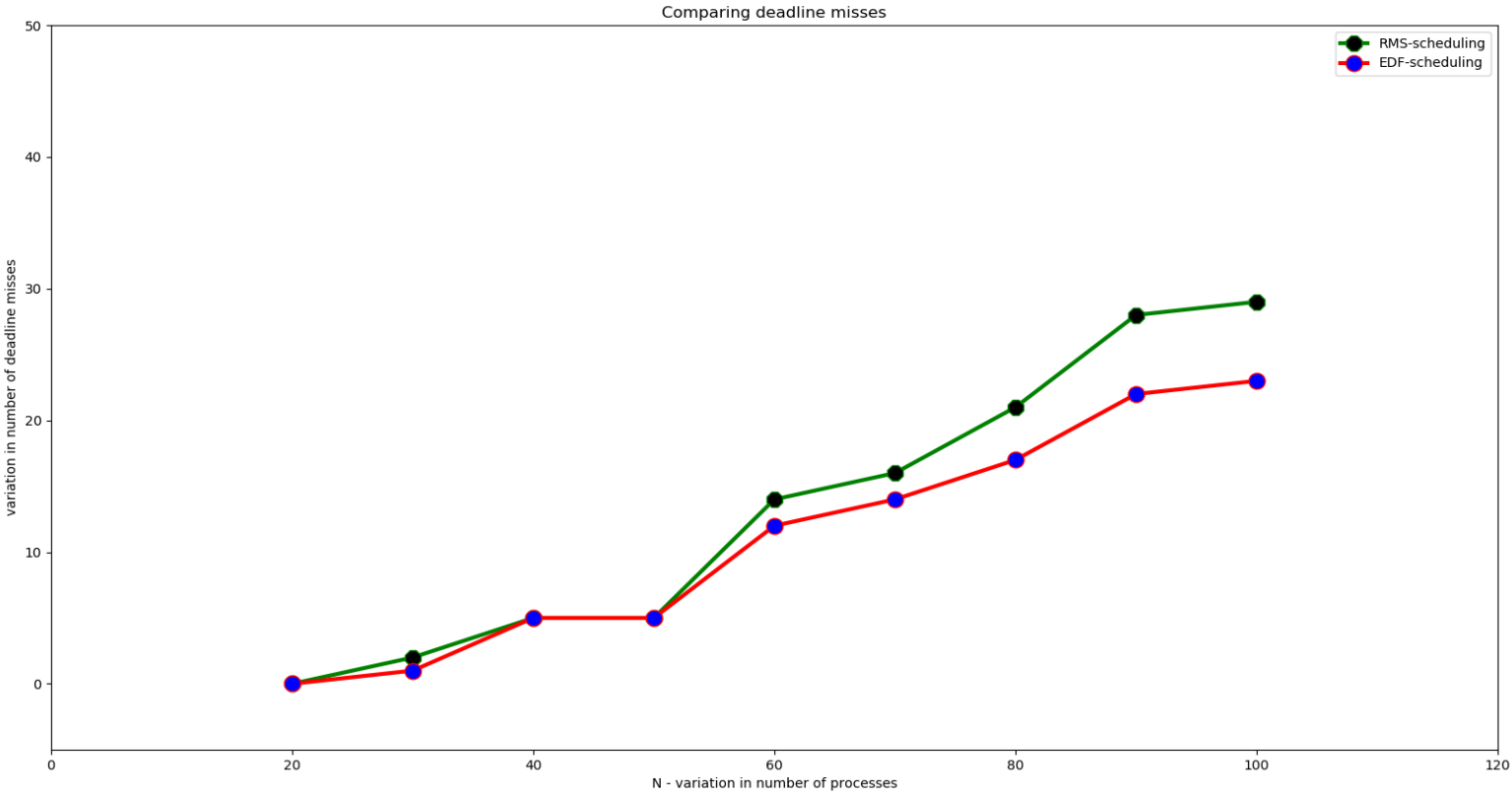
Note: The values of processing times and periods are to be in integer format. As it was not clearly mentioned, this was developed considering that the values are integers.

Challenges faced:

There are many challenges faced while building codes. Some of them are here:

1. While building the method of add process which adds the eligible process into the processing queue.
    - Need to check many edge cases
    - Altering the code to make changes in the method to suit all the cases.
2. While checking the deadline misses, and removing the process, we need to be very careful while removing the process from the queue.
3. The building of method schedule, which executes the processes
    - Should be very careful with the note-taking of properties of the process.
    - Altering the variables of the processes in the process queue or the original details of the process whenever needed.
    - Popping a process from the process queue
    - Updating the waiting time, remaining time of a process.
4. Properly maintaining the strict values of repeat counts of a process and checking if a process needs to add or remove and updating the same in the details.
5. For programs with context switches, some minute extra measures are to be taken for actual values. Like type casting adding or subtracting a unit time wherever needed.
6. Not having a clearer picture of sending the parameters with reference(non C type).

Analysis of algorithms using graphs:

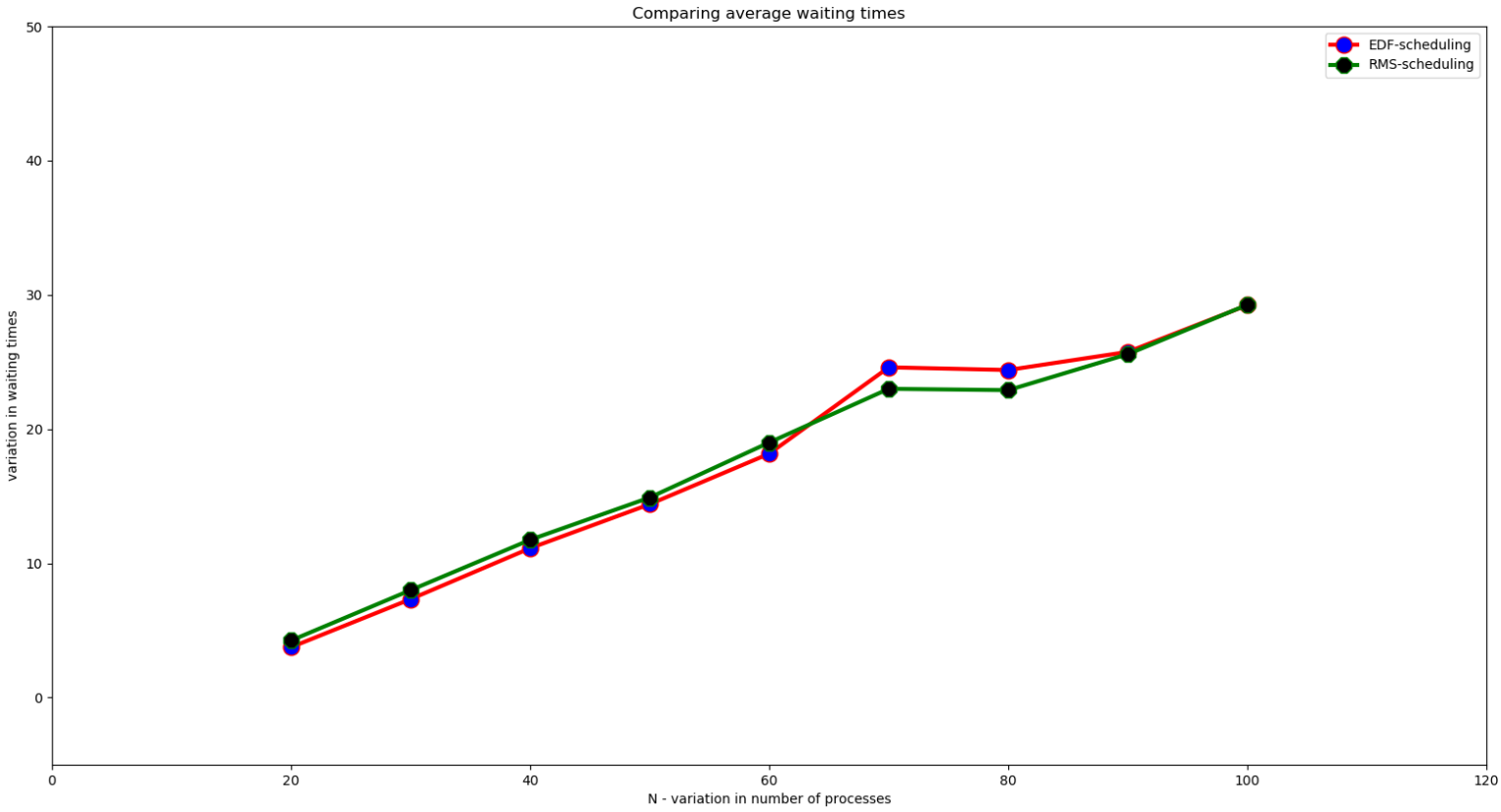


Deadline misses:

The above graph is plotted with the number of processes in the X-axis vs the number of deadlines misses met for all the processes.

In the above graph, the curve in green colour represents the RMS scheduling while the curve in red colour represents the EDF scheduling. It is clear from the graph that the RMS scheduling curve has more slope and is increasing when compared to the EDF scheduling curve. Initially for a lower number of processes the deadline misses are comparatively equal for both the algorithms whereas with an increase in the number of processes the deadline misses are increasing in the RMS scheduling algorithm in a very

rapid manner while the deadline misses occurred with the EDF scheduling algorithm is also increasing but very less compared to RMS. So, the EDF scheduling algorithm is a way bit more efficient than the RMS scheduling algorithm in terms of deadline misses.



Average waiting time:

The above graph is plotted with the number of processes on the X-axis and the variation in waiting times on the Y-axis.

Again similarly to the first graph the curve in red colour represents the EDF scheduling while the curve in green colour represents the RMS scheduling. From the graph, one can find that there are no large differences between the waiting times of

deadlines. On a small scale of processes, the average waiting times are almost equal for both the process with utmost 1 milliseconds of difference. But later on for the given set of processes, the EDF scheduler takes some extra time in order of milliseconds and again comes back to its state. Maybe that is because the ordering of the processes is random. And moreover, the process whose deadline is missed is not scheduled later according to the problem statement. However one can see that in the last, the slope of the EDF scheduling algorithm is decreasing while compared to the RMS scheduling algorithm's curve. So here both the algorithms are almost equally efficient in terms of average waiting times, with EDF having a little more efficient.

Moreover, if we consider the scheduling of processes even if it missed its deadlines, then it would be obvious that the EDF scheduling algorithm will be more efficient than the RMS scheduling algorithm in any case. So in the final, EDF scheduling algorithm is more efficient from the observations drawn from the graph.

Processes Input used for plotting above graphs:

```
10
1    15    60    10
2    35    70    10
3    5     15    10
4    10    20    10
5    5     80    10
6    15    30    10
7    12    150   10
8    2     20    10
9    7     50    10
10   15    350   10
```

THE END!