# Implementing Rate-Monotonic Scheduling & Earliest Deadline First Scheduling through Discrete Event Simulation

**Ananya Mantravadi**
**CS19B1004**

This assignment implements programs to simulate the Rate-Monotonic & Earliest Deadline First (EDF) scheduling algorithms.

The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption. Upon entering the system, each periodic task is assigned a priority inversely based on its period.

Earliest-deadline-first (EDF) scheduling assigns priorities dynamically according to deadline. This differs from RMS in that priorities are not fixed.

I have created a class whose objects represent each process that will enter the system. It contains information about the process like process id, period, processing time, number of times it repeats, etc.

```
class Process{
public:
        int pid;
        long p_time;
        long period;
        long executed_till;
        long arrival;
        long deadline;
        double waiting_sum;
        int repeats;
        int runtimes_left;
        Process(){};
        ~Process(){};
};
```

The processes in RMS are given priorities by sorting them based on their periods - shorter periods are given higher priority. To break ties between processes that have the same period, we look at the process ids.

```
bool compare(Process &a , Process &b)
{       if(a.period!=b.period)
                return (a.period < b.period);
        else
                return (a.pid < b.pid);
};
```

The rest of the implementation includes running a for loop from time = 0 to maximum time (as obtained from the maximum of period*repeats of all processes). At every time instant we check conditions as to whether the CPU is running or not or when a process undergoes one of the following events:

1. Starts its execution - a process with the current highest priority which has the value (executed_till) == 0
2. Finishes its execution - a process which has finished executing when (executed_till) == (period)
3. Is preempted by another process - when another process with higher priority starts executing and the current process is preempted by it.
4. Resumes its execution - when a process that was preempted earlier continues to execute when it is again given the highest priority.
5. Misses its deadline - when a process is yet to finish its execution but it is past its deadline for the period, or when we know that process would not be able to meet its deadline and kill the process.

For each of the above events, we maintain a log (RMS-Log.txt and EDF-Log.txt), and the stats of the execution is maintained in (RM-Stats.txt and EDF-Stats.txt)

The average waiting time of a process is calculated as (Sum of waiting time of all the rounds) / (the number of times process repeats)

The average waiting time of all the process is calculated as (Sum of average waiting time of all the Processes) / (Total number of processes)

## COMPLICATIONS

- It is not always possible to know beforehand if a process will miss its deadline beforehand since there is a chance of some other process with higher priority to preempt this currently executing process. So, a process might be executed for a while, then get preempted, and then might not be able to meet its deadline as it can't then finish its processing time after resuming its execution.
- The waiting time for a process might not be entirely accurate - like in the above case, if a process executed for a while before we kill it after understanding that it would never be able to finish its execution before its deadline, we still add the entire period of the process to the waiting time sum instead of the original definition of (execution time starting - arrival time).
- Since we are allowing for the processes to be skipped if the scheduler knows that it is going to miss its deadline for both RMS and EDF algorithms, the results obtained might not be identical to the actual algorithms as prescribed by the reference book. The number of deadlines missed would be lesser than or equal to the actual algorithms as compared to this implementation.

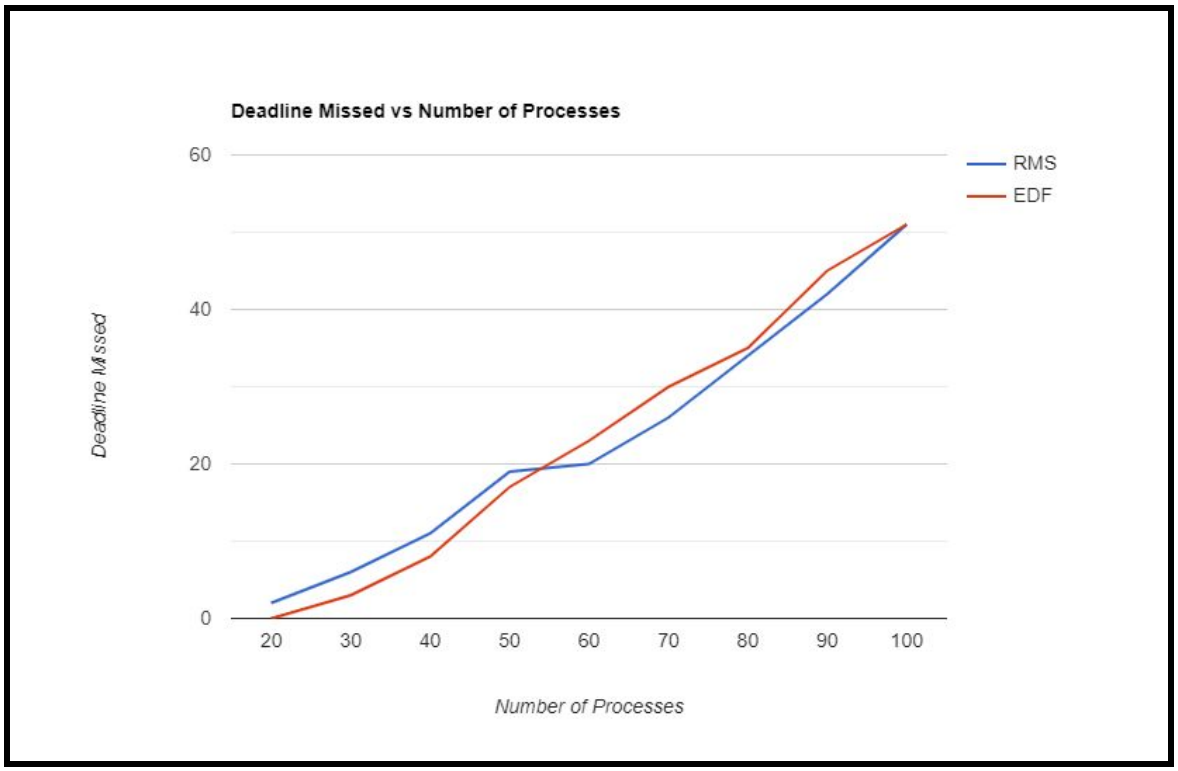
## CONTEXT-SWITCHING TIME

- Given time incurred in selecting and switching to the next process is 10 microseconds.
- Whenever the CPU is allocated to a process for it to start executing or resume executing, it involves the CPU selecting the process from the queue and then start its processing. This will account for 20 microseconds.
- Whenever a process is currently running and we have a process with higher priority available to run, we preempt the current process. This will account for 10 microseconds.
- We add the time taken in the above two points when we want to calculate the time incurred in context switching. The following lines of code have to be uncommented:

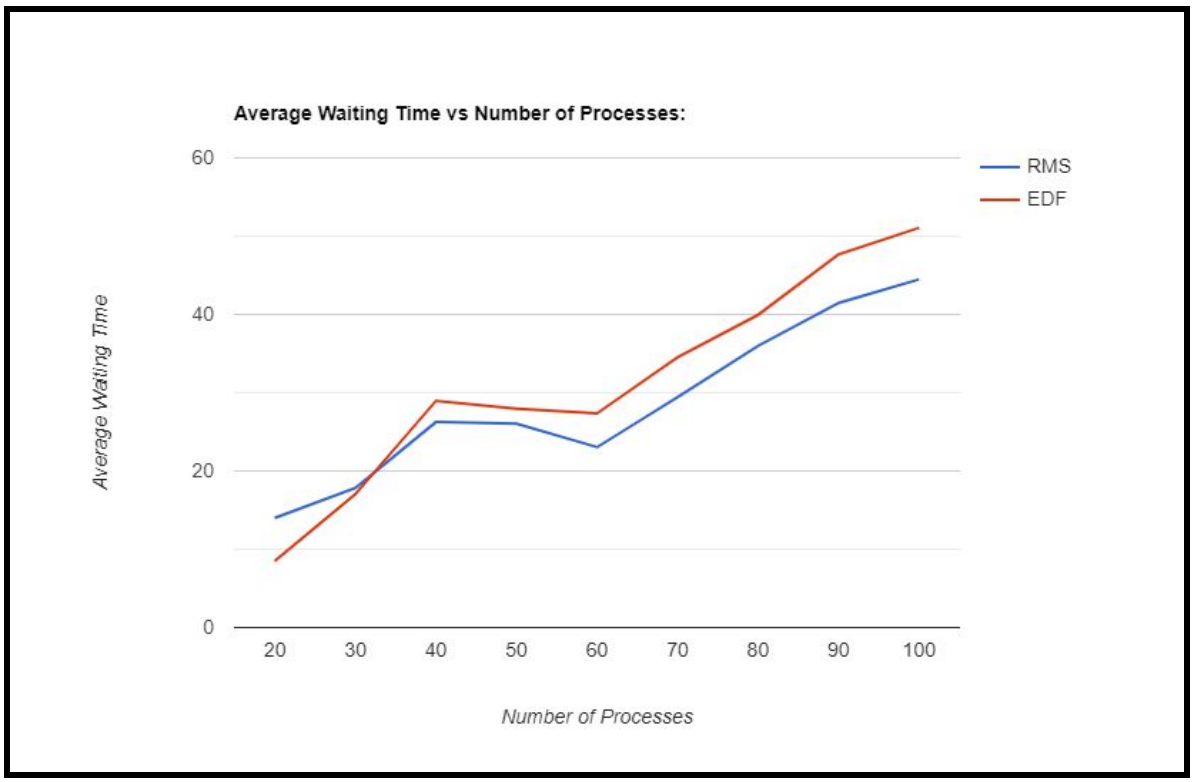
```
79//                                    context_switch += 0.01;

96//                                    context_switch += 0.02;

124//                                   context_switch += 0.02;
```

```
157//     out2<<"Average Waiting Time for all processes with context
switch time: "<<(avg_time+context_switch)/(double)n<<"
Milliseconds"<<endl;
```

## GRAPH 1 - Deadline Missed vs Number of Processes



## GRAPH 2 - Average Waiting Time vs Number of Processes




## CONCLUSION

As the number of processes increases, the number of deadlines missed and average waiting time increases. For higher values of the number of processes that enter the system, EDF scheduling leads to slightly more deadline misses. Also, the average waiting time is more for EDF scheduling. We can thus see that RMS performs marginally better, probably since EDF assigns priorities dynamically which is generally more costly.