Operating Systems-II
Report-Asgn-2

Task:

To implement a program to simulate the Rate-Monotonic & Earliest Deadline First (EDF) scheduling algorithms. Then comparing the average waiting time and deadlines missed of both algorithms. Implement both the algorithms in C++ using discrete event simulation.

Algorithms:

Rate Monotonic Scheduling (RMS) : Rate Monotonic Scheduling is characterised by giving priority to processes which have the least period. This is static-priority scheduling as the period of a process is constant throughout execution.

Earliest Deadline First (EDF): Earliest Deadline First Scheduling is characterised by giving priority to processes which have the earliest deadline. This is a dynamic-priority scheduling as the deadline of a process is increased by its period after every repetition.

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 the ready queue waiting to be dispatched.

Design Decisions:

1. I used a Priority queue which reduces the time complexity of the algorithm to a greater extent.

    a. Choosing Highest priority element - O(1)
    b. Inserting a new element in queue -O(log(n))
    c. Next process which will enter ready queue - O(log(n))
    d. Helps in Discrete Event simulation

2. Data type for priority queue - Using a pair of integers.(`array<int, 2>`) This helps us by providing a direct mapping to the elements in the priority queue.

3. Logging extra information - All logs related to missing deadlines are logged.

4. Using double data type for preventing overflow.

Submitted by: Akash Tadwai , ES18BTECH11019.

Approach and Implementation:

1. To achieve our above mentioned goal we define a struct PCB which contains some standard parameters of Process Control Block like process id , period , CPU burst time , deadline , number of repetitions etc.

```
typedef struct { // PCB block
int  pid; //Process ID
int burst_time; //burst time of the process
int period; // period of the process
int next_deadline; // The next deadline at which the
process is to be completed
double waiting_time; //The waiting time of the process in
all rounds of it's execution
double start_time; // The start time of the process
 bool operator()(const tuple &a, const tuple &b) { //
High priority is given to process with low period and if
periods are same
   if (a[1] == b[1])                                  // The
priority is given to process with lower pid
     return a[0] > b[0];
   return a[1] > b[1];
 }
} PCB;
```

2. I have implemented the algorithms using two priority queues which will help us in our discrete event simulation.

```
#define tuple array<int, 2>
priority_queue<tuple, vector<tuple>, PCB> ready_q, next_processq;
```

3. Events to be considered are when a process:
   a. starts its execution.
   b. finishes its execution.
   c. is preempted by another process.
   d. resumes its execution.

4. The first priority queue is used as a ready-queue(a queue where the PCBs are

Submitted by: Akash Tadwai , ES18BTECH11019.

waiting to get executed on the CPU) and the second priority queue(next-process-queue) is used to know which PCB will enter the ready queue in the near future.

5. Each priority_queue contains a pair of Integers.
   a. In **ready_queue**:
      i. arr[0] represents the process id
      ii. arr[1] represents the priority of the process.
   b. In **next_process_queue**:
      i. arr[0] represents the process id
      ii. arr[1] represents the arrival time of the process in the ready queue.

6. We will run a while loop till all the processes are over, that is till both the priority queues become empty.

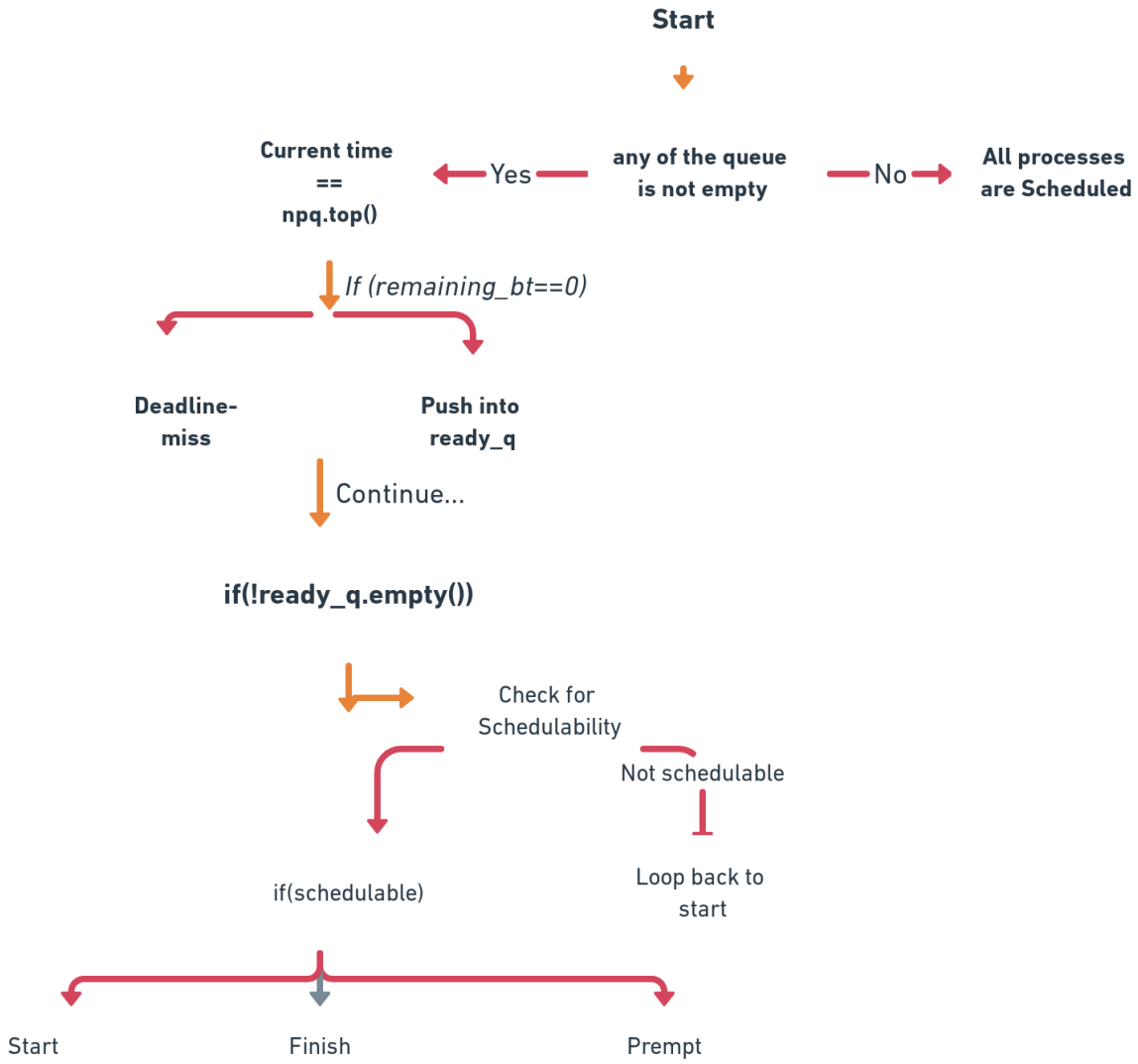
7. At any given time t, if a process enters the ready_queue it will
   a. Complete its CPU BURST
   b. Get preempted by another process
   c. Miss its deadline

8. The above three conditions are mutually exclusive and exhaustive. So they can be implemented easily.

9. Implementation of the above three conditions very slightly with change in algorithm. In RMS we push `{pid, process[pid].period}` into the queue. Whereas we push `{pid, process[pid].next_deadline}` into the queue.

The brief flow of the algorithm is given in the flowchart below.

Submitted by: Akash Tadwai , ES18BTECH11019.



Complications:

1. Floating point operations get involved if we try to include very small time (time required for Context Switch). So unpredictable behaviour may be observed.

2. Finding Edge Cases and verifying them for Context switch case.

Submitted by: Akash Tadwai , ES18BTECH11019.

Output Analysis:

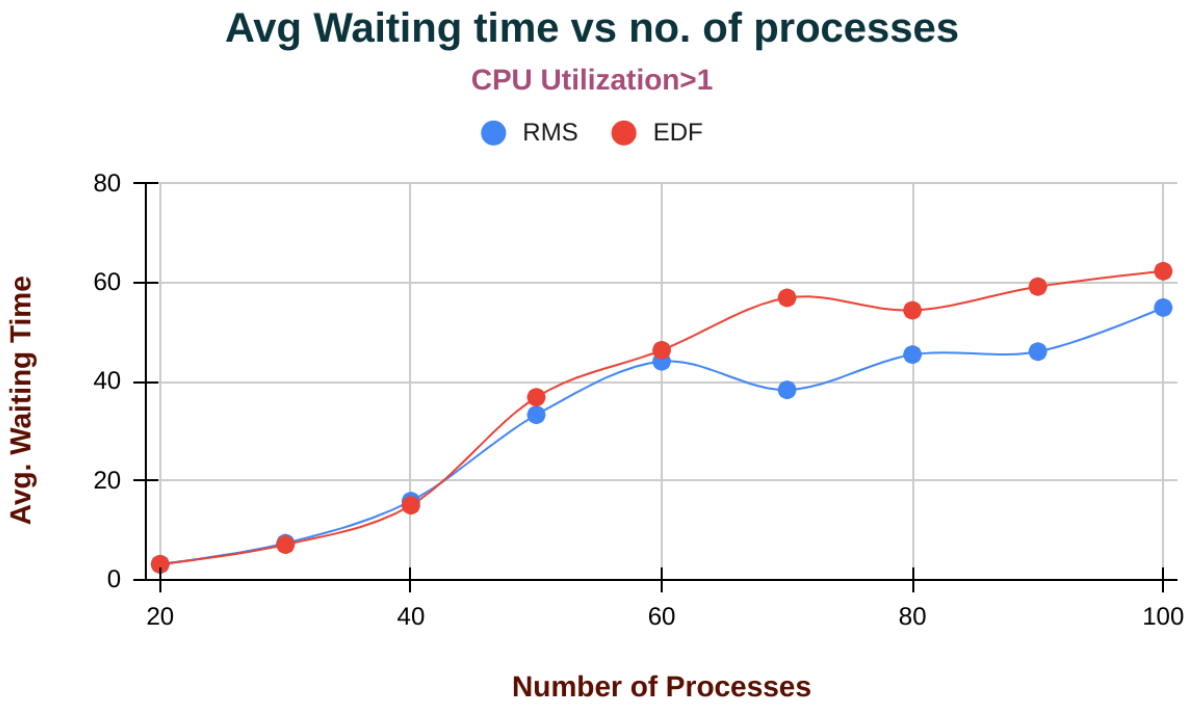
(Plots are available at: Google Sheets)

The tests cases were generated by a C++ Script file with CPU utilisation both > and <1. Then the corresponding waiting times and average times were plotted.

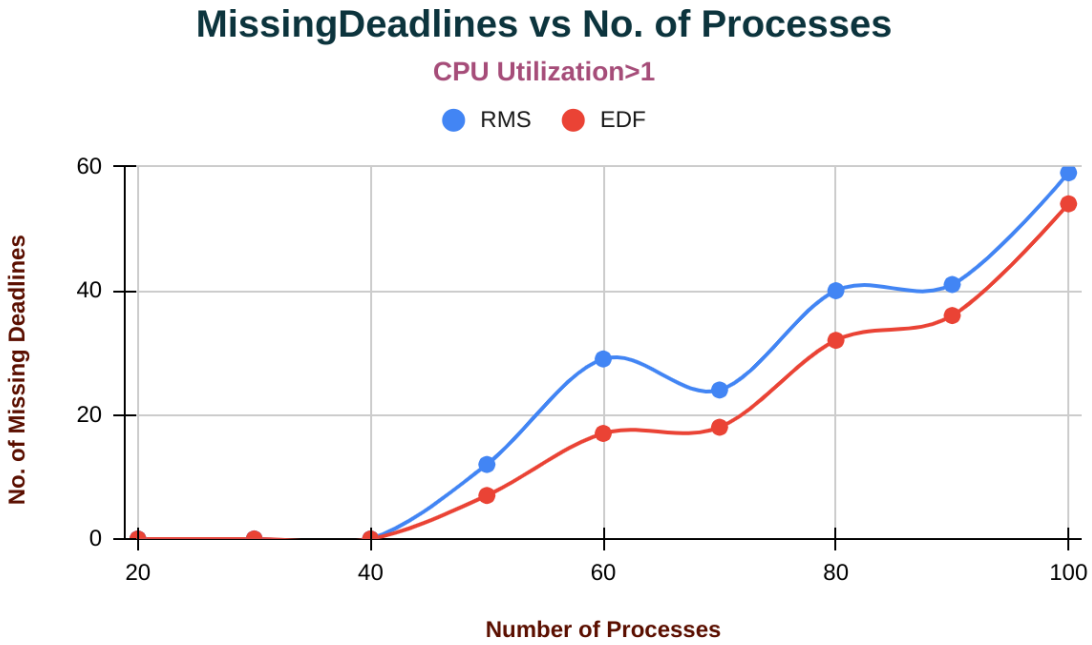
For CPU Utilization > 1:

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

60 <= period <= 110
10 <= process_time <= 30



Submitted by: Akash Tadwai , ES18BTECH11019.



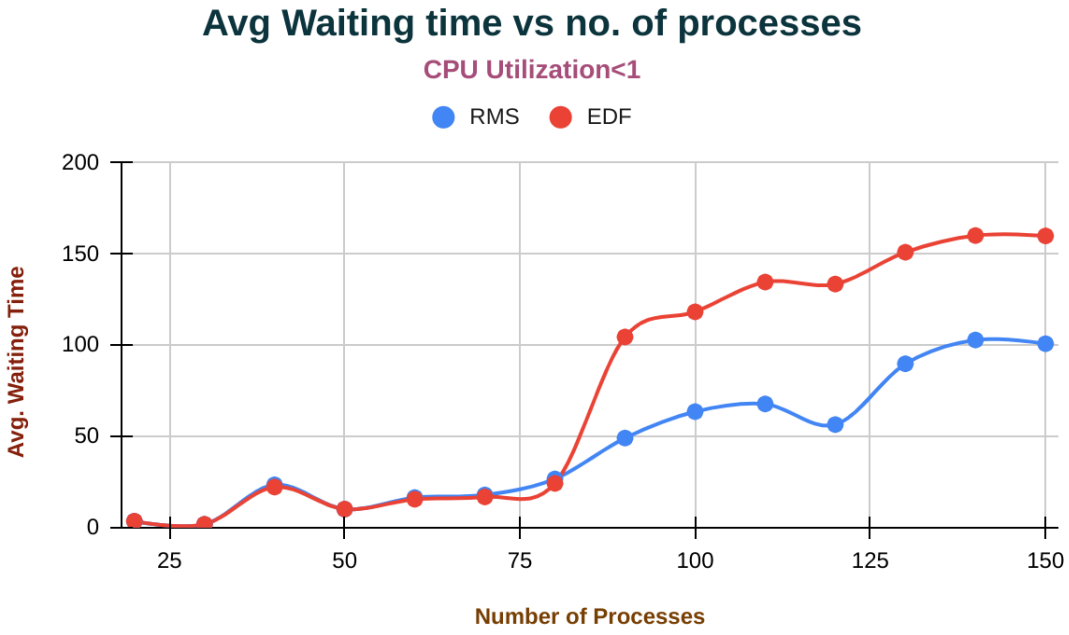
1. From the above graph we can see that as the number of processes increases the average waiting time increases. ***RMS performs better*** in terms of average waiting time.
2. From the above graph we can see that as the number of processes increases the number of deadlines missed increases. **EDF performs** better in terms of missed deadlines.

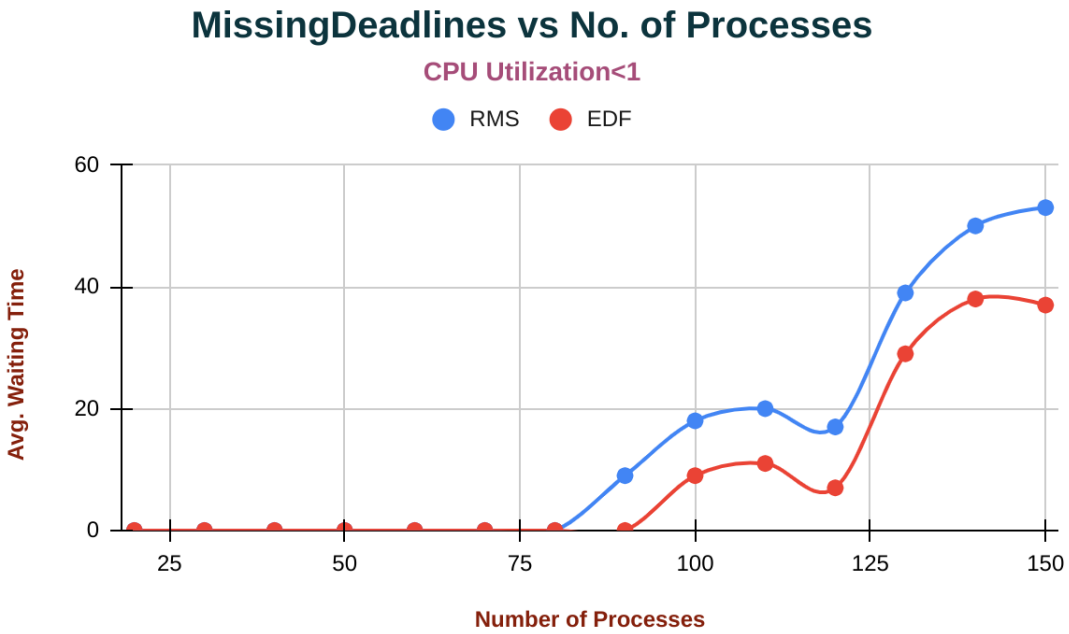
For CPU Utilization < 1 :

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

150 <= period <= 250
10 <= process_time <= 30

Submitted by: Akash Tadwai , ES18BTECH11019.





1. From the above graphs we can see that initially both the scheduling algorithms are having the same average waiting time because the CPU Utilisation being ***less than 1***. As the number of processes increased we can see a slight increase in waiting time of EDF than RMS. Under this setting ***RMS performs better wrt average waiting time***.

Submitted by: Akash Tadwai , ES18BTECH11019.

2. When CPU utilization doesn't exceed 1 we can see that ***EDF has proven to be a better 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 :

$$N(2^{1/N} - 1)$$

Conclusion :

From the graphs we can conclude that CPU utilization is exceeding 1 then RMS is the best scheduling algorithm in terms of Avg. Waiting time and EDF performs better in terms of Deadline Miss.

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

For bonus points:

1. For adding the context switch overhead. We have to just make minor changes in the code. As the **flag** for every pid will be true only if it is present in the ready_q. For implementing context switching we just add the context switch overhead to the globalTimer variable in printInfo() function. This works because, for every case where the process either starts or preempted and then resumes it comes to the printInfo block and hence we just add the context switch time to globalTimer variable.

2. The whole code for context switching is available in **context_switch/** directory.

Submitted by: Akash Tadwai , ES18BTECH11019.