Simulating RMS & EDF

Topic - Simulating 2 Real-time scheduling algorithms Rate Monotonic Scheduling (RMS) & Earliest Deadline first (EDF) and analysing their performance.

Submitted By: Kushagra Indurkhya
CS19B1017

Contents

Introduction

I am simulating two real time scheduling algorithms RMS and EDF both are real time and priority based scheduling for recurring processes in rms priority is given to process with less period (occurs more often) and in edf priority is given to process which has the earliest deadline.

Implementation design

For each process a struct is created containing the following:

Data

- Meta information about the process provided in the input (process_id,processing time,period,number of repetitions)
- Priority (in case of RMS - 1/period)
- Dynamic details about current repetition of the process(starting time,deadline,cpu_execution left)
- Stats (number of deadlines missed/completed,waiting time)

Methods

- constructor - for creating a process and initialising variables to their initial values.

- complete_iteration()-this function updates the starting time, deadline of the process, cpu execution left, number of processes left & the stats, whenever the process completes its deadline.

```
void complete_iteration()
        {
            this->curr_deadline += this->period;
            this->curr_start += this->period;

            this->completed_deadline++;

            this->cpu_execution = p_time;

            this->iteration_left--;
        }
```

- miss()-this function updates the starting time, deadline of the process, cpu execution left, number of processes left & the stats,it takes t as argument which tells when we get to know that the process will miss its deadline.
```
void miss(int t)
        {
            this->missed_deadline ++;

            //Subtracting the waiting time in the current process
            this->waiting_time -= 
t-curr_start-(p_time-cpu_execution);
            //adding period to waiting time
            this->waiting_time += period;

            this->curr_deadline += this->period;
            this->curr_start += this->period;
            this->cpu_execution = p_time;

            this->iteration_left--;
        }
```

All the processes from input-params file are stored in a vector of processes .

To simulate both the algorithm i am running an infinite for loop which increments t by 1 unit in every iteration , in each iteration:

- First we retrieve the process which can be executed and has highest priority as of now.
    - In **RMS** the vector is already sorted by the priority of each process ie (1/period) Process with highest priority comes first and ties are broken with process id (smaller id is given preference)
    ```
    bool compare(process a, process b)
    {
        if(a.priority != b.priority)
            return a.priority > b.priority;
        else
            return a.process_id < b.process_id;
    }
    ```

    since the priority order will remain the same sorting is required only once
    We retrieve the index having the process which can be executed.

    ```
    int get_min_process_start_time(vector <process> p,int t)
    ```

```
{
    for(int i=0;i<p.size();i++)
            if(p[i].curr_start <=t && p[i].iteration_left > 0
)
                return i;
    return -1;
}
```

We are also keeping track of the index of the last process which was executed.

- In **EDF** the deadlines can change in every iteration hence the vector needs to be sorted (by deadline such that earliest deadline comes first and ties are broken by process_id)
  EDF could have given better performance if ties were broken on a first come first serve basis.

```
bool compare(process a, process b)
{
    if(a.curr_deadline != b.curr_deadline)
        return a.curr_deadline < b.curr_deadline;
    else
        return a.process_id < b.process_id;
}
```

Also , we cannot use the index of the last executed process since the vector would have been changed after sorting so we maintain the process_id of last executed process instead

So we first find the process id of the process which can be executed and has earliest deadline by calling this function

```
int get_min_process_start_time(vector <process> p,int t)
{

    for(int i=0;i<p.size();i++)
            if(p[i].curr_start <=t && p[i].iteration_left > 0
)
                return p[i].process_id;
    return -1;
}
```

Then we update the last_idx by finding the id of the last executed process in the current state of the vector and also find the index of the process returned by the above function for easy access.

```
int find_idx(vector <process> p,int id)
{
    for(int i=0;i<p.size();i++)
            if(p[i].process_id == id)
                return i;
    return -1;
}
```

- Then we check if p_idx is valid (! -1) if it is not then it means that no valid process can be found then the it implies cpu will be in idle state for this iteration, and if we have a valid process then:
    - We check if the cpu was idle before this if it was then we start the new process straightaway else:
        - We check if this process is same as the last process if it is not then:
            - We check if the last process was completed successfully then this means last process ended and now the process is either starting or resuming else:
            - It means that the current process is preempting the last process.

  After checking all these and outputting to the log file , we decrement the cpu_execution by 1 and then we check if the process is finished after this decrementation and if it is then we call *complete_iteration()* on it.

  Update the last process index (process_id in case of **EDF**)

- After the execution of this process we check if any process started in the next iteration will surely fall short of completing its deadline if it will then we call *miss()* on it.

- Also we check using a k_flag if all the processes have completed all their iterations if they have then we break from the infinite for loop.

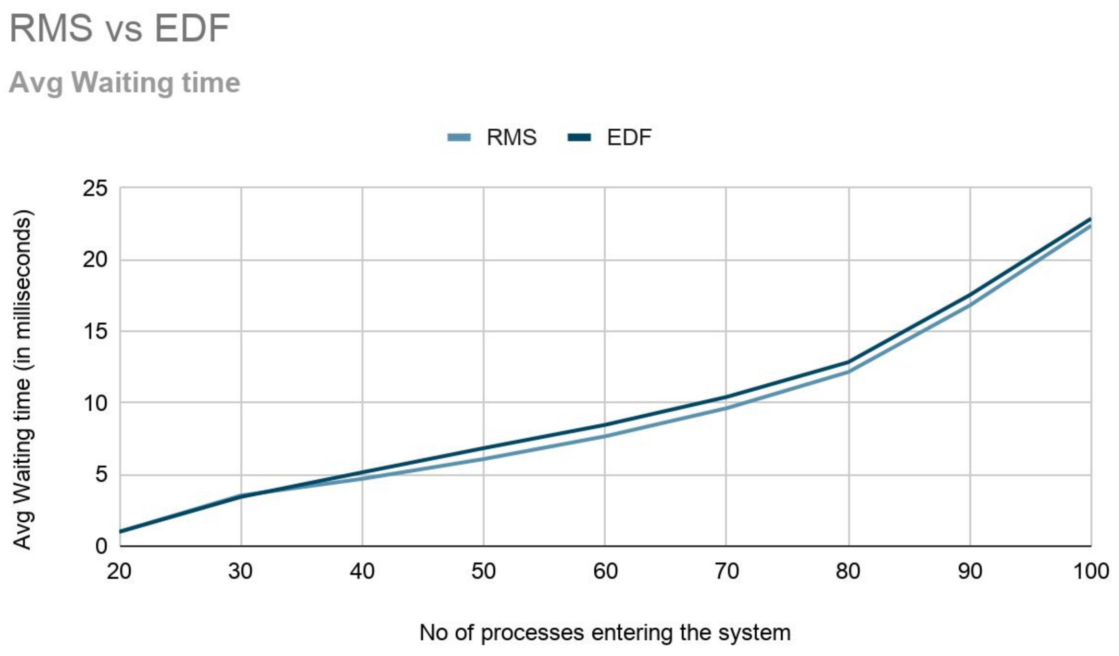
- We if the process was eligible (curr_start < t+1 && iteration_left >0) and did not execute then it was waiting hence we increment its waiting time by 1.

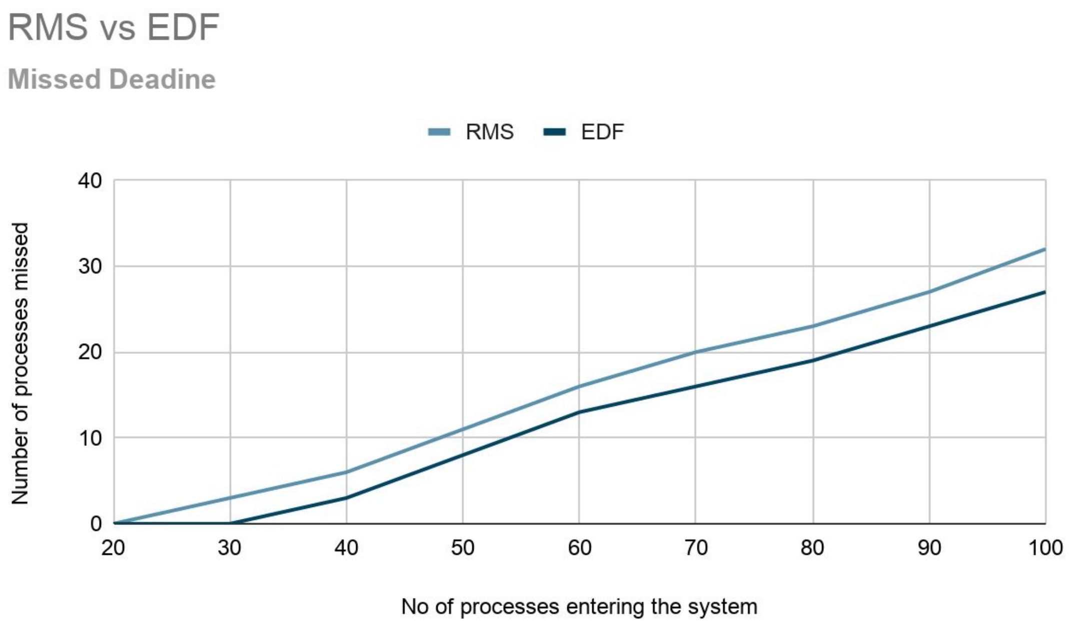
After all the iterations of all the processes are finished we will break out from the for loop and output stats (Total processes,Avg waiting time, completed deadline , missed deadlines) to the stats output file.

(Average waiting time for each process = ∑ (Average waiting time for each instance of that process) / number of instances(k)
Avg waiting time = ∑ (Average waiting for each process) / number of process(n))

Analysis of performance

To analyse the performance of both the algorithms i have plotted two graphs of number of processes against avg waiting time and number of missed deadlines





As we increase the number of processes the waiting time of edf is becoming more than that of rms but the number of missed deadlines is less in edf,Hence **EDF provides better cpu-utilization while rms reduces the waiting time for the process**.

Including the time taken in selection and context-switching

To give more accuracy to the simulation model I have tried to take in consideration the time taken in selection of the process and context switching.
For selection time, i am adding the total time taken in each iteration of the for loop between the start of iteration and just before the execution of the process, to t.
Context switching time is added when :

- a process starts or resumes after the last process is ended ,between the ending and start/resume.
- A higher priority process preempts the running process . between the preemption and start of the higher priority process

The code for the same is commented between “ //************* “

Defining context switching time as 10 microseconds

```
//***************************
// #define CONTEXT_SWITCH 0.010
//***************************
```

These variables are used for keeping track of time used in selection of the process

```
    //***************************
    // clock_t start_time,end_time;
    //****************************
```

Clock is started each time as soon as we enter in for loop

```
            //***************************
                // start_time=clock();
            //****************************
```

Context switching occurs when a process is preempted by another or a process ends and another process starts or resumes, context switching time is added to t

```
//***************************
    // t+= CONTEXT_SWITCH;
//****************************
```

Just before executing the process we add the time taken till now(converting from micro to milli seconds) in selecting the process to t.

```
//***************************
            // end_time = clock();
            // t+= (end_time-start_time)*.001;
            //***************************
            processes[p_idx].cpu_execution--; //Executing the process
```