[INFO-F404] Real-Time Operating Systems
EDF vs LLF - Project report

BUI QUANG PHUONG Linh - ULB ID : 000427796
PAQUET Michael - ULB ID : 000410753
SINGH Sundeep - ULB ID : 000428022
MA1 Computer Sciences

December 2018

Introduction

In this project, we study the difference between two scheduling algorithm on uniprocessors : EDF and LLF. On one side, the practical tools have to been set up such that creating a system generator, parsing information contained in the system file (i.e. periods, WCET and offsets). In the other hand, the main part consisting by implement the EDF and LLF algorithms has to be done based on the system file.

1 Pseudo-code and implementation choices

1.1 Code structure

The code is organized in four parts regrouping several functions:

- Parsing functions
- *Question 1* - EDF feasibility interval computation
- *Question 2* - System generator
- *Question 3* - EDF and LLF implementation

1.1.1 Parsing functions

The parsing part contains all the functions required to parse the different values (offset, WCET and period) of the system's tasks. This will be useful to manipulate easily those values later.

- `readFile(filename)` : reads a file and returns a list of the tasks and its corresponding offset, WCET and periods.
- `getOffsetWCETPeriodLists(systemList)` : takes the list of tasks returned by the `readFile` function and returns the offset list, the WCET list and the periods list regrouping all the offset/WCET/periods values of all tasks.

1.1.2 EDF feasibility interval computation

In this part, a function is calculating the EDF feasibility interval while the other one is printing it in the good format as asked in the first question of the statement. Moreover, to compute the feasibility interval, we need to calculate the least common multiple which is done in the `LCM` function.

- `LCM(numbers)` : calculates the least common multiple of a list of numbers
- `computeFeasibilityInterval(newSystemList)` : computes the feasibility interval following the formula : $[0, max\{D_i | i = 1, \ldots, n\}]$ such that $max(D_i)$ is the greatest deadline value with i the number of tasks.
- `printFeasibilityInterval(feasibilityIntervalUpperBound)` : print the feasibility interval following the format : $[0, O_{max} + 2 \cdot P]$

1.1.3 System generator

This section is dedicated to the generation of systems. To generate a system that satisfies the condition of the utilization percentage, we have first to verify this condition which is done in the `matchRequiredUtilisationPercentage` function. Thus, we can easily generate the system file. Note that we arbitrary decided that the limit of the period/WCET value is 50 while that of the offset is 2. Moreover, for the utilisation percentage condition, we have introduced an error margin value which is represented by the parameter `delta` in the following functions. In summary, we have :

- `matchRequiredUtilisationPercentage(wcets, periods, percentage, delta)` : checks whether or not we have our required utilization percentage with the values generated randomly
- `generateTasks(numberOfTasks, requiredUtilisationProcent, delta)` : generates the tasks by returning the offsets, wcet and periods values randomly generated
- `printFeasibilityInterval(feasibilityIntervalUpperBound)` : writes the generated tasks in the `tasks.txt` file

1.1.4 EDF and LLF implementation

Before implementing EDF and LLF algorithms, the first thing to do is to get the different task's deadlines. To do that, we simply need to get the multiples of the period's value until the limit of the scheduling taking into account the potential offset. We then store those deadline values in a dictionary doing the link between the deadline and their corresponding tasks illustrated as followed, with i the task number :

$$\{task_i : deadlines_i\}$$

Furthermore, the most important point of those scheduling algorithms is to know which task is executed first. In the case of EDF, which means *Earliest Deadline First*, we pick the lowest deadline at time t as indicated by his name. Thus, we need a function which is doing this job, that's why `getSmallestDeadlines` is present. In the case of LLF, which means *Least Laxity First*, a new notion appears : the laxity. It's this new notion of laxity that will determinate the task to execute at time t. The laxity of a job j is computable by the formula

$$l_j(t) = d - t - (e - \epsilon_j(t))$$

where d is the task's deadline, t the current time, e the execution time (WCET) and $\epsilon_j(t)$ the cumulative CPU time used by J.

To summarize, those functions are added to those previously presented to finally implement the EDF and LLF algorithms, first, to get the deadlines :

- `getMultiplesOf(number, limit, offset)` : computes the multiples of a number considering the offset until a certain limit. Here this number is the first deadline of a task and the multiples are the next deadlines.
- `getTasksDeadlines(systemList, upperBound, offsets)` : returns a dictionary linking the tasks of the system and all their respective deadlines

Now that we have all the deadlines, here are the two functions which compute the smallest deadline in case of EDF and the laxities in the case of LLF:

- `getSmallestDeadlines(tasksDeadlinesDict, isJobDoneUntilNextDeadline)` : get the task with the smallest deadline value in the previous computed dictionary in `getTasksDeadlines( systemList, upperBound, offsets)`.
- `computeLaxities(time, tasksDeadlines, e, isJobDoneUntilNextDeadline, CPUTimeUsed)` : computes the laxity of all jobs to later get the smallest value to execute in the laxity list returned by this function.

Moreover, some utility functions are created to make the code more clear such that initialization functions or conditions checking functions:

- `initJobsList(systemList)` : initialiaze a dictionary with i zero, where i is the number of tasks. Those values are the current job number of the tasks, if the current job executed of the first task, then the value of the first key of the dictionary will be 1.
- `initIsJobDoneDict(systemList)` : initialize a dictionary of i boolean values indicating if the current job is done, where i is the number of tasks
- `isSchedulable(systemList, end)` : check whether or not a system is schedulable depending of the feasibility interval
- `isDeadlineMissed(deadline, t)` : check whether or not a deadline is missed, happens when a job is not fully executed before his deadline

All the tools are now available to implement the EDF and LLF algorithms, **the pseudo-code of the two following functions are detailed in the next section**.

- `EDF(system, begin, end)` : the "earliest deadline first" scheduling algorithm
- `LLF(system, begin, end)` : the "least laxity first" scheduling algorithm

Finally, a clear command line or graphical output are available. This printing part is done in those two following functions :

- `printOutputs(tasksExecuted, arrivalJobOutput, begin, end, systemList, preemptionsNb)` : prints the scheduling in the command line
- `printGraph(tasksExecuted)` : displays the graphical output of the scheduling

1.2 Pseudo-code EDF and LLF

Our implementation of the EDF scheduling algorithm is presented in the pseudo-code presented in ?? while the pseudo-code for LLF is presented in ??. The difference of LFF compared to EDF are brought out in red.

Algorithm 1 EDF scheduling

```
1:  procedure EDF(system, begin, end)
2:      Parsing of the system file
3:      Initialization of tasksDeadlinesDict : the tasks deadlines dictionary, and jobs : tasks jobs list
4:      Initialization of empty list tasksExecuted which will contain the tasks executed over time t
5:
6:      if schedulable then                          ▷ end ≤ upper bound of feasibility interval
7:          while the scheduling isn't finished do
8:              for each deadline i do
9:                  if i = currentTime t then
10:                     job arrival
11:
12:             Assign the current executing task j by taking the task whose the current job
13:             has the smallest deadline
14:
15:             if deadline missed then                          ▷ deadline of job_j < t
16:                 break the loop and add "miss" to tasksExecuted
17:             else
18:                 Append j to the list of task executed
19:                 Decrements WCET of j by 1
20:                 Compute preemption number
21:                 if current job of j done then                          ▷ WCET_j = 0
22:                     Increment jobs_j by 1
23:                     Remove the deadline of job_j from tasksDeadlinesDict
```

Algorithm 2 LLF scheduling

```
1: procedure LLF(system, begin, end)
2:     Parsing of the system file
3:     Initialization of tasksDeadlinesDict : the tasks deadlines dictionary, and jobs : tasks jobs list
4:     Initialization of empty list tasksExecuted which will contain the tasks executed over time t
5:     Initialization of empty list laxityOfJobs which will contain the computed laxities for all jobs
6:
7:     if schedulable then                                  ▷ end ≤ upper bound of feasibility interval
8:         while the scheduling isn't finished do
9:             for each deadline i do
10:                if i = currentTime t then
11:                    job arrival
12:            Compute the laxity of jobs and put it in laxityOfJobs
13:
14:            Assign the current executing task j by taking the task whose the current job
15:            has the smallest laxity
16:
17:            if deadline missed then                      ▷ min(laxityOfJobs) < 0
18:                break the loop and add "miss" to tasksExecuted
19:            else
20:                Append the current executing task j to list of task executed
21:                Decrements WCET of j by 1
22:                Compute preemption number
23:                if current job of j done then            ▷ WCET_j = 0
24:                    Increment jobs_j by 1
25:                    Remove the deadline of job_j from tasksDeadlinesDict
```

2 Difficulties encountered

No real difficulties were encountered. The main difficulties are generally related to the Graphical User Interface which was then problems with the corresponding library `matplotlib`. To resolve this, we simply do some researches associated the problems encountered.

3 Graphical part library

To plot the outcome of the scheduling graphically, we used the library `matplotlib` of `Python`. No download is required. The only requirement is to installing the library in your computer using in the command line :

- in Windows : `pip install matplotlib`
- in Linux/MacOS : `sudo apt-get build-dep python-matplotlib`

4 EDF vs LLF : preemption comparison

The difference observed between the number of preemption for an EDF scheduling and an LLF scheduling is that LLF has quite more preemptions than EDF. This difference is explained by the dynamic of priority change for each case. Indeed, the priorities of EDF are dynamic at task level and fixed at job level while LLF is a job-level dynamic priority scheduler. Therefore, there are more priorities change in case of LLF than EDF, then more risks of preemption when scheduling LLF.