Real-Time Operating Systems

INFO-F-404

Scheduling project : Partitioned EDF

19 November 2020

HAUWAERT Maxime : 000461714

Université Libre de Bruxelles (ULB)

Contents

1 Introduction

In this project it was asked to develop a fixed job priority scheduler in a partitioned identical parallel systems simulator. The program is written in Python3 and takes all the tasks as well as all the options, partitions them, then shows the simulated execution.

2 Concepts

In order to fully understand this report a few concepts have to be defined first :

- **Fixed job priority scheduler** : It is a scheduling algorithm that assign a fixed and unique priority to each job during system execution.
- **Identical parallel system** : It is a system in which all the processors have the same computing power.
- **Partitioned scheduling** : It is a method of partitioning several tasks on multiple processors. Once a task is assigned to a specific processor, it cannot migrate. For each processor a local uniprocessor scheduler is used.
- **First fit** : It assigns the current task to the first processor that can take it in order of their indexes.
- **Worst fit** : It is that same as the first fit algorithm but the processors are sorted by their utilisation factor with the one having the least being the first (ascending).
- **Best fit** : It is that same as the first fit algorithm but the processors are sorted by their utilisation factor with the one having the highest being the first (descending).
- **Next fit** : It tries to assign the current task to the last processor used. When it failed the last processor is closed and the task is added to the next processor.
- **EDF** : It is a fixed job priority scheduler where the priorities of the jobs are relative to their deadlines. It always runs the job which has the earliest deadline.

3 Description

The program takes the tasks contained in the specified file then partitions them using the sorting and the heuristic method specified then print the executions of the schedulers for the specified amount of time.

As this project is based only on single core processors, the terms core and processor can be confused with each other.
This project is about unrestricted periodic task so from now on those tasks will just be referred to as tasks.

EDF can schedule everything that is schedulable. In other words if the set of tasks is schedulable on the same processor then there is no job with EDF that will miss a deadline.
But EDF is not really common in real-time operating systems because of different reasons such as when the system is overloaded it is impossible to predict when the deadlines will be missed and it is difficult to implement it in hardware.

4 Implementation

The code can be launched either by executing this command : "./partitioned_edf" followed by the options from the root directory.
All the sources file are placed in the `source` folder.

There are five main steps:

- The presence of the mandatory options and the validity of all the options are verified.
- The task set file is analysed by the parser that return a list of tasks.
- The partitioner partitions the tasks on the different processors and returns the result.
- The scheduler simulates the executions on the different processors and return the timelines of each execution.
- The taskset, the options, the partitions and the timelines of the executions.

4.1 Task

A periodic task is defined by its offset, its computation time, its deadline and its period. This class only adds an argument, its ID (which can be omitted). There is no restriction on the tasks as this project focus on the periodic asynchronous arbitrary deadline systems.

4.2 Job

A job is defined by a release time, a computation requirement and an absolute deadline. This class only adds two arguments : the ID of the task linked to it and its ID. It also adds a variable that represents the number of time unit the job has run. It is useful to know if a job is finished and to detect if its deadline is not met.

4.3 Partitioner

This class represents a partitioner that use the partitioned scheduling method.
It works in two main steps :

- It sorts the specified tasks by their utilisation factor, ascending or decreasing depending on the `sort` option.
- It runs the specified heuristic method to fill the different processors.

The heuristic method, the sorting method and the number of cores are defined in the constructor. And in the `partition` method the tasks are passed.

4.4 EDFScheduler

This class represents an EDF scheduler.
This class has three main functions:

- It can schedule the specified tasks of a processor
- It can schedule the specified partitioned tasks (on multiple cores) by scheduling them one by one.

- It can determine the feasibility to schedule the specified tasks together.

As they are no constraint on the tasks (periodic asynchronous arbitrary deadline), two points have to be checked in order to check the feasibility of the tasks specified with the EDF scheduler:

- Every deadline occurring in $[0, O_{max} + 2P]$ is met.
- The configuration at $O_{max} + P$ and at $O_{max} + 2P$ are equal.

Only and only if those two points are true that the tasks can be scheduled together.

P is the hyper-period is defined as the least common multiple of all the periods of the tasks.
O_{max} is the maximum offset of the tasks.
The configuration is defined as follow:

$$C_S(R,t) = ((\gamma_1(t), \alpha_1(t), \beta_1(t)), ..., (\gamma_n(t), \alpha_n(t), \beta_n(t))$$

`where`
R is the system of the tasks
t is the time
$\gamma_i(t)$ is the time elapsed since the last request of τ_i
$\alpha_i(t)$ is the number of active jobs of τ_i
$\beta_i(t)$ is the cumulative CPU time used by the oldest active job of τ_i

The time limit and the partitioned tasks are passed in the `scheduleAll` method.

4.5 Event

This class represents an event that happened on a processor.
The events can be instantaneous or continuous (for one unit of time). An event is defined with a type and one or no job.
The possible types are the following :

- Run : when the specified job runs a unit of time t to $t + 1$
- Release : when the specified job is released at t
- Deadline : when the specified job's deadline is reached at t
- Idle : when no job is running at t to $t + 1$

The last event is the only event that doesn't need a job.

4.6 Timeline

This class represents a timeline that contains several events.
It associates a time with none, one or several events. The scheduler have to fill it with different events. The only particularity is that the scheduler do not have to add an idle event as it is the default one for the utilisation of the CPU.
This class has three internal lists, one for the CPU state, one for the release events and one for the deadlines events.

4.7 Notes

- When printing the executions of the schedulers it has been decided to print the timelines one by one. Because printing all of them together, with each event having a processor id, would be very unclear with large numbers of tasks.
- In order to calculate the least common multiple the function *lcm* of the math library of Python has been used, so that the users do not have to install any external library. The only downside is that the function was added in the version 3.9 of Python so the users need to have Python version 3.9 or higher installed.
- Because of the mandatory use of the least common multiple, the program can take a lot of time to partition the tasks with a large number of tasks.
- To execute the via the `./partitioned_edf` command line make sure to give the file the execution permission.

5 Test

In order to test the program three `taskset` files have been added and are located in the `test` folder :

- tasksetBase.txt : It is the taskset of the statement of this project.
- tasksetNonSchedulable.txt : It contains two tasks that cannot be scheduled on one processor despite having the sum of their utilisation factors equals to 1.
- tasksetSwitch.txt : During the execution, the first job of the first task has to be interrupted to execute the first job of the second task.

6 Conclusion

This program is a great tool to try the different options (the sorting option and the four heuristic methods) to partition the tasks among the processors and to schedule the partitioned tasks with the EDF scheduler for the specified amount of time.

7 Problems encountered

The only problem encountered has been the structure of the class EDFScheduler. The other classes have not received any major changes during the implementation of the project. On the other hand EDFScheduler have received many major changes as many possibilities have been explored and the best has been chosen. The declaration of the public methods didn't change but the internal variables of the class did.
For example a solution with two lists, one for the active jobs and one for the finished jobs. It was used to fill the timeline of the deadline events. But as the deadline is already known at the release of the job, in the current version when a job is released a release event and a deadline event are added with their corresponding time.

8 User Manual

./partitioned_edf <tasks_file> <-h option> <-s option> <-l number> [-m number]		
tasks_file	Specify the path of the file containing the tasks.	The path can be absolute or relative to the current working directory.
-h option	Specify the heuristic method to use for partitioning the tasks on the cores.	Valid values are : • ff : for the first fit method • wf : for the worst fit method • bf : for the best fit method • nf : for the next fit method
-s option	Specify the order of the tasks by their utilisation factor before running the specified heuristic method.	Valid values are : • iu : for the ascending order • du : for the descending order
-l number	Specify the time limit of the simulated execution.	Valid values are all the natural numbers above 0.
-m number	Specify the number of cores.	Valid values are all the natural numbers above 0. By default 1.
< > = mandatory		[ ] = optional
Warning Python version 3.9 or higher required		

9 Bonus

9.1 Tasks Generator

9.1.1 Description

A tasks generator has been implemented to have different sets of tasks to test the main program. This generator generates random synchronous systems of tasks with constrained deadlines.
A synchronous system with constrained deadlines, is a set of tasks where for each task, its offset is always 0 and its deadline is always between the WCET and the period.
This system has been chosen as it is one of the most common system.

9.1.2 Implementation

The generator takes the following options : `tasks_number`, `period_range`, `utilisation_factor_range` and the `output_file`.(These options are explained in the user manual)
These are the main steps of the generator for generating a random task :

- It sets the offset to 0
- It chooses a random period in the `period_range`
- It uses the period and the `utilisation_factor_range` to calculate the `WCET_range`
- It chooses a random WCET in the `WCET_range`

- It chooses a random deadline in the range [WCET,Period]

The `WCET_range` is defined as

[lower limit of `utilisation_factor_range` $*$ *period*, upper limit of `utilisation_factor_range` $*$ *period*]

The generator can be launched by executing this command : "./tasks_generator" followed by the options from the root directory.

9.1.3 User Manual

./tasks_generator <-n number> <-p range> <-u range> <-o file>		
-n number	Specify the number of tasks to generate.	Valid values are all the natural numbers above 0.
-p range	Specify the range of the period of the tasks.	The lower limit and the upper limit are separated only by a comma. The valid values of the limits are all the natural numbers above 0.
-u range	Specify the range of the utilisation factors of the tasks.	The lower limit and the upper limit are separated only by a comma. The valid values of the limits are all the real numbers from 0 (not included) to 1 (included).
-o file	Specify the file to write the generated tasks in.	The path can be absolute or relative to the current working directory. If the file already exists it will be overwritten.
All the options are mandatory		
The upper limits should always be greater than their corresponding lower limits		
Example : ./tasks_generator -n 10 -p 10,100 -u 0.2,0.6 -o taskset.txt		