Université Libre de Bruxelles

Scheduling project

INFO-F404

Baudru Julien

November 19, 2020

Contents

1 Introduction

The goal of this project is to simulate a scheduling of a set of tasks on a system with several cores. This project consists of two main parts, the first one being the partitioning of the tasks between the cores and the second part consists in scheduling the tasks on each of the cores.
We assume, for the sake of simplicity, that the different cores have the same characteristics, i.e. that the system is homogeneous. We also assume that tasks cannot switch from one processor to another while the system is running, i.e. there is no global scheduling. In addition, we also presume that the values of offset, wcet, period and deadline are multiples of 10, as in the examples given in the course.

2 User manual

2.1 The scheduler

The language used in this project is **Python 3.8.5**, you can download this version of Python via the following link. To run the program, you must first install the following python libraries :

```
pip install binpacking
pip install turtle
```

The library called *binpacking* is used by the class **Partitioner** to distribute the stains on the different cores, for more information about this library you can visit this link.

The library named *Turtle* is used for the graphical display of the program results. This module was chosen as a display module because of its ease of use. It is important to note that it is not the most optimal module for this kind of task, as it is normally 'drawing oriented' and it takes some time to draw the different elements of the graphical interface. The more tasks and cores there are, the longer it will take to display, however this time is never too long and this is not the most important aspect of this project, moreover it is rather pleasant to see the result appear gradually.

To run the program, enter the following command :

```
python partitioned_edf.py <taskset_file.txt> -h ff|wf|bf|nf -s iu|du -l <limit> -m <# cores>
```

Note that if no options are entered at startup, the default options are the following :

```
-h = "nf"
-s = "du"
-l = 150
-m = 1
```

2.2 The task generator

If you want to get more test files, feel free to run the task set generator, for this you have to enter the following command:

```
python task_generator.py <# task in the set> <total utilization>
```

Here is an example file generated for a set of 5 tasks and a total utilization of 1.2 :

```
0 60 150 150
0 30 150 150
0 30 170 170
0 10 120 120
0 80 240 240
```

An other example file generated for a set of 4 tasks and a total utilization of 1.55 :

```
0 20 120 120
0 70 90 90
0 10 120 120
0 70 130 130
```

As we can see in the example given above, the task generator presented here only allows you to generate task sets with periods, deadlines and wcet are divisible by 10, this choice was made for the reasons stated below :

1. For the sake of simplicity.
2. As most of the examples in the reference course also have periods divisible by 10.
3. Because this constraint does not prevent to obtain examples of complex scheduling.

In addition, the periods of the different generated tasks will always have a value of 0 for their offset, this choice was made for similar reasons as those mentioned above.
The generated files are text files with a name as follows :

```
taskset<year>-<month>-<day>-<hour>-<minute>-<second>.txt
```

These files are stored in the folder *taskset_sample*.

3 Structure and execution

3.1 Overall project structure

The different files composing this project follow the structure below :

```
|---doc
|     |---INFO_F404-Report.pdf
|
|---taskset_sample
|     |
|     |---taskset_example.txt
|     |---tasket_example2.txt
|     |...
|
|---utils
|     |
|     |---gui.py
|     |---error.py
|     |---partitioner.py
|     |---scheduler.py
|     |---task_parser.py
|     |---task.py
|
|---partitioned_edf.py
|---task_generator.py
```

The code is divided into two main parts, on one side at the root of the project, there is the two main files to be launched to start the programs (c.f. *partitioned_edf* and *task_generator*) and on the other side, in the *utils* folder, the classes used by these two programs. Each class in this folder has a unique and precise role to ensure that the whole follows the VMC model as closely as possible, in the sense that the view is separated from the rest. Note that classes *Task* and *Error* are used by *partitioned_edf* and by *task_generator*, all other classes are only used by *partitioned_edf*.

The different classes used in the *partitioned_edf* program are shown in the simplified class diagram below.

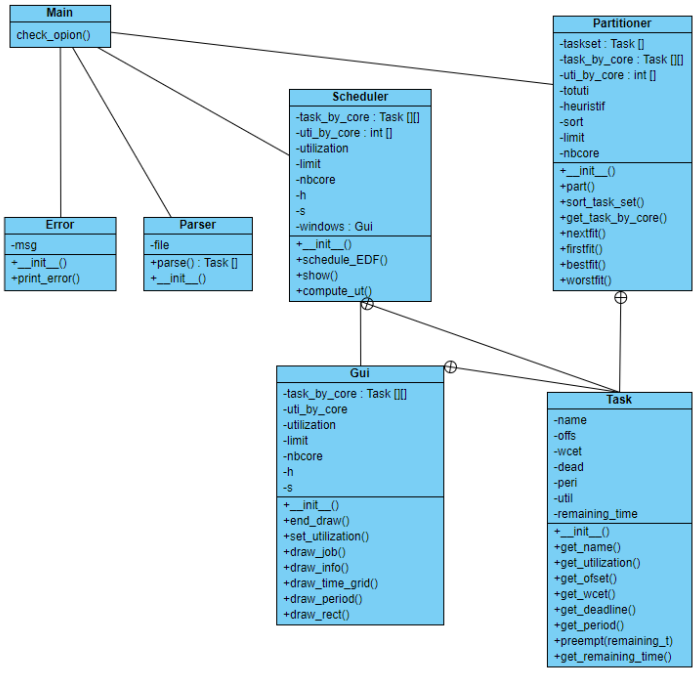

Figure 1: Simplified class diagram

Note : All the code that has been written in this project try to be as compliant as possible with the *snake_case* naming style.

3.2 Steps during the execution

The numbers of the steps below correspond to the crucial points in the execution of the *partitioned_edf* program, you will find them in the pictorial representation of the execution below.

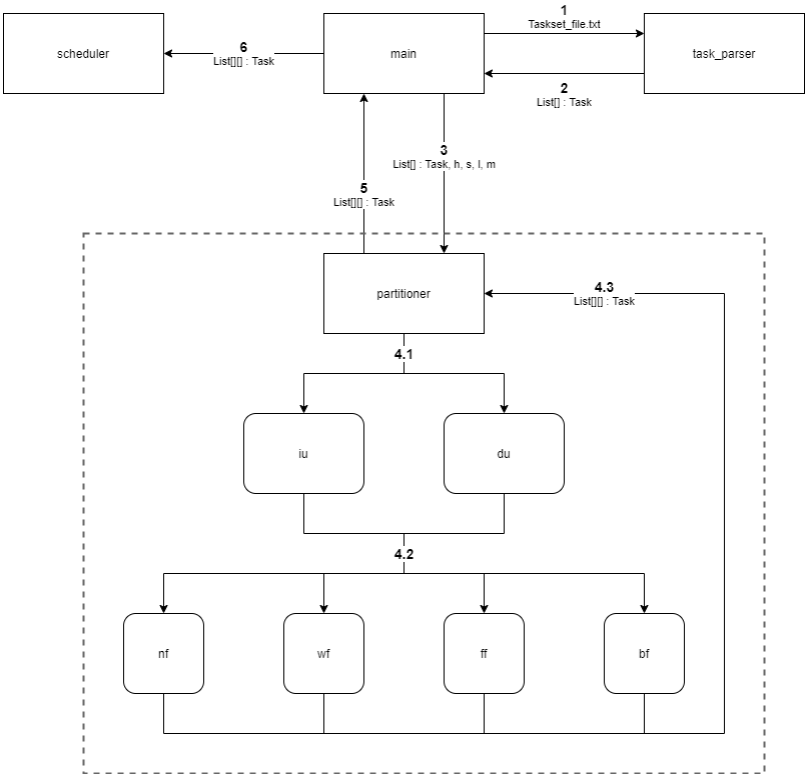

Figure 2: Execution diagram

1 : The raw file containing the set of tasks is sent to the class **parser**.
2 : The parser will process this file and return a list of object of type **task**.
3 : This list is send to the class **partitioner** as well as the options chosen by the user : *-h* for heuristic, *-s* for sort, *-l* for limit and *-m* for the number of core.
4.1: Then the partitioner will call the function **sort_task_set** and depending on the value of the *-s* option, it will sort the task list either in ascending order of utilization (if the value of the option equals *"iu"*), or in descending order of utilization (if the value of the option equals *"du"*).
4.2 : After, depending on the value of the *-h* option and the value of the *-m* option, the partitioner will distribute the tasks on the different cores, either following the "Next Fit" method (if the value of the option equals *"nf"*), or the "Best Fit" method (if the value of the option equals *"bf"*), or the "Worst Fit" method (if the value of the option equals *"wf"*), or the "First Fit" method (if the value of the option equals *"ff"*).
4.3 : The final result is a matrix where each row i is the set of tasks for core i.
5 : This matrix is sent to the main function.
6 : Finally, thanks to the scheduler class, each set of tasks is scheduled for its core with the EDF (Earliest deadline first) strategy.

If no scheduling is possible for the number of cores requested by the user, the program stops and displays an error message via the "Error" class inviting the user to try again with a higher "-m" parameter.

Concerning the scheduling via the EDF strategy, it seems important to specify that the assumption is made that the priorities assigned to the tasks are fixed. Moreover, the deadlines are considered here implicitly.

Note : As said before, the *binpacking* module is used in the **Partitioner** class. This module is called when the user enters as parameter *-h bf*, in fact the "Best Fit" method is in fact a problem similar to the well known bin packing problem. This module provides the best solution to this problem, for this reason it is necessary to provide the maximum capacity of each container and the weight of the different objects that we want to store. In our case, the containers are the cores, their maximum capacity is the maximum utilization of cores (c.f. 1) and the weight of the objects to be stored are the utilization of each task.

4 Problems encountered

The main difficulty encountered in the elaboration of this program was to understand the differences between *Next Fit*, *First Fit*, *Best Fit* and *Worst Fit* and to find concrete examples of these different algorithms in order to verify our implementation. To overcome this difficulty, we found a simple and concise document explaining the differences between these four methods [1].

The second most important difficulty we encountered was, as expected, the implementation of the EDF algorithm. Especially in cases where the tasks of the same core had different periods (and thus different implicit deadlines). We solved the problems related to this point by the trial-error method.

5 Screenshot

In this section you will find few example of the scheduling obtained on different taskset which have been partitioned partitioned with different parameters.

5.1 Example 1

For this example, the task set used is the following:

```
0 80 100 100
0 40 100 100
0 60 100 100
0 50 80 80
```

With as heuristic parameter "first fit" (*-h ff*), as sorting method "decresing utilization" (*-s du*) and 4 cores (*-m 4*), the result produced by the program given below.

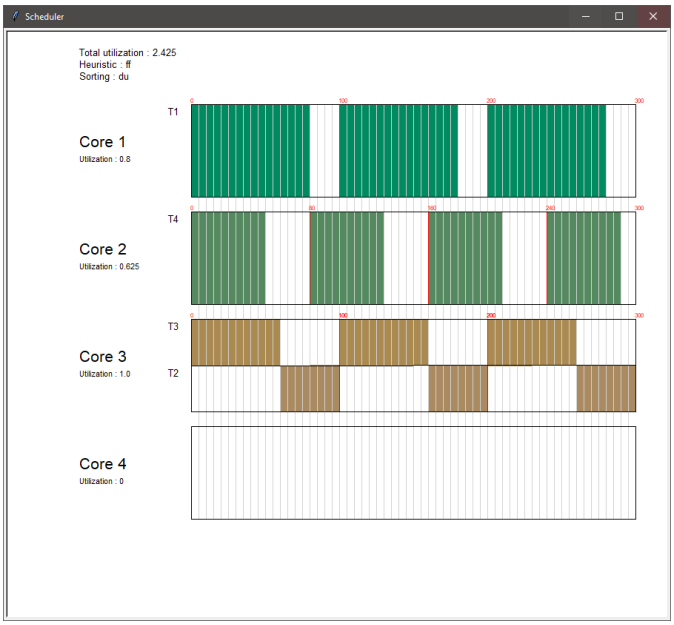

Figure 3: Example 1

We notice that in this configuration the 4th core is not used.

5.2 Example 2

For this example, the task set used is the following:

```
0 80 100 100
0 70 100 100
0 60 100 100
0 50 100 100
0 40 100 100
0 20 100 100
0 10 100 100
```

With as heuristic parameter "next fit" (*-h nf*), as sorting method "decresing utilization" (*-s du*) and 5 cores (*-m 5*), the result produced by the program given this task set is given below.

Figure 4: Example 2

For the same task set, the result for the parameters *-h nf -s iu -m 5* is the following :

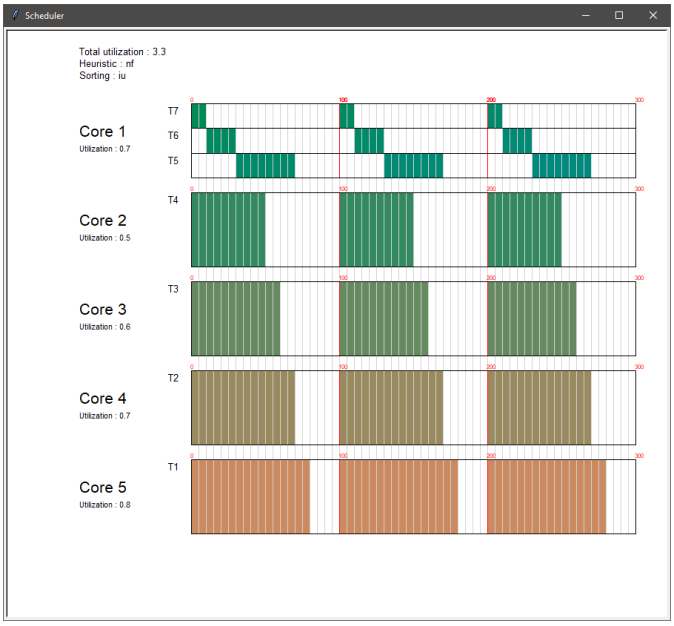

Figure 5: Example 3

For the same task set again, the result for the parameters *-h wf -s du -m 5* is the following :

Figure 6: Example 4

5.3 Example 3

For this example, the task set used is the following:

```
0 10 100 100
0 10 100 100
0 20 100 100
0 45 100 100
0 30 100 100
0 40 100 100
0 50 100 100
0 70 100 100
0 20 100 100
```

With as heuristic parameter "best fit" (*-h bf*), as sorting method "increasing utilization" (*-s iu*) and 3 cores (*-m 3*), the result produced by the program given this task set is given below.

Figure 7: Example 5

5.4 Example 4

For this example, the task set used is the following:

```
0 10 50 50
0 20 50 50
0 60 100 100
0 50 200 200
0 180 200 200
```

With as heuristic parameter "best fit" (*-h bf*), as sorting method "increasing utilization" (*-s iu*) and 3 cores (*-m 3*), the result produced by the program given this task set is given below.

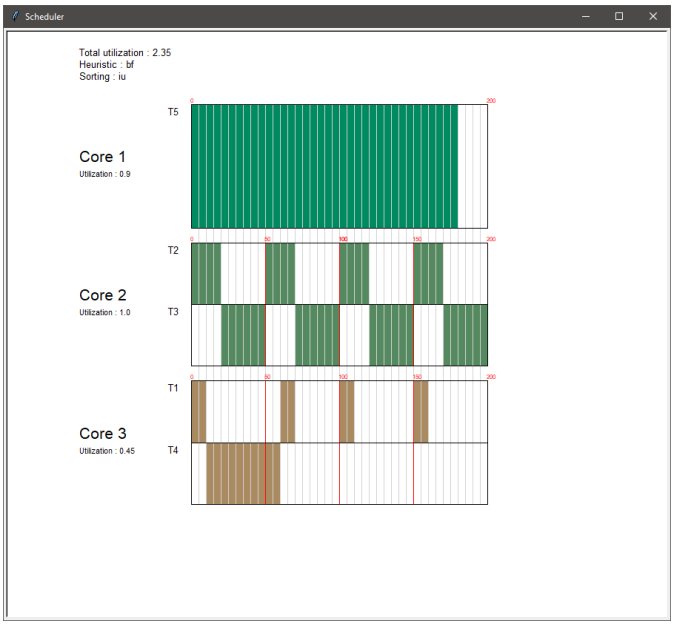

Figure 8: Example 6

In this example, we notice that the two tasks on core N°3 have different periods and that they meet their deadline as expected. Indeed, the task **T1** has a period of 50 and the task **T4** has a period of 200.

6 Future improvements

For information purposes only, among the possible improvements for this program, we have retained the following ideas:

1. A new module for the GUI.
2. An interactive graphical interface, not just the display of the result.
3. An implementation of the *Rate Monotonic* scheduler.
4. A *global EDF* implementation.
5. Adding a *Core* class to the project for more clarity.

7 References

[1] Shannon Jessie. Summary of bin-packing algorithms. http://www.math.unl.edu/~s-sjessie1/203Handouts/Bin%20Packing.pdf, February 2006.

[2] Cecilia Ekelin. Clairvoyant non-preemptive edf scheduling. https://www.researchgate.net/publication/4247643_Clairvoyant_Non-Preemptive_EDF_Scheduling, 2006.