Scheduling project

Group member:
Vincent Gailly
Maxime Renversez

Academic year 2021-2022
Master in computer sciences

Faculty of sciences, ULB

Contents

1 Introduction

This project is divided into three parts. We have to implement an FTP scheduler, a graphical tool to visualize the results of the scheduling and the Audsley algorithm. In this report, we will explain how we did that. In order to do that, we are going to discuss about the structure of our project, the implementation of the FTP scheduler and Audsley algorithm. We will also provide you an user guide. Before concluding, we will explain the difficulties that we met during the project.

2 Project structure (Vincent)

We have decided to do the project in object oriented programming with Python. We have created different classes wich each represent an object and a file which runs the project. We have created these objects :

- a job
- a task
- a scheduler
- a object to visualize the scheduling
- ...

A job is caracterized by a release date, a computationnal time and a deadline.

A completer quand audsely sera fini + faire diagramme uml !!

3 User guide

In the repository of the project, you use the command :

- python main.py option1 option2

Where the value of option1 is “scheduler” if you want to run the scheduler and display the result. If you want to apply Audsley algorithm, the value of option1 must be “audsley”. The parameter option2 is a file which contains all the tasks. For example if you run the command : “python main.py scheduler Tasks.txt”. You obtain the figure 5 as result (see section 4.3).

4 FTP scheduler

4.1 Algorithm

```
1  time = 0
2  job_duration = 0
3  job_start = 0
4  task_number = tasks_list[0].task_number
5  feas_int = computeFeasibilityInterval()
6  while time <= feas_int and verifyDeadlines(time):
7    task_to_execute = canBegin(time)
8    if task_to_execute != -1:
9      if task_to_execute != task_number:
10         if task_number != -1:
11           tasks_list[getTaskIndex(task_number)].schedule_solution.append(
12           job_duration = 1
13           job_start = time
14         else:
15           job_start = time
16           job_duration = 1
17       elif task_number == task_to_execute:
18         job_duration += 1
19       executeTask(task_to_execute)
20     elif task_number == -1 and task_to_execute == -1:
21       pass
22     else:
23       tasks_list[getTaskIndex(task_number)].schedule_solution.append((job
24       job_duration = 0
25       job_start = 0
26     task_number = task_to_execute
27     time += 1
28   if task_number != -1 and time < feas_int:
29     tasks_list[getTaskIndex(task_number)].schedule_solution.append((job_s
30   return not time < feas_int
```

The purpose of this algorithm is to execute the FTP scheduler and returns "TRUE" if it ends correctly and "FALSE" otherwise. If it returns "FALSE" it means that the set of tasks are not schedulable. The variable **time** represents the current time in the scheduler. The variables **job_duration** and **job_start** represent the duration of the execution of one job and at which time the job is executed for the first time (these variables allow us to keep a track of the execution of all the jobs of all the tasks). The variable **task_number** is the number of the task which has its job that is being executed (if the task which executes its job first is not the task 1 it is not a problem because in the list of solution for the task 1 we add the tuple "(0,0)") . The variable **feas_int** represents the upper bound of the feasibility interval which is calculated by the method "computeFeasibilityInterval()". The while loop will be explained in subsection 4.2. The condition at the line 29 allows to add the execution of the job of the task which was executed when a deadline was missed

in the list of solutions of the task.

4.2 Explanations of the while loop

To stay in the loop, two conditions must be respected :

- The time must be less or equal to the upper bound of the feasibility interval.
- No deadlines are missed.

When the program enters in the loop, it first calculates which task can execute its job and stores its number in the variable **task_to_execute**. It is done by the method "canBegin(time)" which takes the current time in parameter and returns a task number. This method respects the task priorities. For example, if two tasks can execute their job at the same moment, it returns the task number which has the higher priority. But if no tasks can execute a job (it is when all the jobs have finsihed their executions and the new jobs are not yet released), it returns "-1". Once we know which task can execute its task, we must dinstinguish four cases. Before explaining these cases, just a reminder of the difference between **task_number** and **task_to_execute**. We can see the first one as the "past" (at time t-1) and the second one as the "present" (at time t).

4.2.1 Case 1

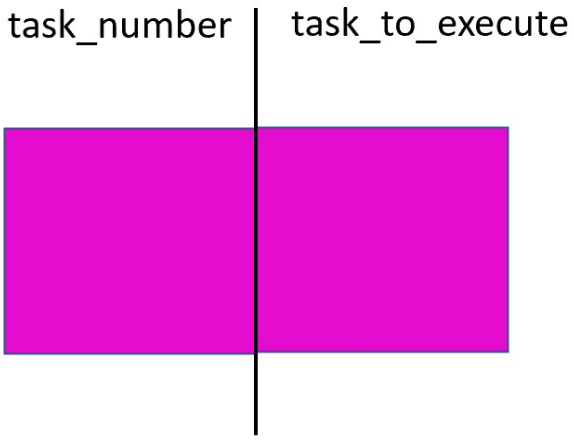

Figure 1: Case 1

It is the case when the job executed at time t-1 is the same as the job executed a time t (see line 17 - 18 in the algorithm). It is the easiest case. Indeed, we just need to increment the duration of the execution of job because the job will be executed one more unit of time.

4.2.2 Case 2

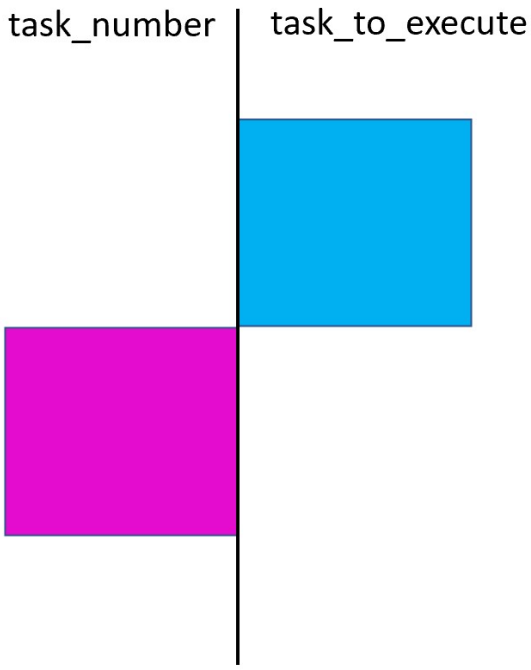

Figure 2: Case 2

It is the case when at time t-1 the job of a task is executed and at time t the job of another task is executed (see line 10 - 13 in the algorithm). In this case, we add a tuple composed of the start time of the job (given by the variable **job_start**) and its duration (given by the variable **job_duration**) in a list belonging to the task whose the job was executed at time t-1. We reset the value of the two variables (**job_start** = time and **job_duration** = 0) for the job which will be executed at time t.

4.2.3 Case 3

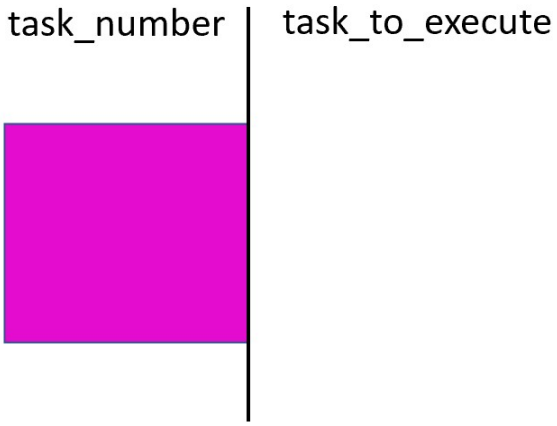

Figure 3: Case 3

It is the case when a time t-1 the job of a task ended and a time t, there is no tasks that can execute their job (see line 22 -25 in the algorithm). So the value of **task_number** is the task number which has its job executed at time t-1 and **task_to_execute** is -1 (because no tasks can execute their job). In this case, we add a tuple composed of the start time of the job (given by the variable **job_start**) and its duration (given by the variable **job_duration**) in a list belonging to the task whose the job was executed at time t-1. We also set the values of **job_duration** and **job_start** at 0.

4.2.4 Case 4

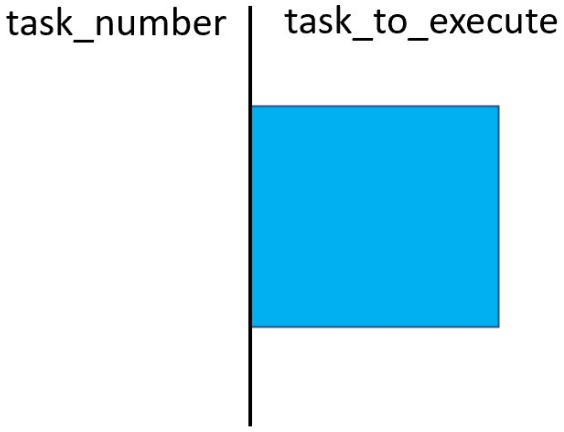

Figure 4: Case 4

It is the case when no job is executed at time t-1 (it means that the value of **task_number = -1**) and a job can be executed a time t (it means that the value of **task_to_execute** is the number of the task which can execute its job). We just need to set the values of **job_start** at time t and **job_duration** at 1 (see line 14-16 in the algorithm) because a new job starts its execution.

4.2.5 Case 5

If the values of **task_number** and **task_to_execute** are -1, we do nothing (see line 20-21 in the algorithm).

4.3 Result of an execution

Consider that we have four tasks to schedule and they are ordered by priority :

Task 1 (Offset = 0, WCET = 10, Deadline = 50, Period = 50)
Task 2 (Offset = 0, WCET = 20, Deadline = 80, Period = 80)
Task 3 (Offset = 0, WCET = 10, Deadline = 100, Period = 100)
Task 4 (Offset = 0, WCET = 50, Deadline = 200, Period = 200)

Thus the priority of the tasks is : Task 1 > Task 2 > Task 3 > Task 4.

The result of the scheduling is :

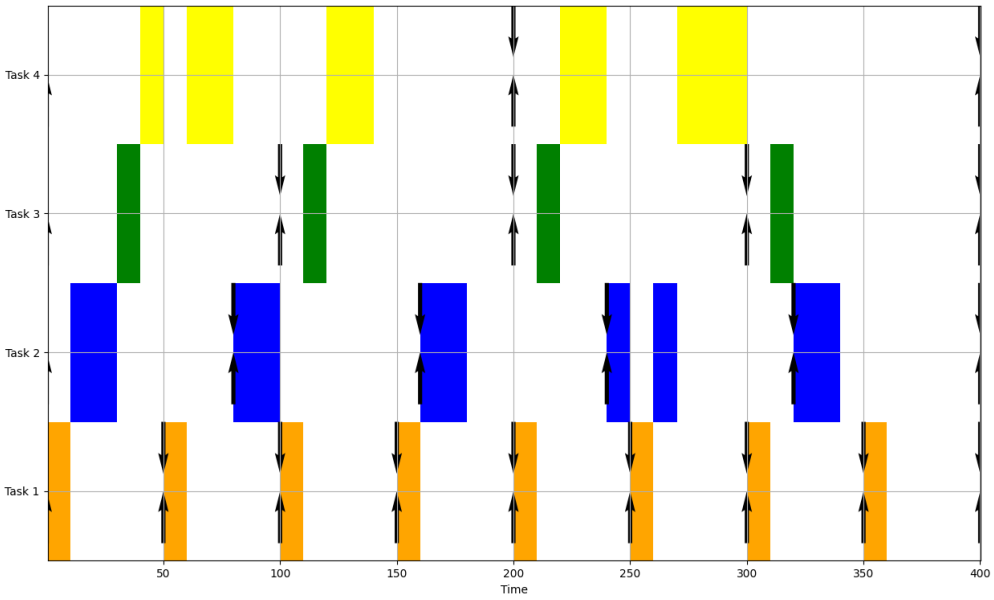

Figure 5: Scheduling 1

The arrow ↑ is when a job can start its execution and the arrow ↓ is the deadline of a job.

But what happens if a deadline is missed ? Consider the following task :

Task 1 (Offset = 0, WCET = 30, Deadline = 25, Period = 60)

This task will always miss its deadline. The result of the scheduling is :

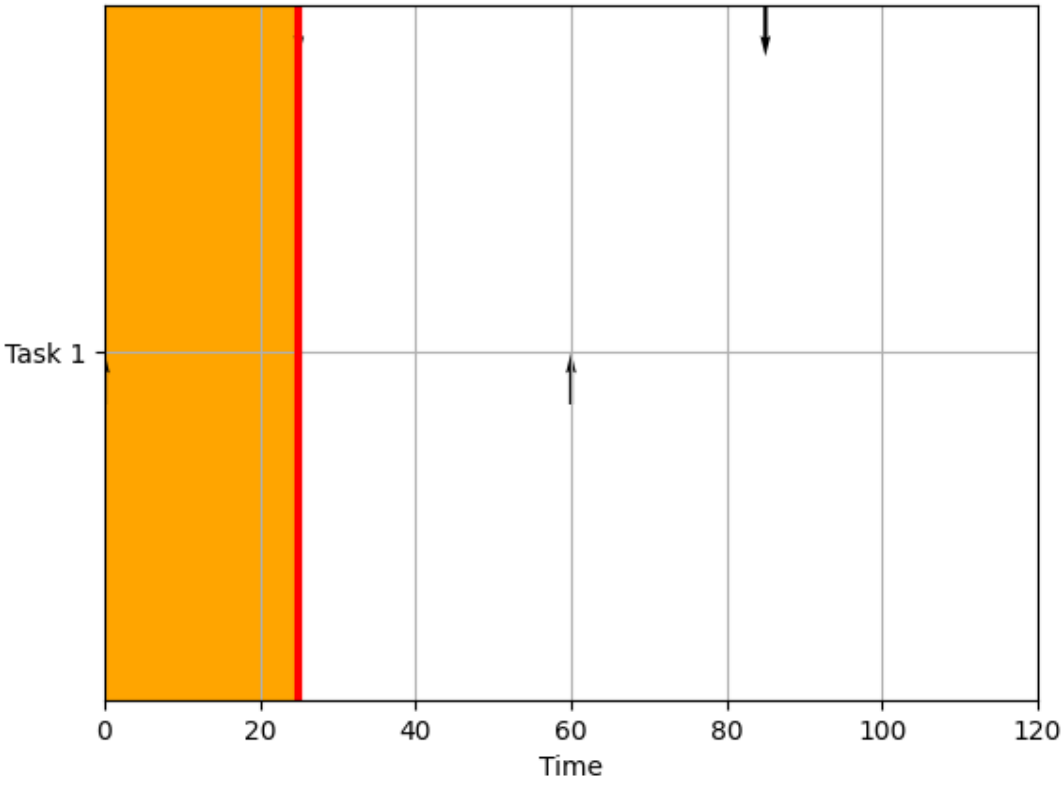

Figure 6: Scheduling 2

The read line shows that the deadline is missed.

5 Audsley's priority assignment (Vincent)

6 Difficulties

The first difficulty that we have met, is to create the visualize tool. We did not know which type of graphic to choose to represent the solution. We have decided to work with Gant diagrams. After that, we have read the documentation of the function that allows to do this kind of diagram. The method requires two arguments : a list of tuples (A,B) and a tuple composed of the minimum "y" coordinate and the maximum "y" coordinate. The "A" is the x coordinate of the start of the rectangle that will be drawn by the function and "B" is the size of the rectangle. Once we knew that, we had to find a way to keep track of the execution of the job (when they started and the duration of their executions). In order to do that, we have create a parameter for each tasks that allow us to save these informations.

Once we had our visual tool, we started to test our scheduler. We noticed some errors. In order to solve them, we thought about the different possible cases (we explain these cases at section 4.2). As soon as our scheduler worked well, we started to implement the Audsley algorithm. The implementation of this algorithm was not a problem for us because we understood well how its works. The only two problems that we met with this algorithm was that we forgot to set the tasks as hard tasks after each iterations (in order to find the lowest priority viable variable) and thus the priority assignment was always wrong. The second problem was about our solution. When we found a lowest priority viable variable, we inserted it with the method "insert(-1,the task)" and therefore our order was always wrong (the solution was to replace the -1 by 0).

7 Conclusion(Vincent)