INFO-F-404 – REAL-TIME OPERATING SYSTEMS

SCHEDULING PROJECT: AUDSLEY'S ALGORITHM

EL MIRI Hamza
000479603
KEZAI Wassim
000357375

31/10/2020

Contents

1 Introduction

The objective of this project is to design and implement a *Fixed Task Priority* scheduler simulator.

Audsley's priority assignment algorithm for asynchronous constrained deadlines task sets, will be used in order to determine whether there exists a feasible priority assignment for the provided task set.

Thus, the implementation will be divided into three major components: a task generator, a scheduler, and an implementation of Audsley's algorithm.

We will detail the work done on the subject, as well as the implementation process and the issues encountered. We'll also go through the file's architecture and how to run the software though command line.

2 User guide

2.1 Library used in the projects :

We utilized three libraries: Numpy, Matplotlib, and Random. To install the package, run the following command using pip:

- pip install numpy.
- pip install matplotlib.
- pip install random.

2.2 Generating tasks:

To run the project, we must first generate the task file, which is done by `Generator.py`. We must also specify the following parameters:

- Number of tasks to generate
- The name of the output file
- The maximum offset range
- The maximum deadline range
- The maximum period range

We can execute the generation of the tasks with this command :

```
Python Generator.py "task_number" "task_file" "max_offset"
                    "max_wcet" "max_period".
```

2.3 Execution of the Main program:

After the task generation, we may start either audsley or the scheduler by executing the `Main.py` file.

2.3.1 Execution of audsley :

The software should output whether or not a feasible FTP assignment was identified. If an FTP assignment is discovered, the `audlsey.txt` task file is created.

This is launched by issuing the following command:

Python Main.py `"audsley" "task_file"`

2.3.2 Execution of the scheduler:

In this section, we will run the FTP scheduler simulator with the specified task set (through a command line argument) if an FTP assignment was found, a graphical visualization of the scheduling will be displayed.

This is launched by issuing the following command:

Python Main.py `"scheduler" "task_file"`

3 Implementation

3.1 Overview

The implementation consists of four main classes:

- Task: The task representation
- Job: The job representation
- Scheduler: The scheduler implementation
- TimeUnit: A utility class used by Scheduler to keep track of new job releases

Figure 1 illustrates their relationships.

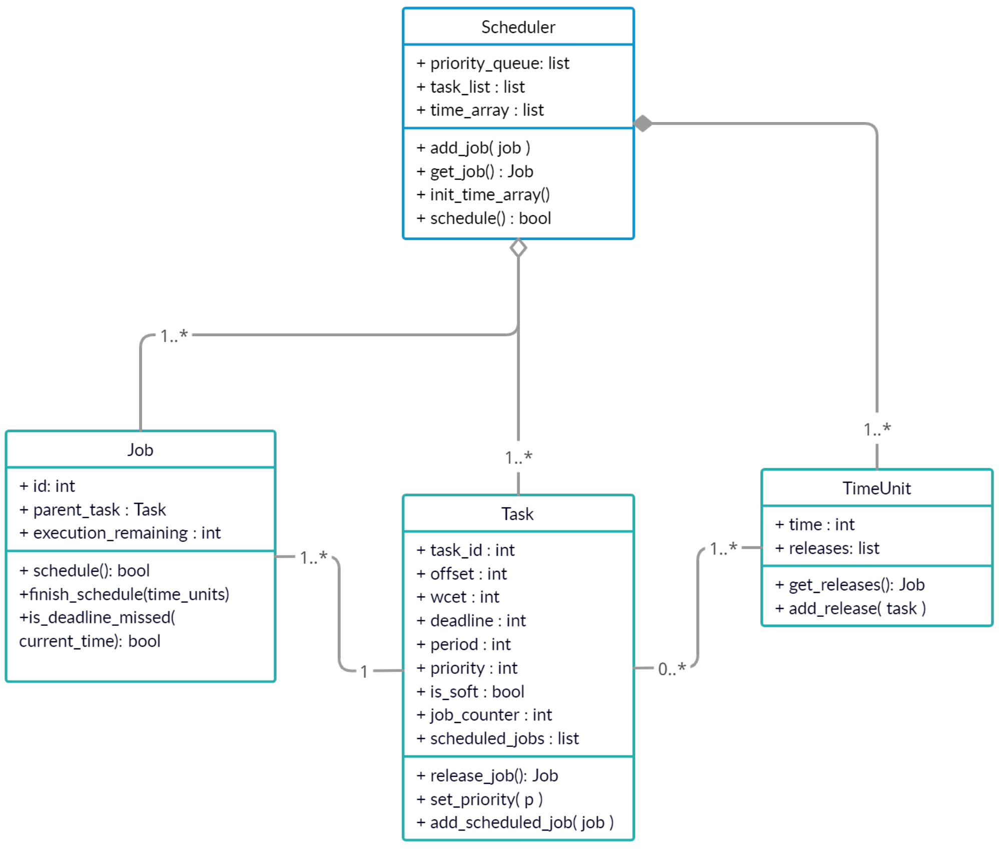

Figure 1: Class diagram

3.1.1 The Task class

It keeps track of its job releases using a counter to ensure that no two jobs released by the same task have identical identifiers which will allow us to calculate a deadline miss while scheduling.

This functionality means that it is important for all references to the task to point to the same instance which was very easy to implement thanks to Python's reference assignments and shallow copies.

At the end of the scheduling, the list `scheduled_jobs` will contain lists of tuples where each list represents a job and each tuple inside a list represents the execution of that job following the format (T, D) with T being the start time and D its duration.

For example the following job: $[(1,5),(9,3),(15,1)]$ has executed during 5 units at time 1, then during 3 units at time 9 and finally, by 1 unit at time 15. We have chosen this representation because if simplifies the plot generation.

3.1.2 The Job class

Each job released by a Task keeps track of its own execution internally while being scheduled. This implementation was chosen because it massively simplifies the scheduler implementation.

In addition, since each job has a unique identifier ranging from $0 - n$, we can calculate a deadline miss using the task information with the following formula:

offset + (job ID * period) + deadline - job's remaining execution - current time.

- **offset + (job ID * period)** will compute the period we're in.
- **+ deadline** to get where the deadline is in time
- **- job's remaining exec - current time** allows us to know whether the job will finish its execution before the deadline or not.

If the result is negative, the deadline has been missed. For example, suppose we're in time step = 110, a Job with ID = 2 which has executed for 0 units and its parent task with offset = 0, deadline = 30, WCET = 10 and period = 50.

The first first part of the formula will yield a result of: $0 + 2 * 50 = 100$, this is when the job was released.

The second part will give us the deadline at time step $100 + 30 = 130$

Finally, $130 - 10 - 110 = 10$ the result is positive and therefore the deadline won't be missed if the job starts being scheduled at time step 110.

If we were at time step = 121 however, the result would be $130 - 10 - 121 = -1$ meaning the deadline will be missed. This allows the scheduler to know beforehand whether to continue scheduling or immediately stop.

3.1.3 The TimeUnit class

It's a very simple utility class used by Scheduler which contains an identifier corresponding to the time step it represents and a list of references to tasks if there are any job releases from those tasks at that time step

3.2 The scheduler

At the moment of initialization, the scheduler will initialize an array of TimeUnits for the interval $[0, O^{max} + 2 * P]$. Each TimeUnit will have a unique identifier starting from 0. This array will represent the timeline.

Next, each task to be scheduled will be appended to the TimeUnit's releases list that correspond to their offset + period * n. This will allow the scheduler to know of the existence of new releases when it reaches that time step.

Finally, the scheduler uses a priority queue to determine which job executes first. Since we chose to implement this queue as a heap, the job who's parent task has the lowest number will have the highest priority.

Since jobs from different tasks cannot have the same priority and 2 jobs from the same task cannot be present in the queue at the same time (otherwise the deadline would have been missed), popping and reinserting the highest priority job is deterministic; the queue will return to its initial state.

The scheduling proceeds as follows. Since some implementation details are being delegated to the other classes, the implementation is quite simple.

Algorithm 1 Scheduler

```
for time in array do
    for new release in time.getReleases() do
        Add release to priority queue
    end for
    Get job with highest priority from the queue
    if The job is different than the previous then
        Finish scheduling the previous job
        -> scheduled sequence will be saved by its parent task
    end if
    Schedule the job for 1 time unit
    if job has execution remaining then
        Push it back to queue
    end if
end for
```

3.3 Audsley's algorithm

If we consider all possible FTP-priority assignments as a first naive but optimal priority assignment, we get n! in the worst-case scenario. .

Audsley proposes an efficient algorithm that takes into account, in the worst-case scenario, $O(n^2)$ FTP-priority assignments.

We've opted for a straightforward recursive implementation.

Algorithm 2 Audsley

```
Requires a list of Task objects
for task in task list do
    Assign lowest priority and set hard deadline
    Assign higher priorities to the other tasks soft deadline
end for
if Assignment is schedulable then
    Remove task from list
    Audsley(task list)
end if
```

4 Difficulties

Several difficulties arose throughout the implementation.We can summarize them in the following sentences:

- The first issue involved the scheduler and visualization tools. We experienced a difficulty when we switched jobs.

  More precisely, if there are no jobs to schedule, we update the previous job and alter the state of the task now under computation; otherwise, iterations continue until a new job release is reached.

  The issue was that the prior job still contained a reference to a job that had completed scheduling. As a workaround, we added a condition that will pad the idle time with the execution of that job.

- The second issue encountered was the software's performance in terms of memory usage and CPU performance.

  Because the scheduler initializes an array the size of the interval, if the tasks' hyperperiod is too large, the array that will be created will consume a significant amount of memory.

- Since each task carefully manages its own state, mainly the job counter and the list of successfully scheduled jobs, it is important that these parameters are carried along when needed and also cleared when not.

  This behaviour is desirable for our implementation because it simplifies the code in certain parts, however, it can also add unneeded complexity in others.

  For example, when Audsley's algorithm assigns priorities to the tasks passed to it as a list, at the end of the execution, the tasks will have the correct priorities assigned and can be directly used in the scheduler.

  On the flip side, during the execution of the algorithm, the internal state of the tasks must be cleared at each iteration to avoid conflicts. Creating copies of objects would fix the issue but it would further increase the memory usage.

5 Conclusion:

We can conclude that we were confronted with various unknown elements, such as the issues mentioned above, but we found an appropriate solution and successfully implemented the scheduler and priority assignements using audsley.

This project helped us enhance our knowledge of the way that Audsley algorithm works and real-time operating systems in general.

This assignment served as good classroom practice, allowing us to exercise and apply theoretical concepts.

We may conclude that we thoroughly discussed the work done on the topic, as well as the implementation process and the challenges encountered. We also discussed the solutions that were presented.

And finally We went through the architecture of the file and how to launch the software from the command line.