UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

INFO-F404
REAL-TIME OPERATING SYSTEMS

Project 1 – Audsley

Author:
Hakim BOULAHYA
Youcef BOUHARAOUA

December 21, 2017

Contents

1 Feasibility interval **2**

2 Simulator **2**
2.1 Periodicity of the requests 2
2.2 Pending jobs 3
2.3 Events 3
2.4 Hard vs Soft simulation 4
2.5 Output 4
2.6 Schedule plotter 4

3 Lowest-priority viable **5**

4 Audsley algorithm **5**

5 Generator **6**

6 Difficulties **7**

7 How to run **8**

1 Feasibility interval

To find the feasibility interval, we implemented a function that finds the least common multiple of a list of task periods. When this is found, it is just needed to find the maximum offset to produce the feasibility interval $[O_{max}, O_{max} + 2P]$. Figure 1 shows the implementation of the feasibility interval calculation.

```
1  def hyper_period(tasks):
2      if len(tasks) == 0:
3          return None
4      elif len(tasks) == 1:
5          return tasks[0][T]
6      return lcm(*[task[T] for task in tasks])
7
8  def interval(tasks_file):
9      tasks = parse_tasks(tasks_file)
10     omax = max([task[O] for task in tasks])
11     P = hyper_period(tasks)
12     print(omax, omax + (2 * P))
```

Figure 1: Functions of the interval calculation

2 Simulator

Implementation choices The simulator of the single FTP simulator is implemented in a discrete manner from `start` to `stop`. For each time steps we execute functions that will update the current running job information, the different requests (arrivals or deadlines) and will store them in data sets. We choose not to print the different actions during the simulation, since the simulation is used for other commands such as the plotter or the audsley algorithm. The output can be produced by a call to the method `FTPSimulation.output()`.

2.1 Periodicity of the requests

Arrivals periodicity The important part of the simulation is to detect the arrivals *i.e.* the job requests and the deadlines of those jobs. We know that the job requests for each task are periodic. Let $\tau_i = (O_i, T_i, D_i, C_i)$ a periodic task. We know that each job request of τ_i have a periodicity of T_i with the first request made at time O_i. We can detect if a request for τ_i occur at time t using the following formula:

$$[(t - O_i) \geq 0] \ \ and \ \ [(t - O_i) \ mod \ T_i = 0] \tag{1}$$

The left-side formula is the detection of the first request. If $t - O_i$ is bigger than 0, then it means that it is possible that τ_i request a job. The right-side formula is the detection of a request at time t. To detect which job it is we can make an entire division of $t - O_i$ with the period T_i. Figure 2 is a excerpt of the arrival detection.

```
offset, period = self.tasks[task_id][O], self.tasks[task_id][T]
cond = (t - offset >= 0) and ((t - offset) % period) == 0
job_id = (t - offset) // period  # module == 0 i.e. no decimals
return (job_id, True) if cond else (None, False)
```

Figure 2: Function of the arrival detection for a task at time t

Deadlines periodicity For each job requested job there is a deadline associated *i.e.* the deadline is also periodic. To detect if a deadline occur at time t of a task τ_i we can use the same formula (1), and testing the formula a time $t - D_i$. This method actually detect if a job was requested, and we are a the requested time plus the deadline, the it is the deadline of this job. Figure 3 shows a excerpt of the deadline detection.

```
deadline = self.tasks[task_id][D]
return self.is_arrival_for(t - deadline, task_id)
```

Figure 3: Function of the deadline detection for a task at time t

2.2 Pending jobs

To be able to process requested jobs, the simulator uses a list of list to handle pending jobs. A sublist per task exists. When a request of a task occurs, the new job is added to the pending jobs sublist of the task. The sublists are ordered by task priority. Therefore the job to process is the first job of the first sublist. If this list is empty, then it process the first job of the second list, etc. When a job finished its computation *i.e.* has been processed C_i time, then it is removed from the pending jobs list. Figure 4 shows the snippet of the function that returns the job to be processed.

```
for sub_jobs in self.pending_jobs:
    for job in sub_jobs:
        return job
return None
```

Figure 4: Job to be processed detection

2.3 Events

For each time steps, the simulator stores relevant information that can be used to output information on the standard output or in a plot. Those information are stored in an object `Event` which is composed of actions that occurs at time t:

- The job requests
- The deadlines (and the missed deadlines)
- The computed job

2.4 Hard vs Soft simulation

It is possible to configure the simulation to consider the deadlines as soft or hard. If the soft configuration is chosen, the simulator will not stop at deadline misses. If the hard configuration is chosen, the simulator will stop at the first deadline miss (see section 7 to see how to run each configuration).

2.5 Output

The output production is a post process function. It has been implemented like this to avoid printing while doing the simulation because the simulation is used by other part of the project such as the implementation of the Audsley algorithm or the schedule plotter. The `output()` method uses the events of the simulator (section 2.3).

The simulator provides two approach to the output. The default mode is the **request mode**. It provide an output as shown in Figure 5b. This mode output the requests, such as arrivals or deadlines, as soon as they occur. So it does output even if a job is processing during a block. This mode has been implemented to provide a more continuous approach to the simulation.

The second approach is the **preemptive mode**. As explained above the process block of a job can be divide in multiple blocks if requests occur during the process. To propose an output that characterise more the preemptive property of a FTP simulator, the preemptive mode provide a preemptive results as shown in Figure 5c.

Section 7 explained how to run the different modes.

```
0 5 10 5
0 3 20 2
```

(a) Tasks set to output (O_i, T_i, D_i, C_i)

```
0: Arrival of job T1J1
0: Arrival of job T2J1
0-3: T1J1
3: Arrival of job T2J2
3-5: T1J1
```

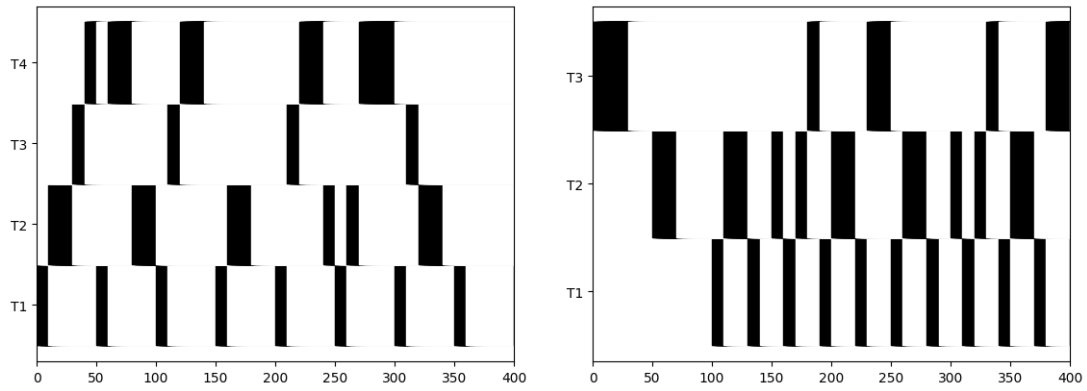
(b) Requests mode output

```
0: Arrival of job T1J1
0: Arrival of job T2J1
0-5: T1J1
3: Arrival of job T2J2
```

(c) Preemptive mode output

2.6 Schedule plotter

The plotter is implemented in the simulation. After running the simulation the method `FTPSimulation.plot()` will produce a file **scheduler.png**, with the result of the simulation. The computed jobs data that are gathered during the simulation are used to plot the result. Figure 6 shows example of plots.



(a) Plot of tasks.txt

(b) Plot of audsley.txt

Figure 6: Plot examples

3 Lowest-priority viable

A task τ_i is lowest priority viable if and only if and only if all his jobs respect the deadlines when :

1. τ_i has the lowest priority
2. The others tasks has a higher priority (the order of thus ones does not influence τ_i)
3. The schedule continues until completion (soft deadlines) $\forall \tau_j$

Following this definition, the algorithm $lowest - priority - viable$ which takes a list of tasks an id and an interval in parameters, check if the id of task given on parameter is LVP by:

- Putting the task with given id at the tail of the list (to respect the first point of the definition)
- Ruining a soft simulation and check if τ_i missed a job without worrying about the other tasks (which respects the second part of the definition)

And to do this we have a list of integers which represent the number of missed jobs for every task when the simulation is running.

4 Audsley algorithm

The classic Audsley algorithm determines a lowest priority viable task in the sub-set $\delta\{\tau_i$ of $n-1$ tasks (i.e., assigning priority level $n-1$). The priority assignment of the Audsley algorithm may leads to a scheduled system (i.e., priority assignment succeed). But in this project we were asked to implement a variant of this algorithm that finds all scheduled system for a given set of tasks. The excerpt in Figure 7 shows the implementation of the variant algorithm.

```
1  def audsley_search(first, last, tasks, includes, level=0):
2      for task_id in includes:
3          sub_tasks, sub_task_id = filter_tasks_list(tasks, task_id,
               includes)
4          if lowest_priority_viable(sub_tasks, sub_task_id, first,
               last):
5              print((" " * level) + "Task \%d is lowest priority
                   viable"
6                     \% (task_id + 1))
7              new_indices = includes[:]
8              new_indices.remove(task_id)
9              audsley_search(first, last, tasks, new_indices,
10                                level + 1)
11         else:
12             print((" " * level) + "Task \%d is not lowest priority
                   viable"
13                    \% (task_id + 1))
```

Figure 7: Audsley search algorithm

The variable level represents the level of the recursion and the procedure takes a list of all tasks and another one without the LVP task, the algorithm will print the tasks by levels of recursion too as explained in the statement of the project. Figure 8 shows an example of output that the function produce.

O_i	T_i	D_i	C_i
100	30	20	10
50	50	50	20
0	150	100	30

(a) Tasks set

```
Task 1 is not lowest priority viable
Task 2 is lowest priority viable
 Task 1 is not lowest priority viable
 Task 3 is lowest priority viable
  Task 1 is lowest priority viable
Task 3 is not lowest priority viable
```

(b) Output of audsley_search function

Figure 8: Audsley results

5 Generator

We were asked to implement a generator of periodic, asynchronous systems with constrained deadlines based on two parameters (the utilisation factor and a number of tasks to generate), the class *Generator* using the generation method *generate* produce the set of tasks. The function create tasks, then checks if this set has the utilisation required if it's the case this set is returned,

if not another random set of tasks is created. For a coherent system the parameters of every task must respect this inequality:

$$C_i \leq D_i \leq T_i$$

where, $\forall\ task_i$:

C_i = worst case execution time (WCET)
D_i = deadline
T_i = represents the period

knowing that the task utilisation and the system utilisation are respectively:

$$U(\tau_i) = C_i / T_i$$

$$U(\tau) = \sum_{\tau_i \in \tau} U(\tau_i)$$

Remark: utilisation of the generated system does not have to be equal to the given percentage but should be very close to it. so the result respects this inequality:

$$U(\tau)_{given} - 2 \leq U(\tau)_{founded} < U(\tau)_{given}$$

Therefore from what was explain above the randomness of the parameters of every task follow this logic:

$O_i \in$ [0,300] The maximum value of the offset were arbitrarily fixed to 300).
$T_i \in$ [1,100] (The maximum value of the period were arbitrarily fixed to 100).
$C_i \in [1, T_i]$
$D_i \in [C_i, T_i]$

6 Difficulties

Except for the simulation there were no real difficulties. It is obvious since most of the part of the project are based on the simulation such as the audsley algorithm that uses the simulation to determine the lowest priority of tasks.

About the periodicity of the simulation, There were two possible implementation to simulate the scheduler in a interval given as a parameter. The first idea was to simulate the scheduler between the feasibility interval $[0, S_n + P)$, but it would be tedious to do it like this, since we would have to find the corresponding interval in $[0, S_n + P)$ of the $[start, stop)$ interval of the simulation. The second possibility is to use the periodicity of the tasks to determine the requests and deadlines of jobs at a specific time step. For simplicity of implementation and understanding we chose the second idea.

About audsley algorithm, the real difficulty was actually a way to determine the lowest priority viable of a specific task with the corresponding set. To do so we just implemented in the simulation a way to continue the simulation if when there is a missed deadline. This way it is possible to detect if a task is LVP only by checking during the simulation if this task had missed deadlines. The audsley algorithm itself was obvious to implement since it is just running recursively the LVP checks.

7 How to run

The project is implemented in `python3`.

Graphical library The library used for the graphical part (the plotter) is **matplotlib**. To install it run (need root permission):

```
$ pip install matplotlib
# or
$ pip3 install matplotlib
```

To install the library locally, the best way is to use a virtualenv:

```
$ .local/bin/virtualenv rtosp1
$ source rtosp1/bin/activate
$ pip install matplotlib
```

Then switch to the project directory and run the project.

Commands The different commands to run the project are listed below, with the additional arguments:

```
$ python3 project.py interval [-h] tasks_file
$ python3 project.py sim [-h] [--preemptive] [--hard] start stop
    tasks_file
$ python3 project.py plot [-h] start stop tasks_file
$ python3 project.py audsley [-h] first last tasks_file
$ python3 project.py gen [-h] number_of_tasks utilisation_factor
    tasks_file
```