CHAPTER 16 SIMULATION EXERCISES

16.1 Introduction

In this session, we lay down the basics of uniprocessor scheduling simulation. For some scheduling such as EDF (4.1, page 34), simulation is a required step to check the feasibility of a taskset.

We introduce two two types of simulators: constant step ones and event oriented ones. The former category use a constant step δt to sequentially increase the time and check for new events at each step. In contrast, the latter category of event oriented simulators precomputes all events that can happen and goes through time from one event to the other. Note that constant step simulators can only represent discrete time while event oriented ones do not have such limitation.

In this practical session, we implement a step-based simulator. You can use your favourite programming language, but be aware that solutions are only provided in Rust and in Python.

16.2 Constant step simulation

A constant step scheduling simulator requires as input a set of tasks, a function to priorise jobs and a time step at which to end the simulation.

Let us note the k^{th} job that task τ_i as released at time t by the tuple $\langle k, c, t, \tau_i \rangle$, where c is the remaining computation time. The notation $J.c$ refers to the remaining computation time of job J.

Algorithm 8 shows a high level view of the main simulation loop. It assumes the existence of other functions such as `deadline_missed` or `is_complete`.

Algorithm 8 Constant step simulation

```
$\tau \leftarrow \{\langle O_1, C_1, D_1, T_1\rangle \dots, \langle O_n, C_n, D_n, T_n\rangle\}$    ▷ The taskset
$P \leftarrow$ a priorisation function (e.g.: RM, EDF, ...)
jobs $\leftarrow \{\}$    ▷ The job queue
$t \leftarrow 0$    ▷ The current time step
**while** $t < t_{\max}$ **do**
    jobs $\leftarrow$ jobs $\cup$ release_jobs($\tau$, $t$)
    **if** deadline_missed(jobs, $t$) **then return** Error
    **end if**
    $J \leftarrow$ P($t$, jobs)    ▷ Elect a job to run
    **if** $J \neq$ None **then**
        $J.c \leftarrow J.c - \delta t$    ▷ Decrease the remaining computation time
        **if** is_complete($J$) **then**
            jobs $\leftarrow$ jobs $\setminus \{J\}$    ▷ Remove from the job queue
        **end if**
    **end if**
    $t \leftarrow t + \delta t$    ▷ Increase the time
**end while**
```

16.3 Exercises

Based on Algorithm 8, the following exercises aim to guide you step by step throughout the implementation of a constant step simulator.

Exercise 20. Implement basic structures for `Task`, `Job` and `TaskSet`. What are the relevant attributes of these structures?

Exercise 21. Implement a method `release_job` for `Task` that takes as input the time t and returns a `Job` if the task releases a job at time t and nothing otherwise. Create a few tests to make sure your method works as expected.

Exercise 22. For the `Job` structure, create a method

- `deadline_missed` for `Job` that takes as input the time t and returns whether the deadline has been missed.
- `schedule` to schedule a job for a given amount of time.

Exercise 23. Create a `rate_monotonic` function that takes as input a set of jobs and returns the job with the highest priority. RM is an FTP algorithm, do not hesitate to go back to Section 15.1, page 92 in order to check how priorities are assigned.

Exercise 24. Create a `schedule` function that takes as input a `TaskSet`, a scheduling function and a time step $t_{\max}$ as input. The scheduler should loop from $t = 0$ to $t = t_{\max}$ with a step of 1, and at each step

1. check for deadline misses
2. check for new job releases
3. if any, schedule the job with the highest priority

If a deadline is missed, the scheduler should show a message and stop early.

Exercise 25. Once your simulator is working, implement the EDF scheduling algorithm (Section 4.1, page 34). EDF is an FJP scheduler (and not)

Exercise 26. How can you improve the efficiency of your simulator? So far, we have increased the time by one step at a time, but is that really efficient? What size of step could you safely take without missing any event?

Answer

For synchronous systems, a very straightforward improvement would be to change the constant step to the greatest common divider of periods, computation times and deadlines. It would still catch every event but the step could be greater than 1.

An other possibility is to use event-based simulation whose principle is to generate the series of events ahead of time and then to iterate through it.