INFO-F-404: Real-Time Operating Systems

Multiprocessor Scheduling Project

2024-2025

Joël Goossens Yannick Molinghen

1. Introduction

In this second project, we consider the case of asynchronous tasks with arbitrary deadlines in the scope of multiprocessor scheduling with m identical cores. The objective of this project is to build a multi-threaded (or multi-processed) tool that checks the schedulability of task sets according to different real-time multiprocessor scheduling algorithms.

2. Project details

This project should be realized by group of two. There are two deliverables: the code itself preferably written in Rust or in Python[1] and a PDF report.

2.1. Task set file

Your program has to parse task set files. A task set file is a file that contains a description of a set of tasks with their respective offset O_i, computation time C_i, deadline D_i and period T_i. Task set files are simple Comma Separated Value (CSV) files where the line i encodes the O_i, C_i, D_i, T_i of task i. An example is shown here below where $\tau_1 = \{O_1 = 0, C_1 = 2, D_1 = 40, T_1 = 50\}$, and $\tau_2 = \{O_2 = 10, C_2 = 80, D_2 = 200, T_2 = 200\}$.

```
0,  20,  40,  50
10, 80, 200, 200
```

2.2. Scheduling

You have to implement both partitioned and global scheduling algorithms.

2.2.1. Partitioned scheduling

There are two components to partitioned scheduling:

- a partitioner to partition the task set among the m processors according to a combination of sorting order and heuristic
- a local prioritization algorithm to prioritize jobs at simulation time. The prioritization algorithm is identical on all cores.

For the partitioner, you have to implement First Fit, Next Fit, Best Fit and Worst Fit heuristics and you should be able to order task by increasing of decreasing order of utilization. For instance, you might use the WF (worst fit) and DU (decreasing utilization) combination to partition the tasks among the cores.

For the scheduling algorithm, you have to implement EDF. You should be able to reuse the same scheduler as the one of the first project (provided you already took offsets into account).

[1]Contact me if you want to use another language.

2.2.2. Global EDF scheduling

You have to implement global EDF. With global scheduling, task migrations (from one core to an other) is allowed and you can neglect the migration time. Global EDF schedules on m cores the m jobs with the earliest deadline.

2.2.3. EDF$^{(k)}$

You have to implement the EDF$^{(k)}$ algorithm. Refer to the course material for a complete explanation of the algorithm. Note that k is given as user input and does not have to be computed.

2.2.4. Schedulability of a system

There exist different ways of determining the schedulability of a system given an algorithm, some require system simulation, others don't.

You should identify the cases where
- some necessary condition is not met
- a sufficient condition is met
- or simulation is required.

When simulation is required, you should simulate the system in the smallest time interval possible and check for deadline misses. It is your task to correctly identify the smallest feasibility interval.

Important: There exist no proven feasibility interval for asynchronous tasks with arbitrary deadlines for global scheduling. In that case, we ask you to use $[O_{\max}, O_{\max} + 2P)$ to check for deadline misses, but bare in mind that it does not guarantee that the system is schedulable.

2.3. Parallelization

Whenever possible and relevant, you should parallelize the run your code on multiple cores.

3. Running the program

3.1. Program inputs

The program should accept the following arguments:
- the task set file to consider
- the number of cores m
- the version of EDF to use (partitioned, global, or EDF$^{(k)}$)
- the number of workers w to run the simulation (set the number of cores of the computer by default)
- in the case of partitioned scheduling
  - the heuristic to use (first fit, next fit, ...)
  - the ordering of tasks (increasing or decreasing utilization)

```
# With Python
$ python main.py <task_file> <m> -v global|partitioned|<k> [-w <w>] [-h ff|nf|bf|wf] [-s iu|du]
# With Rust
$ cargo run <task_file> <m> -v global|partitioned|<k> [-w <w>] [-h ff|nf|bf|wf] [-s iu|du]
```

Advice: We strongly advise that you use a dedicated library (such as `clap` in Rust or `argparse` in Python) to parse command line arguments. These libraries make your life easier and your program much more robust to argument parsing.

3.2. Program outputs

The program should exit with different values depending on how you defined the schedulability of the system. The exit codes and their meanings is shown in Table 1.

Exit code	Description
`0`	Schedulable and simulation was required.
`1`	Schedulable because some sufficient condition is met.
`2`	Not schedulable and simulation was required.
`3`	Not schedulable because a necessary condition does not hold.
`4`	You can not tell.

Table 1: Exit code for each case.

Important: To specify the exit code of your program, you can use `std::process::exit(<code>)` in Rust or `exit(<code>)` in Python. If your program does not exit with the appropriate code, it will not pass the automatic tests and your project will be poorly graded !

4. Report

Write a report with a tool designed for scientific writing such as Typst or Latex where you discuss at least the following topics (not necessarily in this order):

- Parallelization:
  - What can be parallelized and what can not?
  - What did you choose to parallelize, and on what criteria did you choose so? Explain your methodology.
  - Consider the dataset of task sets provided on the UV. Measure the average execution time your program with partitioned EDF, BFDU and $m = 8$ when you vary the number of workers from 1 to 32. Draw a plot of the average program execution time according to the number of workers and discuss the results by referring to the theory of parallelization.
- Explain and represent visually (possibly on multiple figures) the process of determining which method to use to determine the schedulability of a task set, including the decision on the feasibility interval to use.

Feel free to include any additional matter that you think is relevant.

5. Evaluation criteria

The project will be graded out of 20 points balanced as follows:

- Scheduling algorithms are correctly implemented **/5**
- The code is readable, properly structured and uses appropriate abstractions **/3**
- Report
  - Conciseness, clarity and relevance of the content **/6**
  - Structure **/2**
- Global appreciation **/4**

You have to hand in your project as a zip file containing your code and the report on the “Université Virtuelle” no later than the 15th of December 2024, 23:59.

Questions: Feel free to ask questions during practicals or by email at yannick.molinghen@ulb.be.