RTOS-1

INFO-F404: Dual priority

Christopher Lample, Ali Dhanani, Samineh Sadat Naghavi

1. Introduction:

In this project, we study the scheduling of implicit deadline, periodic task sets using a dual priority scheduler. Each task is assigned to two fixed priorities, for before and after its promotion deadline. A job begins with the first priority and if not completed upon reaching its promotion deadline, continues its execution at its second priority.

Our implementation contains all functionality specified in the original problem statement: a random generator for task sets, an algorithm for finding optimal promotion deadlines, a simulator for the scheduler, as well as a visualisation of the schedule.

2. Program execution

In implementing the schedule plotter, we made use of the library Matplotlib [1]. This can be installed with:

$ pip install matplotlib

The project was developed and tested with Python 3.7. The command line arguments and file formats adhere exactly to the original problem statement. Documentation is available for all commands by specifying **--help**. For example:

$ python3 project.py --help
$ python3 project.py gen --help

3. Implementation

In addition to the description below, a flow chart of the entire program is available in code-diagram.jpeg.

3.1 Task Generation

To generate a system of random, periodic, asynchronous tasks with implicit deadlines, simply run:

$ python project.py gen <number of tasks> <utilization> <output file>

The tasks are output to the specified file with the format: **Offset; WCET; Period**.

When generating the tasks, the offsets are simply generated randomly. Special care had to be taken in generating both the worst case execution times and the periods, though, since these affect the utilization of the task set. Our solution made use of the fact that the total utilization is the sum of the utilizations for each individual task. With C_i representing the worst case execution time of task i and T_i representing its period, we have:

$$U(\tau) = \sum \frac{C_i}{T_i}$$

Our algorithm generates the worst case execution times and periods task by task. For the first task, we select a random utilization less than total utilization and also select a random period. From this, we can then directly calculate the worst case execution time. We then repeat this process for the next task using the remaining utilization (the total utilization minus the utilization of the existing tasks). This is continued until all tasks are generated.

To avoid having a value of zero for the worst case execution time, some additional constraints are used in selecting the random task utilization and period. When selecting a random task utilization, we not only make sure that it is less than the remaining total utilization, but we also ensure that there is at least 1% of utilization left for each of the remaining tasks.

Since the period and worst case execution times are integers, we also need to ensure that the randomly selected period isn't too small for the randomly selected task utilization. For example, if the task utilization is 10% and the period is selected to be 9 seconds, the worst case execution time would be rounded to 0 seconds. We avoid this by selecting a period with a minimum value of $100/\tau_i$ where τ_i is the task utilization in percent form. This ensures that the worst case execution time is at least 1 second.

The advantage of this algorithm is that the resulting tasks are random - we don't restrict the possible values except to obtain the specified utilization. The resulting total utilization is generally close to the specified value (aside from the rounding of floating point values to an integer).

3.2 Promotion Deadlines

The program also contains functionality for determining the promotion deadlines of task sets using the RM + RM dual priority scheduler, with the First Deadline Missed Strategy. It can be run with:

$ python3 project.py fdms <tasks file>

where the task file is in the same format output by the generator.

To determine the promotion deadlines, we followed the suggested algorithm in the original problem statement. We first assign tasks two priorities, using the RM + RM schedule. We

also set initial promotion deadlines equal to the period for each task. We then repeatedly simulate the task set, decrementing the promotion deadline and restarting when a job fails. We stop when a promotion deadline becomes negative, indicating an unfeasible task set, or when we successfully simulate until the hypperiod, indicating feasibility with the given promotion deadlines.

The relevant code can be found in the **scheduler.priority** module.

3.3 Simulation

The resulting schedule for a given task set can be determined with:

$ python3 project.py simulation <tasks file> <end time>

where the task file is in the same format output by the generator, and the end time is an integer.

To simulate the task set, we first generate the priorities and promotion deadlines using the same functionality described in the previous section. We then begin a loop corresponding to the time, which increments second by second. At each point in time, we determine if a previous job has failed, add new jobs which have been released, and finally execute the highest priority job for one second (removing it if it has finished). In order to display output to the user, we also keep track of which job has been run at each particular second.

The relevant code can be found in the **scheduler.simulation** module.

3.4 Schedule Plotter

In addition to the simulation mentioned in the previous section, our project also outputs the resulting schedule as a graph. This can be run with the command:

$ python3 project.py simulation_graph <tasks file> <end time>

The graph will appear in a new window, and the window will include an icon for saving the graph to a file.

The schedule plotter reuses the exact same functionality as the simulation described in the previous section. The only difference is that it displays the output differently. To generate the graph, we used the library Matplotlib, which appears to be very mature and widely used. For each execution of a job, we draw a corresponding horizontal bar in the graph. To make the separation between jobs more clear, we alternate between drawing jobs with blue and red. For example, all executions of job 0 task 1 will be blue, all executions of job 1 task 1 will be red, and so on.

The relevant code is in the **scheduler.io** module, in particular the **display_simulation_graph** function.

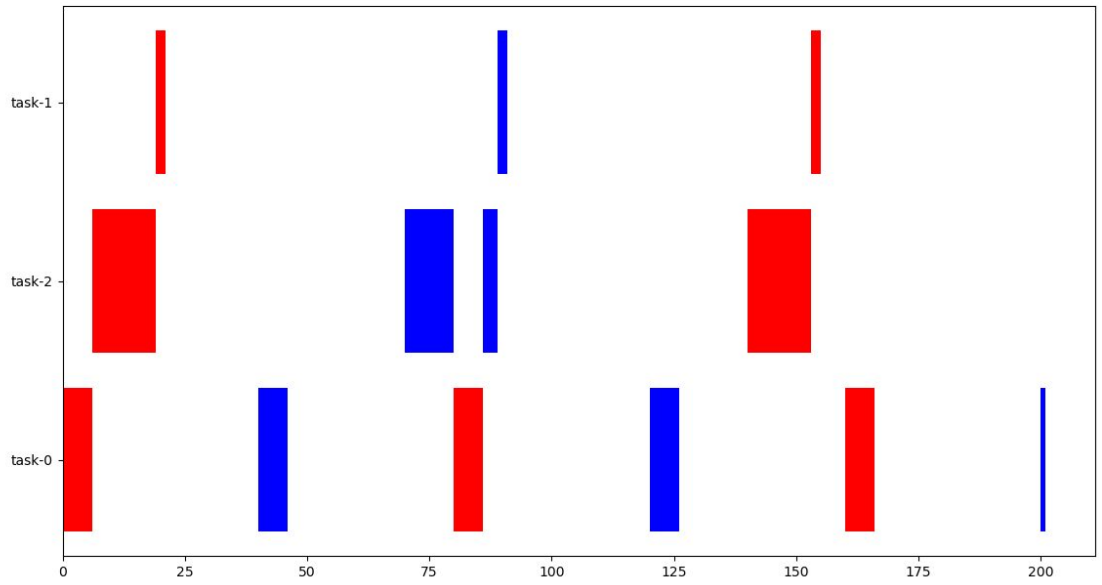

Figure 1. The simulation graph representing the graphical form of tasks over time.

4. Difficulties and Limitations

In implementing the promotion deadlines functionality, there were some unexpected issues related to our task generation. When determining the promotion deadlines, our program simulates the task set until the end of the hyperperiod, the least common multiple of the task periods. For a large number of tasks with long periods, this value can become extremely large. This caused our program to appear to run indefinitely, and it took some debugging to find the root cause. To prevent problems like this from occurring in our randomly generated task set, we reduced the maximum period to 100.

5. Conclusion

In this project, we studied the dual priority scheduling problem for synchronous, implicit deadline, periodic tasks. Our resulting program contains a variety of functionality. It allows for generating random asynchronous tasks, determining promotion deadlines using the First Deadline Missed Strategy, simulating the resulting schedule, and also displaying the

schedule as a graph. With this functionality, it would be possible to further investigate the First Deadline Missed Strategy and see how it behaves with a variety of task sets.

6. References

[1] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007