Files available at https://github.com/rtas25-paper74/rtas2025-paper74-artifact
Videos of the physical car tests at https://github.com/rtas25-paper74/rtas2025-paper74

ROS2 Preemptive Mode-Switching Executor demo

`cd` into `ros_exec_demo`, and build the container with the following command:

```
docker build -t rtas_eval .
```

Then, run the container:

```
docker run --privileged --cpuset-cpus="0" \
-v ./ros2_ws/results:/root/ros2_ws/results -it rtas_eval:latest
```

The `--privileged` flag is required to allow the callbacks threads to run with SCHED_DEADLINE. The `--cpuset-cpus="0"` flag is used to pin the container to CPU 0 (if your system has a heterogeneous architecture, you can change this value to the desired CPU core). The script will output a json file in the `results` folder with the results of the demo. The `analysis/analysis_demo.py` script can be used to render a timeline of the demo.

The demo and custom executor are implemented in the `preempt_executor` package. The executor is implemented in `preempt_executor/src/preempt_executor.cpp`, and the demo is implemented in `preempt_executor/src/run.cpp`. The demo creates two timer-triggered callbacks, `publisher1` and `publisher2`, both of which have a period of 200ms (in LO mode), and execution time of 40ms. Each publisher callback publishes to a worker callback, `worker1` and `worker2`, respectively. The worker callbacks have a period of 200ms (since they are triggered by the publisher callbacks), and an execution time of 40ms each. These four callbacks form two chains of two callbacks each, and results in a system with 0.8 utilization. An additional timer, `mode_switch_timer` is created with a 1000ms period (and has a negligible runtime). Each time this timer triggers, the executor switches from LO to HI mode, or from HI to LO mode. This extra task simulates an external event causing a mode switch. In HI mode, the period of `publisher1` is reduced to 100ms. In order to keep the system schedulable, `publisher2` is stopped. When the system switches back to LO mode, the period of `publisher1` is increased back to 200ms, and `publisher2` is restarted.

The user provides a mode-switch routine to the executor in the form of the `mode_switch_callback` function. The executor will call this function in the first idle instance after user code requests a mode switch by setting `requested_mode` and `requesting_mode_switch`. In this function, the user can change the timing properties of the callbacks, or perform any other necessary actions to switch modes. The executor runs all callbacks in their own thread, as described in the paper, and each thread is run under SCHED_DEADLINE, allowing preemption as needed.

The system runs for 10 mode switches, then stops.

The mode switches are clearly visible in the timeline generated by the `analysis/analysis_demo.py` script, which will appear in a window when run, and also output a "preempt_executor.pdf" file with the timeline.

We highly recommend running this *outside* of a VM, as the VM may interfere with the real-time performance of the system. While the demo performs reasonably well on a generic Linux kernel on a laptop, the real-time performance is significantly better on a real-time kernel, or on an headless system.

UPPAAL models

`bouncing_ball2.xml` re-creates the multi-mode ball-and-paddle system from the introduction. `schedule_test3_resp_test_{HI,LO}.xml` implements the timing models described in Model 1 and Model 2, and are configured to reproduce values from Table 1. `bouncing_ball2.xml` runs in UPPAAL 4.1.20, and `schedule_test3_resp_test` runs in UPPAAL 5.0.0. UPPAAL can be downloaded from here, and a license can be obtained from here.