Graduate Program in Computer Science and Engineering
DISI, University of Trento | TeCIP, Scuola Sant'Anna, Pisa

Real-Time Systems: course project

Scheduletrace: a schedule visualizer for Linux

Author: Davide Kirchner
`davide.kirchner@yahoo.it`
Student ID 167679 (UniTN)
Supervisor: prof. Giorgio Buttazzo
February 21, 2015

Contents

1 Introduction

This document describes `scheduletrace`, a utility to graphically display the scheduling of a set of ad-hoc periodic tasks running on a Linux system.

Section 2 provides an introduction to the program's features and its usage. Section 3 describes the program's architecture, detailing its main components and digging into some of the implementation issues that arose during its development. Section 4 shows the outcome of running a priority-inversion–prone task set with the three different available priority-inversion protocols.

2 Usage

The user interface articulates into two main types of interactions: command-line parameters that affect the behaviour of the program and shall be specified when launching the application, and the interactive commands needed to control the task set and its visualization on the fly.

2.1 Command-line arguments

Most of the command-line arguments deal with controlling the inputs and outputs of the program, ranging from controlling the logging verbosity (`-q`, `-v`) to the size of the graphical window (`-H` and `-W`). Most notably, the `-f FILE` argument can be used to read the task set specification from a file (defaulting to the standard input), with the syntax described in the README file: a task is described by its period, relative deadline, priority and by a list of instructions of the type "run x operations while owning resource Rn".

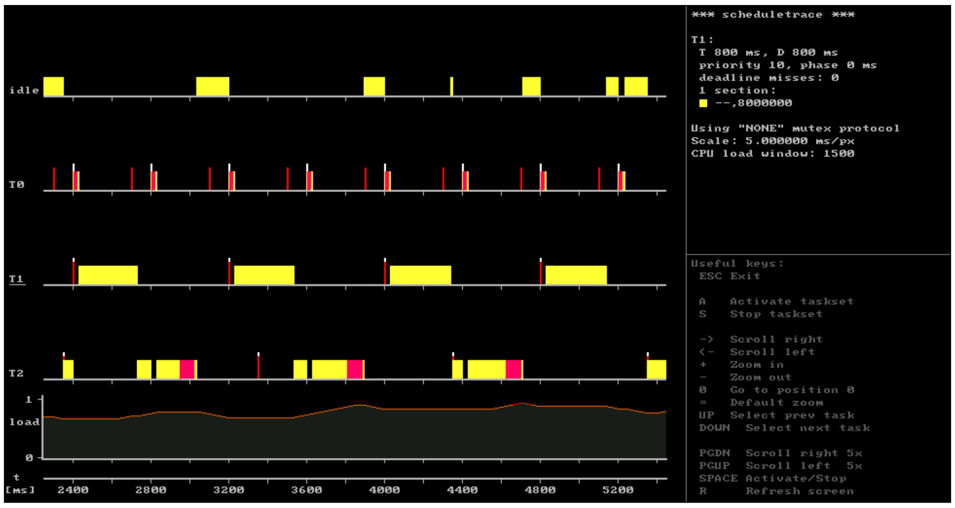
Other interesting command-line parameters are detailed in section 3 when appropriate, while the complete list of valid arguments can be obtained by running the program with the `-h` or `--help` flag.

2.2 Interactive commands

When running with the Graphical user interface, the user is presented with a screen divided into three sections: the biggest one on the left is dedicated for the display of scheduling information, while the rightmost section is further split into two, for displaying contextual information and for providing the list of available commands.

After the GUI has loaded, the user can issue the following commands:

- *A*: Activate the task set.
- *S*: Stop the task set.
- *Space*: Behaves as either *A* or *S*, according to the situation.
- *+/-*: Adjust the scale on the time axis.



```
*** scheduletrace ***

T1:
 T 800 ms, D 800 ms
 priority 10, phase 0 ms
 deadline misses: 0
 1 section:
 ■ --,8000000

Using "NONE" mutex protocol
Scale: 5.000000 ms/px
CPU load window: 1500

Useful keys:
 ESC Exit

 A    Activate taskset
 S    Stop taskset

 ->   Scroll right
 <-   Scroll left
 +    Zoom in
 -    Zoom out
 0    Go to position 0
 =    Default zoom
 UP   Select prev task
 DOWN  Select next task

 PGDN  Scroll right 5x
 PGUP  Scroll left  5x
 SPACE Activate/Stop
 R     Refresh screen
```

Figure 2.1: The graphical user interface during a sample run.

- *P*/*M*: Aliases for +/-, more comfortable but less intuitive.
- =: Restore the default zoom level.
- ←/→: Pan the time axis to the left or right. The shift amount depends on the current zoom level, so as to result in a comparable on-screen distance regardless of the zoom level.
- *PgUp*/*PgDown*: Pan the time axis 5 times faster than the corresponding arrows.
- *0*: Pan the time axis to position 0.
- ↑/↓: Select a previous/next task, respectively.
- *Q*, *Esc* or *Control-C*: Instruct the program to exit gracefully, terminating the task set if still active.
- *R* or *F5*: Forces the GUI to redraw itself. Although this should never be needed in practice, it has been added for debugging purposes, and might turn useful in case of any GUI-related malfunctioning.

Note that it is not possible to restart a stopped task set while keeping the same trace. The task set can simply be restarted by quitting and re-running the application.

3 Architecture

At the program start-up, the taskset file is parsed: for each periodic task a `struct task` is initialized. In the meanwhile, the shared resources (i.e. instances of `struct resource_set`) required by the task set are initialized, and finally one thread per task

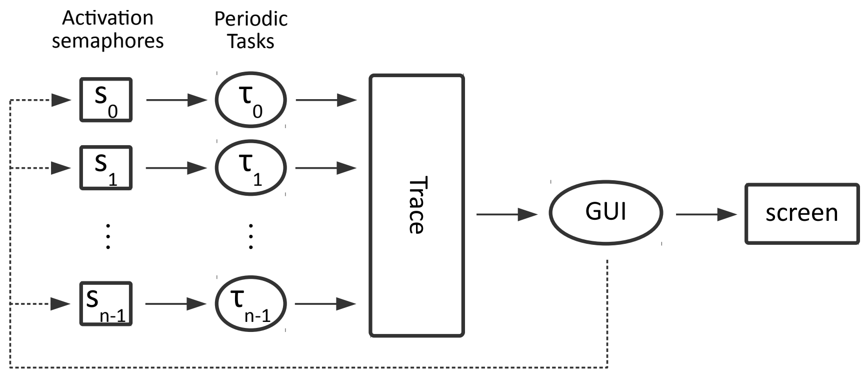


Figure 3.1: A representation of the interactions among the spawned threads. Note that, other than the ones here represented, the periodic tasks themselves may have extra dependencies and interactions, according to the provided description file.

is created. After creation, each thread waits on its activation semaphore until the user instructs the activation, which happens synchronously for all the tasks.

Figure 3.1 represents the spawned threads and their dependencies, excluding the interactions required by the shared resources specified in the given task set file. Also note that while the tasks do synchronize for writing to the shared trace, the reading thread does not require mutual exclusion.

3.1 Tasks and tracing

In order to track the execution of the tasks in user space, tasks share a global tick counter (the `tick` field in `struct taskset`). Thus, each task operation consists of an atomic increment to the tick counter, after having checked whether the private counter is up-to-date with the global one: if this is not the case, a context switch is detected and saved into the global `struct trace` instance.

Because these operations are required to be executed atomically, the tasks synchronize using a binary semaphore (the `task_lock` field in `struct taskset`). This set-up causes the tasks to behave approximately as if they were running on a single-processor machine, where the global tick counter emulates the processor's clock.

3.1.1 Task scheduling and priorities

All threads that are associated to tasks are instructed to be scheduled with the so-called "real-time" scheduling algorithm `SCHED_RR`[1], and they are assigned a fixed priority as specified in the task set definition file. This way, the threads scheduling is left to the kernel, while the program simply observes and tracks its behaviour as described above.

3.1.2 Tracking idle time

In order to also track idle time, an extra simplified “idle” task is run together with every task set. Its body consists of busily acquiring the global lock and incrementing the tick, while running with a priority that is lower than any other thread. This way, even when no other thread is running, the trace is kept up-to-date.

In order to prevent the idle thread to consume an excessive amount of resources, it is possible to force it to sleep briefly at every iteration using the `--idle-sleep` flag. This is especially useful if the user wants to observe the console output while the task set is running: apparently, a busily-running thread prevents the console output to be flushed properly; however, because this appeared to have an impact on the task scheduling, it is turned off by default. The same result was pursued by issuing a call to `pthread_yield()` (which can be enabled with `--idle-yield`), but it turned out to have no visible effect.

Note that, in some cases, the idle thread may be executed for a few iterations even if a higher-priority task is ready: this is probably to be imputed to the semaphore behaviour described in section 3.1.5.

3.1.3 Shared resources

Emulated shared resources are handled with the use of POSIX mutex: each task section can operate on one resource (or on the special “no resource” `R0`). The user can select the protocol to be used for priority-inversion avoidance through the `-p PROTO` or `--protocol=PROTO`, which shall be among `NONE` for no protocol, `INHERIT` for priority inheritance or `PROTECT` for priority ceiling: this is mapped directly on the type of mutex that is initialized, as described in the corresponding POSIX man-page[2].

Note that when using priority ceiling (`PROTECT`), there is no need to manually specify the ceiling, as it will be automatically inferred from the tasks' definitions. When running with the default output verbosity, the program will report to the console the priority ceiling for each resource.

3.1.4 CPU load

Together with the schedule, the CPI load is plotted on the graphical interface. Note that, because the tasks duration is measured in “number of operations”, the program does not have the notion of the remaining computation time for the tasks, so it can't compute the *instantaneous load* as defined by Buttazzo and Stankovic[5].

Instead, the displayed load is computed a posteriori by simply measuring the $\frac{\text{non-idle time}}{\text{time interval}}$ ratio in a given sliding window. For this reason, the load can only be computed after a whole time window has passed since the tasks activation. Note that, given this definition, the measured load is always in the $[0, 1]$ interval: an overload condition is thus indicated by a load of 1 lasting for a sustained period.

3.1.5 CPU Affinity setting

Note that, even if the global `task_lock` semaphore should be enough to guarantee that the tasks behave as if they were on a single-processor machine, this results in an enormous number of context switches, which in turn cause the trace space to be filled quickly and the GUI to have problems in displaying it. My best guess is that this happens because threads are allowed to run on different processors while outside their critical sections, quickly reaching their next call to `sem_wait`, causing the kernel to preempt the running thread after a ridiculous amount of steps (3–20 in some trials) in the effort to guarantee non-starvation of waiting threads. To prevent this, by default, task threads are forced to run on the same processor (chosen at start-up time) using the affinity mechanism[3][4]. To restore the old behaviour and see what it looks like, use the `--no-affinity` flag (when doing this, I advise opening another shell and preparing to issue `killall -9 scheduletrace`).

3.2 Graphical User Interface

The program's GUI is implemented using the Allegro graphics library[1]. All the GUI-related work is performed by a dedicated periodic thread which, with a 60 Hz frame rate, refreshes the plotting area on the screen reading the execution information from the shared `struct trace` instance and according to the current visualization settings (scale and offset) required by the user.

In order to reduce the possible interference of the GUI thread on the tasks scheduling, this operation was chosen not to be semaphore-protected. This is the main reason why the trace is implemented as an ever-growing buffer, rather than a circular buffer or a linked-list: this way, the GUI can safely read all the recorded events up to the current length without the possibility for the tasks to write over past events. Note this also implies that once the trace gets full, new events are discarded.

4 Sample runs

The application comes with some sample task set files in the `tasksets` folder. Note that, because the task sections lengths are specified with their operations count, it is possible that on other machines (or changing the external workload) their behaviour differ from the intended one. For the same reason, it is advisable to set a static CPU frequency during the tests, so as to increase uniformity across consecutive runs.

Figure 4.1 shows an example of running `scheduletrace` with `-f tasksets/inversion` and `-p none` (so that no priority-inversion avoidance protocol will be in use), while figures 4.2 and 4.3 show a similar situation using, respectively, `-p protect` and `-p inherit`, highlighting the different behaviours of the resource sharing protocols. The `inversion` task set is designed to cause priority-inversion, with the high-rate task `T0` sharing a resource with the low-rate `T2`: while the length of the critical section of `T2` is

[1]Allegro – A game programming library: `http://alleg.sourceforge.net` .

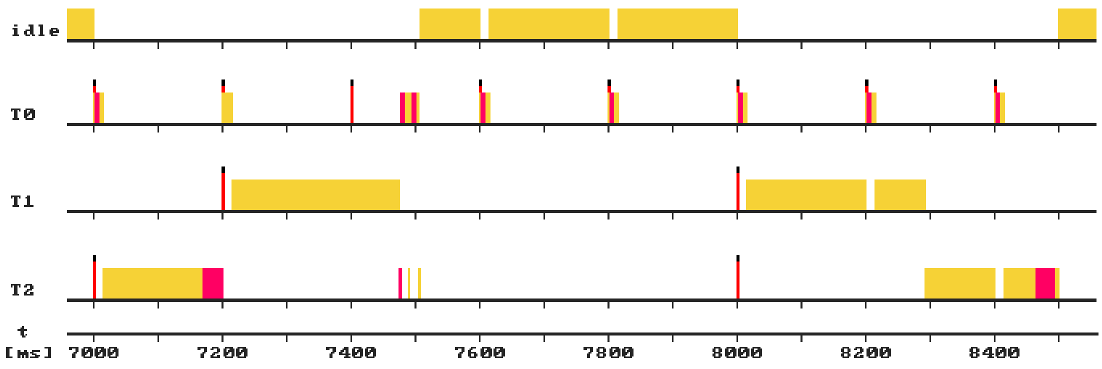


Figure 4.1: The scheduling trace during a priority-inversion phenomenon: at 7200 ms, task `T2` is running a critical section and is preempted by `T0`. However, when `T0` tries to enter its own C.S. it can't, and it must wait until `T1` has completed for `T3` to end its own C.S. and finally be allowed to run, after missing its deadline at 7400 ms. Note that colors have been modified for printing friendliness.

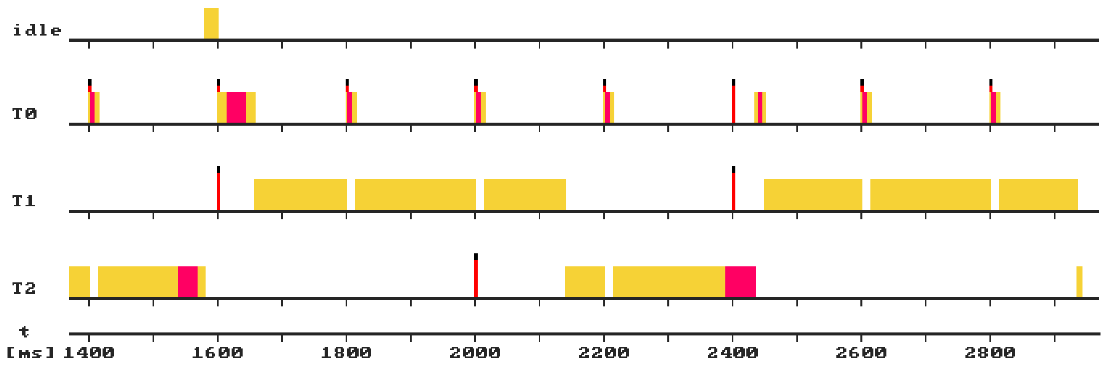


Figure 4.2: The scheduling trace in a situation similar to the one in figure 4.1, but with the priority ceiling protocol enabled: at 2400 ms, `T0` and `T1` are activated while `T2` is in its C.S, but this time, because `T2` is owning the resource, it keeps executing until the end of its C.S., then `T0` and `T1` can run without blocking and finally `T2` completes its execution.

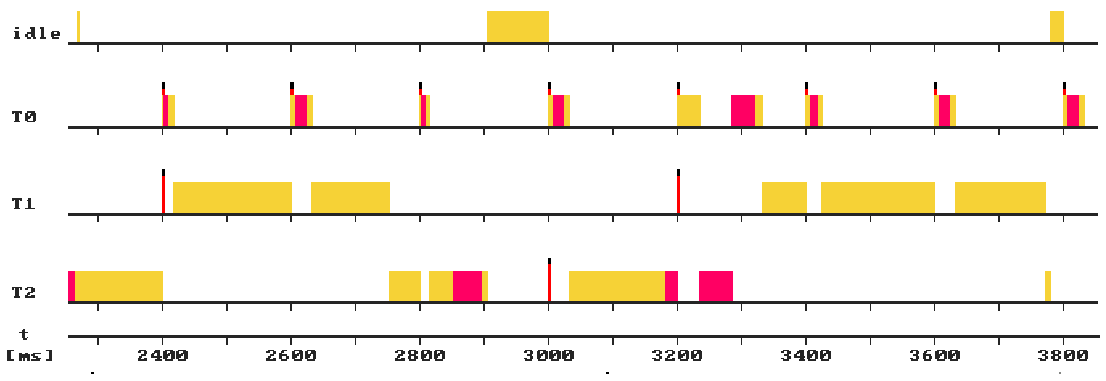


Figure 4.3: The scheduling trace in a situation similar to the one in figure 4.1, but with the priority inheritance protocol enabled: at 3200 ms, `T0` and `T1` are activated while `T2` is in its C.S, but when `T0` requests to enter the C.S. `T2` inherits its priority, allowing it to release the resource and let `T1` complete on time.

comparable to the run-time of `T0`, in case of priority-inversion the mid-rate task `T1` can get to execute before `T0`, causing it to miss its deadline.

5 Conclusions and limitations

`scheduletrace` is a useful tool for graphically displaying what is going on with the scheduling of periodic tasks, which may find applications for didactic purposes or, in the most rosy case, to ease the development of scheduling algorithms.

It has, however a number of limitations and some missing features that may ease its usage. As discussed in section 4, it is quite a pain to produce reproducible behaviours when defining a task set; while this is intrinsic into the scheduling problem in general, sharing with other a critical run could be eased by allowing to load and browse the trace produced by a previous run. Other interesting extensions could be to provide the capability of asynchronously activate the tasks, or to include different types of aperiodic servers and the possibility to specify and activate aperiodic jobs.

As discussed in section 3.1.4, the tool currently does not support estimating computation times. In principle, it could be achievable to estimate at run-time the computing times and thus be able to compute and plot the instantaneous load deadlines and remaining computation time into account, while adaptively correcting the speed estimate for the CPU as perceived by the running program. However, this would require adding non-negligible complexity to the program. A simpler approach would be to report the average computation speed to the user and allow them to run the next instance using that estimate.

Further smaller improvements that could increase the usefulness and usability of this tool are detailed in the README file.

Further documentation

Further bits of information about `scheduletrace` can be found bundled with its source code:

- For the building instructions, refer to `README.md` file in the project tree root. Spoiler: put yourself in the project root and run `scons` or, if missing, `make`.
- For a detailed list of command line options, compile and run with the `-h` or `--help` flag.
- The taskset file format is also described in the `README.md` file.
- A list of interactive commands is available directly on-screen, when `--no-gui` is *not* specified.

References

[1] VV.AA. *SCHED(7).* Linux Programmer's Manual. Linux man-pages. 2014.

[2] VV.AA. *PTHREAD_MUTEXATTR_GETPROTOCOL(3P).* POSIX Programmer's Manual, IEEE/The Open Group. 2013.

[3] VV.AA. *PTHREAD_SETAFFINITY_NP(3).* Linux Programmer's Manual. Linux man-pages. 2014.

[4] VV.AA. *SCHED_SETAFFINITY(2).* Linux Programmer's Manual. Linux man-pages. 2014.

[5] G. C. Buttazzo, J. Stankovic. *Adding robustness in dynamic preemptive scheduling.* In D. S. Fussel and M. Malek, editors, Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems. Kluwer Academic Publishers. 1995.