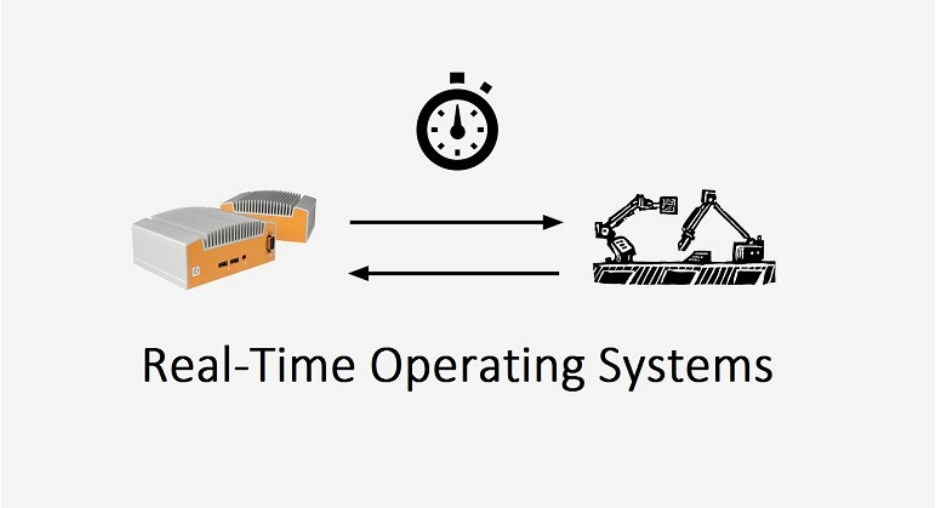

Real-time Operating Systems

Tutorial 1: RT Services on Linux
2022/2023

Paulo Pedreiras
DETI/UA/IT
pbrp@ua.pt

September 27, 2022

1 Objectives

- Creating periodic processes and observe its temporal behavior on Linux
- Utilization of real-time services of Linux

2 Procedure

- Download the sample code provided at the course webpage ("LinuxRT-Services.tar.gz")
- Carry out the steps indicated at Sections 2.1 and 2.2.
- Where appropriate, take note on your notebook of the code modifications and results observed, in order to evaluate the impact of the RT services.

2.1 Observation of the behavior of a periodic task in Linux

- Analyze the provided source code.
  Pay special attention to the technique used to generate periodic activations and to the structure of a "task".
- Tune the "Heavy_Load" function to take an execution time of around $20\,ms$.
- Launch the "task" and then other processes, concurrently, in different terminals, and observe the behavior.
  - Use processes that generate intensive CPU utilization and I/O operations (e.g. backup a big folder with tar, watch Youtube videos).
  - Repeat the operation several times and annotate the results.
  - Note that the impact depends on your hardware and OS configuration. Launch at least as many load processes as CPU cores present on the computer used for executing the tests.

2.2 Applying the RT services of Linux

- [**A1**] Modify the program in such a way that the periodic thread receives a fixed real-time priority, hard-coded in the source code. Repeat now the experiments above, making sure that it is used a similar interfering load (i.e., the same application set is used).

- [**A2**] Modify the code developed in **A1** to allow passing two command line arguments: i) priority (integer, [1,99]) and, ii) periodicity (integer, ms resolution, range [50,500] ms).
  Execute several instances of the application, in different terminals, with different priorities and periodicity. Periods should not be harmonic and should generate a different phase conditions at runtime (e.g. use {90 ms, 95 ms, 107 ms, ...}
  Analyze the results.

- [**A3**] Modify the code developed in **A2** so that tasks are all assigned to CPU 0.
  Execute several instances of the application, in different terminals, with different priorities and periodicity. As in [**A2**], use periods that generate different phase conditions at runtime.
  Analyze the results. usar periodos nao harmonicos

3 Deliverables

The following elements should be uploaded for evaluation:

- A compressed file with a Makefile and the source code corresponding to Assignments A2 and A3

- A **one page** report, in "pdf" format, with your analysis of:
  - The improvements observed in A1 with respect to the execution of the task using the standard Linux scheduling services
  - The impact of priorities in Assignment A3
  - Note: your analysis should be supported by numerical values.

4 Notes

4.1 Note 1

The execution of programs with real-time priorities requires superuser privileges. The command "sudo" allows a user to execute certain commands with this kind of privileges.

4.2 Note 2

To allow the observation and interpretation of the interference pattern between tasks in multi-core CPUs, it necessary to force all processes to execute on the same CPU (aim of Assignment A3). That can be done with the following code, which should be placed at the beginning of the program.

```
...
#define _GNU_SOURCE /* Required by sched_setaffinity */
...
/* Variables*/
cpu_set_t cpuset;
...
/* Forces the process to execute only on CPU0 */
CPU_ZERO(&cpuset);
CPU_SET(0,&cpuset);
if(sched_setaffinity(0, sizeof(cpuset), &cpuset)) {
    printf("\n Lock of process to CPU0 failed!!!");
    return(1);
}
...
```

4.3 Note 3

To verify if processes are indeed executing with real-time priorities, there are two possible approaches:

4.3.1 Programmatically

Use the following functions within the thread:

gettid() Returns the integral thread ID that can e.g. be used with **"chrt -p TID"**

sched_getscheduler(0) Returns the scheduling policy used by the thread

sched_getparam(0,&parm) Returns the thread RT priority

The equivalent can be made on the main thread. The only difference is that function **"getpid()"** should be used instead of **"gettid()"**, for obvious reasons.

4.3.2 Via a console

The easiest way of identifying the real-time processes is using "htop", which, by default lists both RT and "normal" processes. Note that this tool shows RT processes as having a negative value that is in absolute value 1 unit higher than the actual real-time priority. E.g. a RT process with priority 30 shows up as "-31".

The more standard "top" command can also be used, but by default it does not list real-time processes. To make them visible it is required the use of the "-H" argument.

Once the PID/TID is identified, the full process/thread info can be obtained again via **"chrt -p TID"**.

4.3.3 A remark about priorities

The issue of process/thread priorities is confusing, because there is (or should be) a separation between how priorities show up at user space and how they are handled internally by the kernel. To complicate things further, the RT services are an addition that had significant implications on the kernel, resulting is some inconsistencies in how a few attributes are handled in different parts of the OS.

Interested students can find an interesting discussion on this topic in the following Stack overflow forum discussion ("Which real-time priority is the highest priority in Linux"):

https://stackoverflow.com/questions/8887531/which-real-time-priority-is-the-highest-priority-in-linux