INFO-F-410 Embedded Systems Design
Linux and Real-time

Gilles Geeraerts

Academic Year 2012–2013

1 Introduction

In this practical, we will briefly review two ways to obtain some kind of real-time support in Linux. We will first see that the 'classical' (so-called *vanilla*) kernel of Linux supports real-time to some extent. Then, we will consider a more robust solution, the *Real Time Application Interface*, or RTAI for short, which is a patch to the Linux kernel that extends Linux to support hard real time features, such as precise timers and periodic tasks.

To enable real-time features of the vanilla kernel, and to run RTAI, we need root access to the machine. Therefore we will work using a virtual machine. In particular, RTAI uses so-called Linux Kernel Modules (or LKM for short) that can be inserted and discarded at run-time. We'll first start with the vanilla kernel rela-time features. Then, we'll review the basics of LKM, see what services are offered by RTAI and how to use them.

2 Virtual Machine

As RTAI requires the kernel to be patched, and loading and unloading LKMs requires root privileges, we will use a VirtualBox virtual machine to experiment during the practical. The VM we will use is a Debian, that boasts several kernels, among which a 2.6.32.20 patched against RTAI 3.9

Using the VM directly is not convenient, so we will mount its virtual file system as an SMB share.

Task 1 *Let's set up the VM…*

1. *Start Virtual Box and check that a virtual network called* `vboxnet0` *exists. Go to Fichier → Paramètres → Résau and add a virtual network called* `vboxnet0` *if necessary.*
2. *Then open the virtual machine and launch it. The archive for the virtual machine can be found in* `/serveur/logiciels/tp-infof410`*. You should be able to uncompress the machine in* `/var/tmp` *as you need about 3.5 Gb.*
3. *Obtain the IP address of the VM by typing* `ifconfig eth1` *at command prompt.*
4. *Mount the VM's hard drive. In the File Manager, go to Fichier → Connexion au serveur. Choose Partage Windows. Give the IP under Serveur, type* `//homes` *under Partage and* `root` *under Identifiant and click OK. Then, give* `INFOF410` *as Domaine and* `infof410` *as Mot de passe, then click Se connecter.*

3 Real-time threads under Linux

Since version 2.6, it is possible, with the *standard Linux kernel* to use *real-time tasks*. The main issue with standard (i.e., non real-time tasks) is that the standard Linux scheduler is not predictable enough to allow real-time. For instance, the man page of the `uspleep` system call reads:

> The `usleep(usec)` function suspends execution of the calling thread for (**at least**) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.

This is, of course, not acceptable in a real-time critical system.

To avoid this, real-time tasks in Linux are equipped with a *priority*, which is a number ranging from 0 to 99. Tasks with priority 0 use the 'classical' time sharing scheduler of Linux. Tasks with priority ≥ 1 are actual real time tasks.

The scheduler uses these priorities to deal with preemption:

A task of priority j cannot run as long as a task of priority $k > j$ is active.

Remark that several tasks can exist at the same priority level. To deal with this, we can choose between two schedulers:

- `SCHED_FIFO`: no preemption at the same level of priority.
- `SCHED_RR`: *round robin scheduler*. Inside each priority level, tasks are interrupted on a regular basis to let the other tasks of the same priority level execute.

Exercise 1: A real-time process

To turn a task into a *real-time task*, we use the `sched_setscheduler()` system call, whose parameters are:

- The PID of the process, 0 for the current process.
- `SCHED_FIFO` or `SCHED_RR` to select the scheduler.
- A `const struct sched_param *`, where the `sched_param` structure contains only one field `sched_priority`, which is the priority.

Task 2 *Based on the* `/root/RTLInux/TP/canevas.c` *template, write a simple process that runs at priority* 99*, and calls* `alarm(15)`*, then enters an active loop* `while(1) ;`*. The* `alarm` *system call will trigger a signal after 15 seconds and kill the process. Observe how the machine reacts (or not) during the execution of the real-time task.*

Quite surprisingly, the machine still reacts (although very slowly) during the execution of the real-time task. Yet, the shell executes at level 0,. . . The reason is that Linux still keeps about 5% of the CPU time for tasks of lower priority, to avoid completely freezing the machine. Linux is not really a real-time operating system, we will see later that there are better solutions for hard real-time.

Exercise 2: Real-time threads

Let us now create real-time threads inside a real-time process. To create real-time threads, we will use the following schema:

```
pthread_t thr ; // The pthread structure that represents the thread
pthread_attr_t attributes ; // The thread attributes

struct sched_param parameters ; // See above

// The function that the thread will execute
void * function(void * arg) {
  /* ... */
}

// Initialises the pthread_attr_t structure with default values
pthread_attr_init(&attributes) ;

// Choose the priority, as before
parameters.sched_priority = 50 ;

// Set the scheduling policy for the thread
pthread_attr_setschedpolicy(&attributes, SCHED_FIFO) ;

// Set the priority for the thread
pthread_attr_setschedparam(&attributes, &parameters) ;

// Activate the value set in attributes.
// Otherwise, the value is ignored and the thread inheritates
// the value of its parent !
pthread_attr_setinheritsched(&attributes, PTHREAD_EXPLICIT_SCHED) ;

// Create the thread
// the last parameter is a void * pointer that will
// be mapped to the function parameter.
pthread_create(&thr, &attributes, function, NULL) ;
```

Task 3 *Using the template in* `/root/RTLinux/TP/canevas-thread.c`*, write a process that creates* `NB_THREADS` *real-time threads whose priority is proportional to their ID (i.e, thread 4 is more prioritary than thread 3, and so on), and that execute the* `fonction_thread` *function given in the template. Make sure that the threads are launched by increasing order of id (first thread 1, then thread 2,...), and check that they finish in reverse order (thread* `NB_THREADS` *is the most prioritary, and so on). Remark that, when creating several threads, the* `sched_param` *and* `pthread_attr_t` *can be reused.*

Task 4 *What happens if you execute the same thread without real-time support ? Observe the interleaving.*

4 RTAI: *real* real-time under Linux

Exercise 1: let's write an LKM !

We will write a first LKM. Each LKM is composed of at least one C file containing:

```
#include <linux/kernel.h>
#include <linux/module.h>

int init_module()
{
 // ...
  return 0;
}

void cleanup_module()
{
 //...
}
```

The `init_module()` and `cleanup_module()` functions are called respectively when the module is loaded or unloaded.

To compile the module, one relies on the build system of the Linux kernel. To do so, we must use a special Makefile:

```
XTRA_CFLAGS += -I/usr/realtime/include -D__IN_RTAI__

obj-m += module.o

all:
	cp /usr/src/rtai-3.9/Module.symvers .
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

where `module.o` must be replaced by the name of the source file of the module (if the source file is `myfile.c`, we should have `myfile.o` in the `Makefile`). Do not change the `modules` at the end of the `all` rule !

Then, we can compile our module by typing `make` as usual. The `Makefile` instructs `make` to use the Makefile of the kernel (located, in our case, in `/usr/src/linux-rtai`), in order to compile the module. After the compile, we get a `.ko` file that can be loaded with the `insmod` command, and unloaded with `rmmod`. To observe the messages produced by the module, use the `dmesg` command, which lists the last kernel messages.

Task 5 *Write an LKM that says 'Cheeeeese Gromit' when loaded and 'May the Force be with you' when unloaded (or any other line from your favorite movies). Compile it, load it and unload it. Observe the messages using* `dmesg`. *To print messages, use the* `printk` *function.*

Exercise 2: A first real-time task

Throughout the rest of the practical, you will need to refer to the API guide of RTAI, which is available on-line at: `https://www.rtai.org/documentation/magma/html/api/` or `http://gatling.ikk.sztaki.hu/~kissg/doc_rtai/manual.html`.

We will now write a first task than can profit of the RTAI features. Our first task will be a *one shot* task, which means that it will not be periodic.

For that purpose we need to:

1. Include the `rtai.h` and `rtai_sched.h` files.

2. Write a function (with one argument of type `long`) that contains the task to be run.

3. In the initialisation of the module:

   (a) Set the scheduler to *one shot*, by calling `rt_set_oneshot_mode()`.

   (b) Start the timer with `start_rt_timer()`. This initialises the programmable timer of the machine [4]. It will be used to ensure the respect of the real-time constraints.

   (c) Initialise a `struct` of type `RT_TASK` with `rt_task_init` (see the documentation for the details). This effectively creates the task.

   (d) Start the task with `rt_task_resume()`.

Task 6 *Write an RTAI task that just says 'Hello'. Compile it and test it. This requires that the RTAI LKMs are loaded as well. You can load the modules from* `/usr/realtime/modules`*:*

```
insmod /usr/realtime/modules/rtai_hal.ko
```

then

```
insmod /usr/realtime/modules/rtai_sched.ko
```

Exercise 3: Using arguments

We can use the argument of type `long` that has to be passed to the function implementing the task to pass information to the task, or to provide them with an identity.

Task 7 *Modify the code of the previous example to have two tasks that run the same function, which prints 'Hello I am task n', where $n \in \{1, 2\}$ is the identity of the task. This identity is given to the task when calling* `rt_task_init`*. Compile it and run it.*

Exercise 4: using sleeps

With RTAI the behaviour of the `sleep` is predictable. The sleeps can be achieved with `rt_busy_sleep()`, with a parameter in nanoseconds, or with `rt_sleep()` with a parameter in ticks of the timer. Timings in nanoseconds can be converted to ticks of the timer using the `nano2count()` function.

Task 8 *Write a one shot real time task that executes 1000 times the following actions:*

1. *Get the current time using* `rt_get_time_ns()`

2. *Compare this value to the previous current time, and print the difference. The printed value is thus the time elapsed since the last sleep.*

3. *sleep for 100 msec using* `rt_sleep`.

Check that the task is indeed awaken periodically, and that the period is very stable.

Exercises 5 and 6: Periodic tasks

To obtain a task that gets scheduled periodically, we must:

1. Write the function for the task so that it calls `rt_task_wait_period()` each time it has finished one of its job. The general canvas of such is task is thus:

```
void my_task(long arg) {
  while(1) { // for a task that run forever
    // Do some job
    rt_task_wait_period() ;
  }
}
```

2. Compute the period in terms of ticks of the timer. For that purpose, we use the `nano2count` function that converts a value in nanoseconds to a value in ticks.

3. Start the timer using an appropriate tick length, using `start_rt_timer`.

4. Set the scheduler to periodic, using `rt_set_periodic_mode()`.

5. Initialise the task as in the one shot mode.

6. Make the task periodic using `rt_task_make_periodic`. We can use the `rt_get_time()` function to get the current time (this is necessary for `rt_task_make_periodic`, that requests an activation time for the task).

7. Start the task using `rt_task_resume`

Task 9 *Write a periodic task with period of 70 microseconds. Each job of the task prints the current time (obtained using* `rt_get_time_ns()`*). Limit the task to 100 jobs. Compile it and run it.*

Task 10 *Add a second task that does the same as the former, but with a period of 30 microseconds. What should be the sequence of activation moments ? Let the jobs print their task identifier, observe the result with* `dmesg`*, and compare it to the sequence you had predicted.*

References

[1] Brian Anderson, *Linux Loadable Kernel Module HOW TO*. Accessed on March, 5th, 2013. `http://tldp.org/HOWTO/Module-HOWTO/`

[2] *RTAI API documentation*. Accessed on March, 5th, 2013. `https://www.rtai.org/documentation/magma/html/api/`

[3] *RTAI*. Web page of the Robot LAB at Radboud University Nijmegen. Accessed on March, 5th, 2013. http://www.cs.ru.nl/lab/rtai/.

[4] *82C54 CHMOS Programmable Interval Timer*. Intel. http://download.intel.com/design/archives/periphrl/docs/23124406.pdf.