Exercise 6: BRTT and Xenomai

In this exercise we will do the same things as in the previous exercise, just that we will be using Xenomai which improves upon the real-time properties of Linux. When using Xenomai, a thread/task can run with a higher priority than the rest of the Linux system, including Linux itself.

To use Xenomai we will boot into a Linux Mint distro. This has, for some reason, not been installed on all the computers at the lab. So, if you cannot find this distro you need to change lab seats. To boot Mint do the following: reboot the computer, when the GRUB menu is shown (the menu where you can choose between Linux and Windows) you should see Linux Mint 13 Maya as an option, just below you have the option **Advanced options for Linux Mint 13 Maya**, select this using the arrow keys and enter, then select Linux Mint 13 Maya, kernel 3.5.7-xenomai (there should only be one). The username and password combination is the same as for Ubuntu.

The following links are to some Xenomai documentation that you will need to use to complete the exercise. The first link is to the Xenomai API reference, where the relevant functions for this exercise are in the "Task management services" section. The second cs.ru.nl link is a tutorial with some more explanation and code examples.

https://www.xenomai.org/documentation/xenomai-2.6/html/api/group__native.html

http://www.cs.ru.nl/lab/xenomai/exercises/

1. Xenomai

Xenomai introduces two classes of scheduling, primary and secondary mode. In primary mode a task is running with a higher priority than any Linux activity. Linux and services provided by it are ran in secondary mode. A task will switch between these two modes dependent on what calls it is making. Xenomai calls puts the task in primary mode, and non-xenomai calls puts the task in secondary mode. Primary mode tasks are always scheduled in front of secondary tasks.

If your computer becomes sluggish after starting a RT application, you can kill the program with ctrl+c. If the computer freezes completely, you have probably started an infinite loop with higher than Linux priority. Ctrl-c will not work; you have to power off and on the computer. Needless to say, you should avoid this.

1.1. Necessary compiler flags

In order to compile a Xenomai application, you will need to include a few extra compiler and linker arguments. The program `xeno-config` returns the relevant compiler and linker flags when invoked with `--skin native --cflags` or `--skin native --ldflags` respectively.

A simple makefile is available on Blackboard, this includes all the necessary configuration and will compile all .c-files in the current directory.

1.2. Memory and Xenomai

To avoid a real time task to be swapped to disk, Xenomai provides the `mlockall()` function. Calling this, with the argument `MCL_CURRENT|MCL_FUTURE`, will lock the current memory allocations and future memory allocations to the main memory. This is important because when executing under hard real time constraints we want to avoid all unpredictable time delays. This call should be the first thing your program does.

1.3. Task creation summary

The three main functions you will need in order to make a new task are

- **`rt_task_create`**`(RT_TASK *task, const char *name, int stksize, int prio, int mode)`
  task: Address to an RT_TASK variable, that you must create elsewhere
  name: Some descriptive name for this task
  stksize: Stack size for this task. 0 sets a default size.
  prio: Base priority. 0-99, where 0 is the lowest.
  mode: Task creation mode. Flags can be OR'd together. The flag you will need is:
  `T_CPU(cupid)`: sets the CPU affinity for this task

- **`rt_task_start`**`(RT_TASK *task, void(*entry)(void *cookie), void *cookie)`
  task: Address of the task, which has previously been passed to `rt_task_create`
  entry: The function that is to be run in this task
  cookie: The arguments passed to the entry point function, or NULL if none

- **`rt_task_set_periodic`**`(RT_TASK *task, RTIME idate, RTIME period)`
- **`rt_task_wait_period`**`(unsigned long *overruns_r)`

  Look up the API reference to see how these two functions work. Use `TM_NOW` as the start time, and note that the period is an integer specifying nanoseconds.

To make sure you have gotten everything right so far, you should create a periodic task that just prints to the screen.

Note that calling `printf()` *will force your task to enter secondary mode, since this is a Linux syscall. This is not worth fixing right now, but for later in this exercise and in the next exercise (Xenomai Semaphores), you will want to avoid calling "normal" printf. Instead, use* `rt_printf()` *found in* `rtdk.h`*, and call* `rt_print_auto_init(1)` *at the start of your program.*

2. BRTT reaction test

Last week we performed a reaction test using periodic POSIX threads. In this exercise we will do the same test with Xenomai tasks instead of normal pthreads. To get a fair comparison, we must test with the same conditions as last time, replicating CPU affinity (single core vs. all four), the periodic interval, and any disturbance threads.

The disturbance threads should be created as normal pthreads, just as in the previous exercise. Creating the disturbance threads as Xenomai tasks will likely lock up your system, so don't do that. Set the CPU affinity for the disturbance threads the same way you did last week, and use `T_CPU` to set the affinity of the Xenomai tasks.

The delay for holding the response pin low should be a call to a Xenomai sleep or `rt_timer_spin(ns)`, not a Linux sleep, as this will put the task into secondary mode. You may also find that you do not need this delay at all.

When creating busy-polling Xenomai tasks, be very careful to not populate all CPU cores with these non-interruptible tasks, as this will lock your system up completely. As long as the tasks regularly yield, and you set a timeout for the maximum duration the task can run, it should be fine. Here is some code to show you how to do this:

```
unsigned long duration = 10000000000;  // 10 second timeout
unsigned long endTime = rt_timer_read() + duration;

while(1){

    // do work

    if(rt_timer_read() > endTime){
        rt_printf("Time expired\n");
        rt_task_delete(NULL);
    }
    if(rt_task_yield()){
        rt_printf("Task failed to yield\n");
        rt_task_delete(NULL);
    }
}
```

Task A:

Create three busy-waiting Xenomai response tasks, and record the response times both with and without 10 pthread disturbance threads. Make sure all response tasks and disturbances are running on the same CPU core.

Compare the results to last week (where the response tasks were also pthreads). How does the introduction of the disturbance threads affect the response time of the Xenomai tasks?

Task B:

Create three periodic Xenomai response tasks, with a period of 1ms (or whatever is the same as what you used last week).

Recall from last week - did any periodic pthread response thread have a predictable worst-case? And now with Xenomai tasks, has the worst-case become predictable?