Exercise 7: Xenomai Semaphores

In this exercise we will continue to work with Xenomai, and will be concentrating on semaphores and mutexes. We will look at problems like priority inversion and deadlock.

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

http://www.xenomai.org/documentation/xenomai-2.6/html/api/index.html

Use the same makefile as for the Xenomai response testing.

1. Synchronization

The first task is to perform a barrier synchronization. Xenomai has only counting semaphores. The semaphore header file is *native/sem.h.*

After creating a RT_SEM variable the semaphore can be initialized with the following function:

```
Int rt_sem_create(RT_SEM *sem, const char *name, unsigned long icount,
int mode)
```

It is important to run the `rt_sem_delete()` function at the end of the program.

The `rt_sem_broadcast()` function can be used to send a signal to all tasks that are waiting for a semaphore. These tasks will then start at the same time, i.e. they are synchronized. The functions will start in the order they arrived in or by priority level, depending on what mode the semaphore is created with.

Other functions you will need for using semaphores can be found in documentation.

Assignment A:

Create two tasks, with different priorities that will wait for a semaphore after they have been initialized.

Create a third task with highest priority, and use it to synchronize the two first tasks. The third task can be the main function itself, or you can create it in a separate task. To make main a rt task use the following function:

```
Int rt_task_shadow (RT_TASK *task, const char *name, int prio, int mode)
```

task can be sett to *NULL*. The other arguments are the same as a rt_task_create call.

Use the following pseudocode for this assignment:

```
task_init();
sleep(100ms);
broadcast_semaphore();
sleep(100ms);
delete_semaphore();
exit_program();
```

Use `rt_printf()` to show the results.

Just as last week, all threads should run on the same CPU, so `rt_task_create()` must specify this. This must be done in all the exercises.

2. Solving Priority Inversion with Priority Inheritance

Priority inversion is a classical problem with three tasks with low, medium and high priority, called L, M and H, is running. L takes a semaphore, and then gets preempted by M which runs for a long time. When the H also wants to take the semaphore, it must wait until M is completed so L can return the semaphore. Priority inversion can cause H, the most important task to miss its deadline.

If the system uses priority inheritance, then L will inherit the priority of H, when H waits for the semaphore L have taken. This means that H no longer must wait for M.

Assignment B:

You must create 3 tasks, with low, medium and high priority. These should all start at the same time, so you should use your sync code from the previous assignment. The tasks should do as follows:

L: start at time 0, lock the resource, busy-wait for 3 time units, unlock the resource.

M: start at time 0, sleep for 1 time unit, busy-wait for 5 time units.

H: start at time 0, sleep for 2 time units, lock the resource, run for 2 time units, unlock the resource.

You can use the function shown below to print a message along with the tasks base/current priority. `rt_timer_spin()` is a busy wait delay function provided by Xenomai. `rt_task_sleep()` is the no-work sleep function. Both takes arguments in ns. There is a issue with `rt_timer_spin()` which is explained in the appendix. Below is a working busy wait function and a wrapper for the `rt_task_sleep()`. Both functions use ms as the resolution.

Test this by first using a semaphore as the resource lock, then by using a mutex. Semaphores does not implement priority inheritance, this is only implemented with mutexes. The mutex header file is *native/mutex.h*. Which of those provide priority inheritance? Draw how you expect them to behave prior to implementing and draw the results after completing the assignment.

```
/*
The base and current priority of the running task is printed together
with a provided message.
*/
void print_pri(RT_TASK *task, char *s)
{
      struct rt_task_info temp;
      rt_task_inquire(task, &temp);
      rt_printf("b:%i c:%i ", temp.bprio, temp.cprio);
      rt_printf(s);
}

int rt_task_sleep_ms(unsigned long delay){
    return rt_task_sleep(1000*1000*delay);
}

void busy_wait_ms(unsigned long delay){
```

```
    unsigned long count = 0;
    while (count <= delay*10){
        rt_timer_spin(1000*100);
        count++;
   }
}
```

3. Solving Deadlock with Priority Ceiling

You have previously been fixing a deadlock situation in Linux. An effective method for avoiding deadlocks is priority ceiling. We will be using the ICPP solution, which assigns a priority to the resources protected by mutexes. This priority should be higher than the priority of any of the tasks. When a task locks a mutex, it will have its priority temporarily increased to that of the resource.

Assignment C:

Create two tasks of different priorities, L and H.

L: start at time 0, take mutex A, busy-wait for 3 time units, take mutex B, busy-wait for 3 time units, return mutex B, return mutex A, busy-wait for 1 time units.

H: start at time 0, sleep for 1 time unit, take mutex B, busy-wait for 1 time units, take mutex A, busy-wait for 2 time units, return mutex A, return mutex B, busy-wait for 1 time units.

This should result in a deadlock, when both threads wait for the mutex that the other holds. You should not move on before you have a deadlock. Draw how you expect the tasks to behave and draw the results of the program.

Assignment D:

Solve the deadlock using ICPP. You can change the priority of a task with `rt_task_set_priority()`. Be sure to set the priority back to the original when both mutexes are returned, the last busy-wait time unit should be executed with the tasks original priorities, and check that the last busy-wait periods of the two tasks are performed as they should and with the correct priorities. Draw how you expect the tasks to behave and draw the results of the program.

> *You might need to do a short sleep (`rt_task_sleep(1)`) after lowering the priority. Otherwise it is possible that the change of priority does not take effect.*

4. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

1 Show the priority inversion problem with and without priority inheritance, and show your graphical representation of what is happening.

- What happened in these two cases?
- Did you see priority inheritance?

2 Show the solved deadlock using ICPP, and show your graphical representation of what is happening with and without ICPP.

- Explain what happens.

5. Appendix

The `rt_timer_spin()` function relate to the realtime clock when burning CPU cycles. It is implemented as a busy-wait-until. While this is a good way to achieve accuracy, it fails when a function is preemted in a spin.