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.

1. Synchronization

Xenomai's counting semaphores are found in `native/sem.h`. Follow this link to the API reference:

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

It is important that you call rt_sem_delete() at the end of the lifetime of the semaphore, as semaphores are allocated on a dedicated memory region for semaphores that can run out of space (though it will take a very long time before that happens).

The decrement and increment functions are called `rt_sem_p()` and `rt_sem_v()`, respectively.

The function `rt_sem_broadcast()`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.

Task A:

Create two tasks, with different priorities that will wait for a semaphore broadcast after they have been initialized. All tasks should run on the same CPU core.

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.

Here's some pseudocode to show what we want:

```
start two tasks
sleep 100ms
broadcast semaphore
sleep 100ms
delete semaphore
exit program
```

Use `rt_printf()` instead of normal Linux `printf()`. Include `rtdk.h`, and call `rt_print_auto_init(1)` at the start of the program.

To make `main` a Xenomai task, use the function `rt_task_shadow()`, with the `task` parameter set to `NULL`. The remaining arguments are the same as a call to `rt_task_create`.

2. Priority inversion and inheritance

When a high priority task H is sharing resources with a low priority task L, we may encounter scenarios where H has to wait for L to release the resource. This is known as *priority inversion*. If we also have a medium priority task M, this task will prevent L from running and releasing its resource, causing H to wait further. This is known as *unbounded priority inversion*, and is typically undesirable for real-time systems, as it makes priorities somewhat meaningless.

In order to fix this, we use *priority inheritance*, where L will inherit H's priority level as soon as H requests the resource. This pushes L ahead of M, letting H get its resource as soon as possible, without unbounded interruption from a medium priority task.

Task B:

Create a program that exhibits unbounded priority inversion.

Create three tasks, with three different priority levels. These should all start at the same time, so use the barrier synchronization code from the previous task. The tasks should do as follows:

- **Low**: Lock the resource. Busy-wait for 3 time units. Unlock the resource
- **Medium**: Sleep for 1 time unit. Busy-wait for 5 time units.
- **High**: Sleep for 2 time units. Lock the resource. Busy-wait for 2 time units. Unlock.

Use `rt_task_sleep()` to sleep. It takes an integer number of nanoseconds as its only argument. Use `rt_printf()` to print out when a task starts and stops its busy-wait "work".

The available busy-wait function `rt_timer_spin()` continuously reads the CPU clock, which will keep ticking even if some other task is being run. Because of this limitation, we call `rt_timer_spin()` with short enough delays that we are unlikely to be rescheduled for its duration, then repeat that in an outer loop in order to get longer delays:

```
void busy_wait_us(unsigned long delay){
    for(; delay > 0; delay--){
        rt_timer_spin(1000);
    }
}
```

Create a drawing (or ascii-art?) of how you expect this program to run. Then run the program, and verify that you get symptoms of priority inversion.

Task C:

Unlike semaphores, mutexes have priority inheritance built in. Modify the code from the previous task to use a mutex instead of a semaphore as the resource.

Again, draw how this should work, and verify that it does in fact work as expected.

You can get the current task's priority by calling rt_task_inquire():

```
struct rt_task_info temp;
rt_task_inquire(NULL, &temp);
rt_printf("Base prio: %i, Current prio: %i\n", temp.bprio, temp.cprio);
```

3. Deadlock and Priority Ceiling

You have previously fixed a deadlock in the dining philosophers problem, by carefully inspecting the deadlock situation and identifying its root cause. While careful consideration most certainly has its place in software design, it also has its limits, and bugs will make it through.

In this task, we will be using the Immediate Ceiling Priority Protocol (ICPP) solution, which should entirely eliminate the possibility of deadlocks when using resources (as long as you don't do something stupid like sleeping while holding a resource). It does this by assigning priorities to the resources protected by mutexes, where this priority should be higher than the priority of any of the tasks that use this resource. You must therefore know what tasks use what resources ahead of time. When a task takes a resource and locks the corresponding mutex, it will have its priority temporarily boosted to that of the resource.

Task D:

Create two tasks of different priorities:

- **Low**:
  Take mutex A
  Busy-wait for 3 time units
  Take mutex B
  Busy-wait for 3 time units
  Return B
  Return A
  Busy-wait for 1 time unit
- **High**:
  Sleep for 1 time unit
  Take mutex B
  Busy-wait for 1 time unit
  Take mutex A
  Busy-wait for 2 time units
  Return A
  Return B
  Busy-wait for 1 time unit

This should result in a deadlock, when both tasks are waiting for a resource held by the other. Make sure that this program deadlocks before you move on to the next task.

Task E:

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. Instrument your code with enough printf's that you can draw how this program behaves, and make sure that it behaves as you would expect ICPP to work.

It is recommended that you create some simple data structures to manage the priorities, one structure that contains the resource (its mutex and priority, maybe more items?) and one structure that contains a task's base and boost priority (again, possibly more items?). With this, you should be able to create fairly elegant `icpp_lock` and `icpp_unlock` functions.