Lab 3: Threads (Tasks), Priorities, Synchronization

Farhang Nemati, March 2016

In this lab you will practice working with real-time tasks using POSIX Threads (pthreads). You will practice the timing issues, interference among real-time tasks and synchronizing them on mutually exclusive accessed resources.

A. Delaying a task without sleeping

Usually when to delay a task you would use one of the delaying functions, for example *sleep(), clock_nanonsleep()*, etc. Calling these functions suspends the caller task (the task goes to sleep). However in this part you are required to delay a task without making it sleep (it should perform busy-wait for a given time interval). Create a task that turns on a LED for 1ms. The task will have a period of 4ms. This means that every 4ms the task turns on the LED for 1ms. For the project to run you have to include "rt" and "pthread" libraries; in "Project->VisualGDB Project Properties->Makefile settings->Library names" write rt pthread (notice that there is a space between them).

B. Two tasks with priorities

B.1.

In this part you need to create and run two tasks. The first task (*task1*) will be the same as you created in Section A. The second task (*task2*) will control the second LED on the breadboard. *task2* will have a period of 6ms and it also turns on its LED for 1ms. For delaying *task2* you have to use both busy-waiting and sleeping; for keeping the LED for 1ms use busy-waiting and for the remaining time (5ms) it has to suspend (sleep). Try the follwing alternatives when creating the tasks:

1. First create *task1* and then *task2*,
2. First create *task2* and then *task1*

Are both of the LEDs turned on in both alternatives? Explain the reason if one of them is not shining

B.2.

In the program assign priorities to the tasks. Notice that you have to give a priority to a task that is in the range of minimum and maximum priorities. There are functions to get the minimum and maximum possible priorities (see the hints!). Try the following alternatives:

1. Give a higher priority to *task2*
2. Give a higher priority to *task1*

Explain you observations when running the program.

B.3.

Change *task1* so that it performs its delays the same way as *task2* meaning that it delays with busy-waiting for the time the LED is on and it sleeps for the rest of its period. Increase the busy-wait duration (during which the LED is on) of *task1*and *task2* to 2ms and 3ms respectively. Try the following alternatives:

1. Give a higher priority to *task1*.
2. Give a higher priority to *task2*.

Explain you observations. Which priority setting is better?

C. Three tasks with synchronization

In this part you will create 3 tasks (*task1, task2,* and *task3*). *task1* and *task2* will control the LEDs using the PWM that you implemented in Lab 2. The periodic task *task3* will be running the terminal console to get a command from the user. Assign priorities to all 3 tasks. It's recommended that *task3* has a lower priority than other 2 tasks (why?). The command that is entered by the user in the terminal should be put in a shared variable by *task3*. The global variable is accessed by all 3 tasks. *task1* and *task2* check the command and use the command if it applicable to them and reset the command. If the command is not applicable to a task it should ignore the command. The program can hold only one command and if another command is entered before the previous one is consumed the new command should replace the old one. Since all 3 tasks access (share) the variable holding the command, you have to protect it. The program accepts a float number as the command according to the following protocol:

n.m

where *n* indicates the task number (1 for *task1* and 2 for *task2*) and *m* indicates the brightness of the LED controlled by task *n*. For example if the command is 1.25 it means that the LED controlled by *task1* will be shining for 25% of its full brightness. If the command is 2.5 it means that the LED controlled by *task2* will be shining for 50% of its full brightness. The LEDs have to continue with same state until a command changes it. If the value of *n* is 0 the tasks have to be canceled and the program exits. Any command other than those has to be ignored.

Hints: Consider using the following POSIX functions for this lab:

```
//The header file containing the declarations
#include <pthread.h>

//To Create and run a task
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_function)(void *), void
*arg);

//To get the current time from a clock. use CLOCK_REALTIME as clk_id
int clock_gettime(clockid_t clk_id, struct timespec *tp);

//To change the priority of a task (thread) use SCHED_FIFO as policy. param.sched_priority will be the
priority of thread
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);

//To get the maximum possible priority of a policy
int sched_get_priority_max(int policy);

//To get the minimum possible priority of a policy
int sched_get_priority_min(int policy);

//To cancel a thread
int pthread_cancel(pthread_t thread);

//To lock a mutex
int pthread_mutex_lock(pthread_mutex_t *mutex);

//To try to lock a mutex
int pthread_mutex_trylock(pthread_mutex_t *mutex);

//To unlock a mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```