Assignment 1. Real-time Tasks Models in Linux (100 points)

Assignment Objectives:

1. To program real-time tasks on Linux environment, including periodic and aporadic tasks, event handling, priority inheritance, etc.
2. To use Linux trace tools to view and analyze real-time scheduling.

Assignment:

As shown in the following diagram, periodic and aporadic tasks in real-time systems can be simply expressed as endless loops with time- and event-based triggers. In the task body, specific computation should be done and locks must be acquired when entering any critical sections.

```
TASK periodic_task()
{
        < local variables >
        initialization() and wait_for_activation();
        while (condition) {
                <task body>
                wait_for_period();
        }
}
```

```
TASK aperiodic_task()
{
        < local variables >
        initialization and() wait_for_activation();
        while (condition) {
                <task body>
                wait_for_event();
        }
}
```

(from "*Ptask: an Educational C Library for Programming Real-Time Systems on Linux*"
By Giorgio Buttazzo and Giuseppe Lipari, *EFTA 2013*)

In this assignment, you are asked to develop a program that uses POSIX threads to implement these task models on Linux environment. The task body is defined in the following BNF:

<task_body> ::= <compute> { < CS> <compute> }

<CS> ::= <lock_*m*> <compute> <unlock_*m* > | <lock_*m* > <compute><CS><compute> <unlock_*m* >

where "lock_*m*" and "unlock_*m*" are locking and unlocking operations on mutex *m*, and "compute" indicates a local computation. To simulate a local computation, we will use a busy loop of *x* iterations in the assignment, i.e.

```
int i, j=0;
for (i = 0; i < x; i++)
{
        j = j + i;
}
```

The input to your program is a specification of a task set which is shown in the following example:

```
line 1:  2 3000                      // there are 2 tasks and the execution terminates after 3000ms
line 2:  P 20 500 200 L3 300 U3 400  // task 1 is a periodic task of priority 20 with period 500ms.
                                     // In its task body, it runs the busy loop for 200 iterations, locks
                                     // mutex 3, runs the busy loop for 300 iterations, unlocks
                                     // mutex 3, and finally runs 400 iterations of the busy loop.
line 3: A 10 1 500                   // task 2 is an aperiodic task of priority 10. It is triggered by
                                     // event 1 and then runs 500 iterations of the busy loop.
```

Your program should read in a task set specification and create a thread for each task to perform task operations given in the input file. To verify the scheduling events of the tasks, you will need to use "*trace_cmd*" to collect "*sched_switch"* events from the Linux internal tracer *ftrace*. The traced records can then be viewed via a GUI front end *kernelshark*.

Additional requirements of the program are:

1. The task set specification is given in an input text file and you may assume that the format is always correct (i.e., no need to detect errors from the input file).
2. At most 10 mutex locks are needed for shared resources and two external events are considered. Event 0 and 1 arrive when we click the left and right buttons of a mouse, respectively.
3. When a task overruns, its next iteration should be started immediately as soon as the task finishes its current task body.
4. The priority numbers given in input files are real-time priority levels and tasks are scheduled under the real-time policy SCHED_FIFO.
5. All tasks should be activated at the same time. When the execution terminates, waiting tasks (wait_for_period or wait_for_event) should exit immediately. Any running or ready task should exit after completing the current iteration of its task body.
6. When tasks lock and unlock resources, make sure that the fastpath PI-enabled pthread mutexes are used.

Along with the program, you are required to prepare a report. In the report,

1. You need to verify the scheduling of the tasks using *kernelshark*. At least you need to show that 1) tasks are periodically scheduled under the SCHED_FIFO with the assigned priorities, 2) how a task is scheduled when the task overruns.
2. Please use a task set with priority inversion involved to compare and analyze the scheduling of tasks with and without PI-enabled mutexes.

Here are some suggestions you can consider:

- Since the number of tasks in the task set is not fixed, it is difficult to hard code all tasks. One approach is to develop two generic task functions (for periodic and aperiodic respectively) which take the specification of task body, and period or event, to perform task execution. In the main program, threads can be created with proper priorities and then call the task function with the corresponding parameters. Note that these two generic task functions must be reentrant.
- You may need a separate thread to read in mouse-click events from the device "evdev" and dispatch the events to the destination aperiodic tasks.

Due Date

This is assignment is due at 11:59pm on Feb. 12.

What to Turn in for Grading

1. Please create a working directory "*lastname_assign1*" to include all your submission:

readme		// text file
*lastname*_report1.pdf		//
subdirectories:	bin	// to hold binary files. It should be empty
	include	// for .h files
	source	// source program, make file
	test	// test cases and trace files

2. Compress the directory into a zip file named *cse522-firstname-lastname.zip*. Please note that, for convenience, we only accept zip files. Points will be deducted if the name conversion and the directory structure are not followed.

3. Comment your source files properly and write the readme file to describe how to use your software. Also, make sure there will be no warnings when compiling your source code.
4. Submit the zip archive to Blackboard by the due date and time.