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

As shown in the following diagram, periodic and aporadic tasks in real-time embedded 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();
        wait_for_event();
        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 as the following sequence:

<compute_1> <lock_*m*> <compute_2> <unlock_*m* > <compute_3>

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

```
volatile uint64_t x;
while(x > 0) x--;
```

The input to your program is a specification of a task set. The task parameters of each task is defined in *struct Tasks*:

```
struct Tasks
{
        int task_type;              // 0 for periodic and 1 for aperiodic task
        int task_num;               // task id (not pthread id)
        int event_key;              // Event number to trigger aperiodic task
        int priority;               // priority of the task (to be used as sched_priority in
                                    //                   struct sched_param)
        int period;                 // period for periodic task
        int loop_iter[3];           // loop iterations for compute_1, compute_2 and compute_3
        int mutex_m;                // the mutex id to be locked and unlocked by the task
};
```

For instance, in the given *task_model.h*, a task set of 5 tasks are defined:

```
#define THREAD0 {0, 0, 0, 70, 10, {100000, 100000, 100000}, 2}
#define THREAD1 {0, 1, 0, 65, 28, {100000, 100000, 100000}, 1}
#define THREAD2 {0, 2, 0, 64, 64, {100000, 100000, 100000}, 2}
```

```
#define THREAD3 {1, 3, 0, 80, 10, {100000, 100000, 100000}, 2}
#define THREAD4 {1, 4, 7, 85, 10, {100000, 100000, 100000}, 0}

struct Tasks threads[NUM_THREADS]={THREAD0, THREAD1, THREAD2, THREAD3, THREAD4};
```

Your program should use the task set specification defined in *task_model.h* and create a thread for each task to perform task operations periodically or upon event arrivals. 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. A "Makefile" should be used to compile source programs and generate a target executable file. This target executable file must be named as "assignment1".
2. The task set specification is given in *task_model.h* which should be included in your program. You can modify task parameters and the numeric constants defined by other macros to test your program. However, please do not insert additional code in *task_model.h* since a *task_model.h* with different numeric values will be used to grade your program.
3. The numeric constant *TOTAL_TIME* in *task_model.h* indicates the total execution time in *ms* that your program should run. When the execution is completed, all threads should be terminated properly. Any waiting tasks (wait_for_period or wait_for_event) should exit immediately. Any running or ready task should exit after completing its current iteration of task body.
4. At most *NUM_MUTEXES* mutex locks are needed for critical sections of task body. In addition, 10 external events are considered. Event *i* arrives when we release key *i* on the keyboard, where *i=0, 1, … 9*. The path name, "KEYBOARD_EVENT_DEV", for the keyboard device is defined in *task_model.h*. You may need to change this path name for your execution environment. Also, an event may trigger the execution of multiple aperiodic tasks.
5. For a periodic task, if an invocation of task body is not done before the end of the period, its next invocation should be started immediately as soon as the task finishes its current task body.
6. The priority numbers given in input files are real-time priority levels (i.e., *sched_priority* in *struct sched_param*) and tasks are scheduled under the real-time policy SCHED_FIFO. Your "main" thread and any other threads created should also be scheduled under the real-time SCHED_FIFO with proper priorities.
7. All tasks should be activated at the same time.
8. When tasks lock and unlock critical sections, make sure that the PI-enabled pthread mutexes are used.
9. All threads and the main process should be run on a specific CPU. This can be done with *taskset* command to set proper CPU affinity.

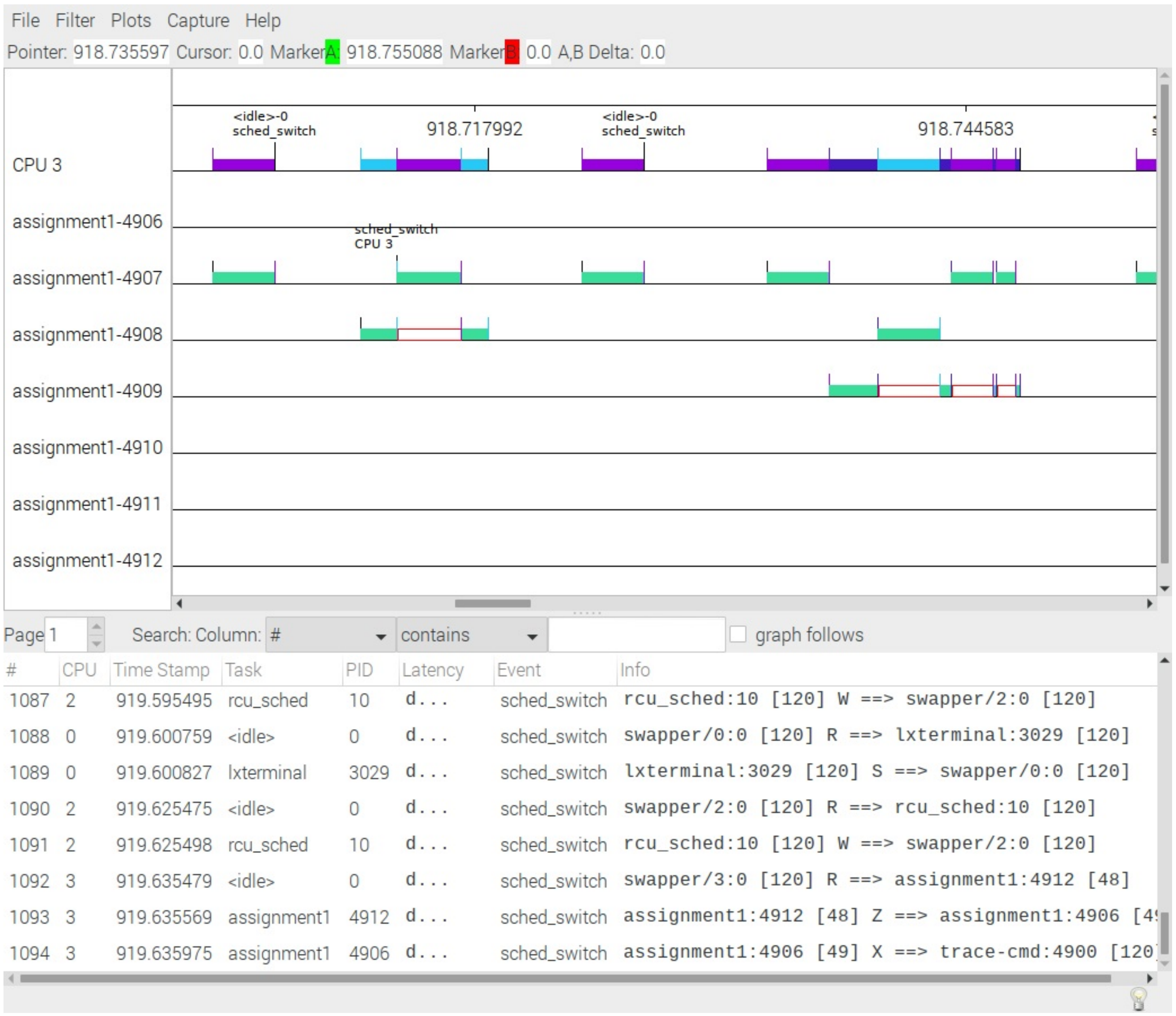
Along with the program, you are required to submit a pdf file that contains a screenshot from *kernelshark* to verify the scheduling of the tasks and the list of task parameters that you use to generate the screenshot. On the screenshot, please identify at least one occurrence of task blocking and preemption. An example screenshot is as follows.

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 in 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 keyboard events from the defined device and dispatch the events to trigger corresponding aperiodic tasks.
- You may need to install *trace-cmd* and *kernelshark* in your Linux system. On Raspberry pi, you can do the following steps:

  sudo apt-get update –y
  sudo apt-get install –y trace-cmd kernelshark



Pointer: 918.735597 Cursor: 0.0 MarkerA: 918.755088 MarkerB: 0.0 A,B Delta: 0.0

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
1087	2	919.595495	rcu_sched	10	d...	sched_switch	rcu_sched:10 [120] W ==> swapper/2:0 [120]
1088	0	919.600759	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> lxterminal:3029 [120]
1089	0	919.600827	lxterminal	3029	d...	sched_switch	lxterminal:3029 [120] S ==> swapper/0:0 [120]
1090	2	919.625475	<idle>	0	d...	sched_switch	swapper/2:0 [120] R ==> rcu_sched:10 [120]
1091	2	919.625498	rcu_sched	10	d...	sched_switch	rcu_sched:10 [120] W ==> swapper/2:0 [120]
1092	3	919.635479	<idle>	0	d...	sched_switch	swapper/3:0 [120] R ==> assignment1:4912 [48]
1093	3	919.635569	assignment1	4912	d...	sched_switch	assignment1:4912 [48] Z ==> assignment1:4906 [4
1094	3	919.635975	assignment1	4906	d...	sched_switch	assignment1:4906 [49] X ==> trace-cmd:4900 [120

Due Date

The due date is 11:59pm, Feb. 14.

What to Turn in for Grading

- Create a zip archive file named, named “ESP-LastName-FirstInitial-assgn01.zip”, to include your source files (.c and .h), Makefile, readme, and your screenshot in pdf format and submit the zip archive to Canvas by the due date. Note that any object code or temporary build files should not be included in the submission.

- Please make sure that you comment the source files properly and the readme file includes a description about how to make and use your software. Don't forget to add your name and ASU id in the readme file and in your report.
- There will be 20 points penalty per day if the submission is late. Note that submissions are time stamped by Canvas. If you have multiple submissions, only the newest one will be graded. If needed, you can send an email to the instructor and TA to drop a submission.
- The assignment must be done individually. No collaboration is allowed, except the open discussion in the forum on Canvas. The instructor reserves the right to ask any student to explain the work and adjust the grade accordingly.
- Failure to follow these instructions may cause deduction of points.
- Here are few general rule for deductions:
    - No make file or compilation error -- 0 point for the assignment.
    - Must have "–Wall" flag for compilation -- 5-point deduction for each warning.
    - 10-point deduction if no compilation or execution instruction in README file.
    - Source programs are not commented properly -- 10-20-point deduction.
- ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed.

How your program will be graded on Raspberry pi:

1. Unzip your submission
2. Replace *task_model.h* with a different one
3. Run make command
4. Use trace-cmd to generate the trace data of ./assignment1. The command to be used is
   sudo trace-cmd record –e sched_switch taskset –c 3 ./assignmen1t
5. Use kernelshark to view *trace.dat*
6. Examine your source code