Assignment 1 Report

The objective of the assignment, to schedule real time tasks on Linux and to use Linux trace tools to analyze the results, was achieved successfully. Many sample job sets were used to analyze if the scheduling done was acceptable. These sample files, along with the "trace.dat" file generated by the Linux trace tools, are available in the "test" folder. The folder also contains the screenshots of the GUI front end, "Kernelshark", which was used to view the trace records. A few of the important screenshots have been reproduced in the report, to help explain the results obtained.
The code for scheduling jobs given in the input file is available in the file "scheduler.c" present in the "source" directory. It can be compiled using the makefile in the same folder. The program starts the jobs given in the input file, the name of which is provided in the command line, and runs for the time specified in the same input file.
No separate header file was used, so the "include" folder is empty.

All jobs are properly scheduled
A screenshot of the trace, as seen in Kernelshark, is shown below:

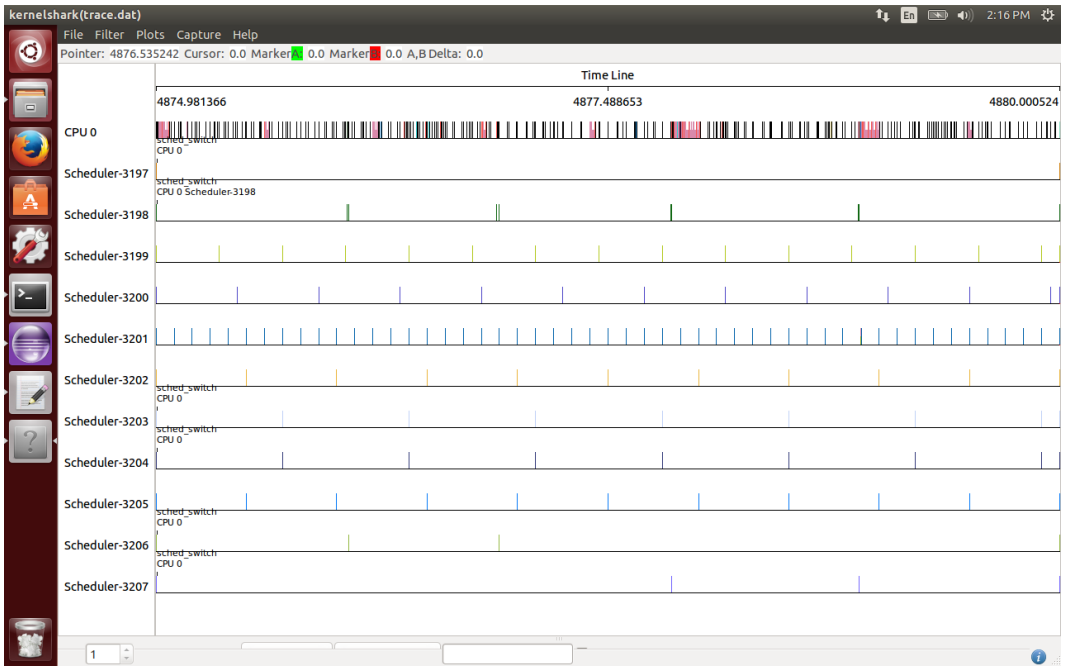

Figure 1: Job Scheduling

This is the trace run of the program with the input file "Sample 3.txt". As we can see, all the periodic jobs have been scheduled according to their period, and the aperiodic jobs execute when their corresponding event triggers them.

When many jobs arrive at the same time, the highest priority jobs executes first

A magnified image of the above trace shows which job executes first, if a number of jobs arrive together. The screenshot is reproduced below:

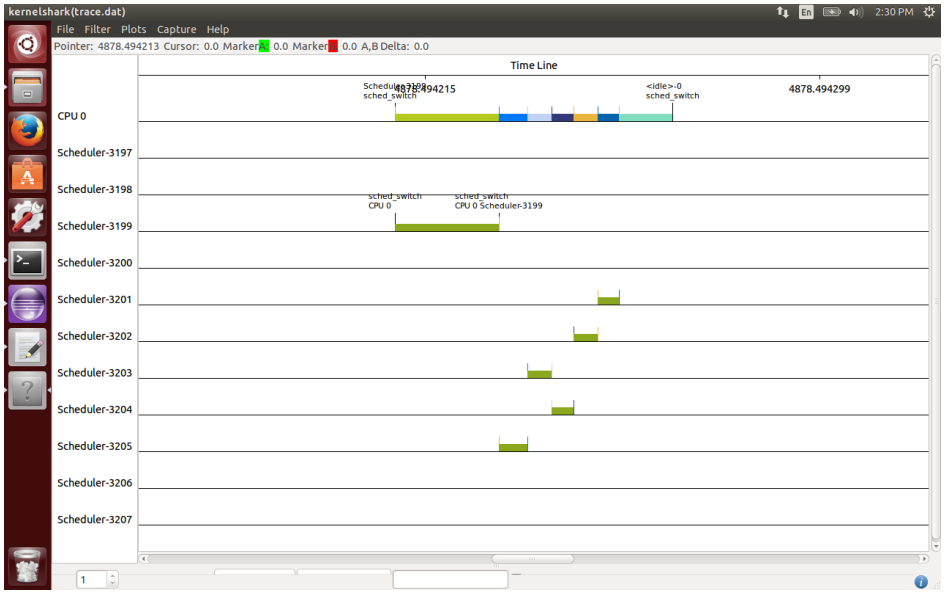

Figure 2: Priority

As we can see, there is a moment when six jobs arrive at the same time. We can clearly see in the above screenshot that the execution order of the jobs depend on their priorities. The first two threads are the threads executing the main function and the read event. The third thread (Scheduler-3199 as seen in the trace) is the first job as given in the sample file. It has the highest priority, and thus executes first. The third job (Scheduler-3201) has the least priority, and executes last. The other jobs execute in their relative priority order as well. Although the priorities of the job cannot be seen in the above screenshot, it can be verified with the input sample file.

If a task overruns, its next iterations begins as soon as it finishes its current task body

If a task overruns, the program makes sure that its next iteration begins as soon as it finishes its current iteration, provided no other higher priority job arrives. A magnified screenshot from running the jobs listed in the sample file "Sample_OT" is shown below:

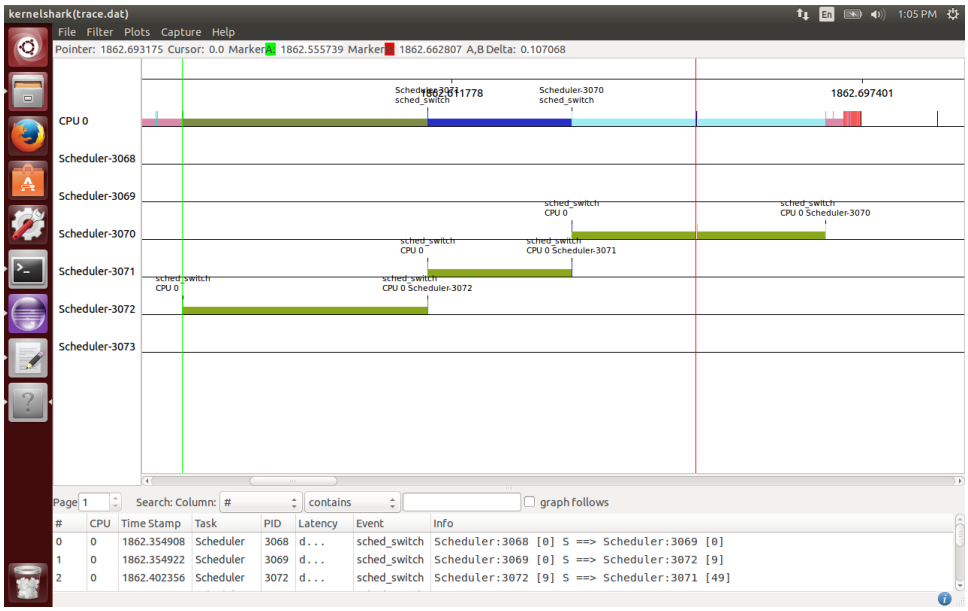

Figure 3: Task Overrun

As we can see, the first job (Scheduler-3070) is not able to finish within its period of 100 milliseconds. So, it stars its next iteration as soon as it finishes its current iteration.

Priority Inversion

We observe priority inversion while running the jobs in the file “Sample_PI.txt”, if we do not use priority inheritance in the mutexes. The screenshot is shown below:

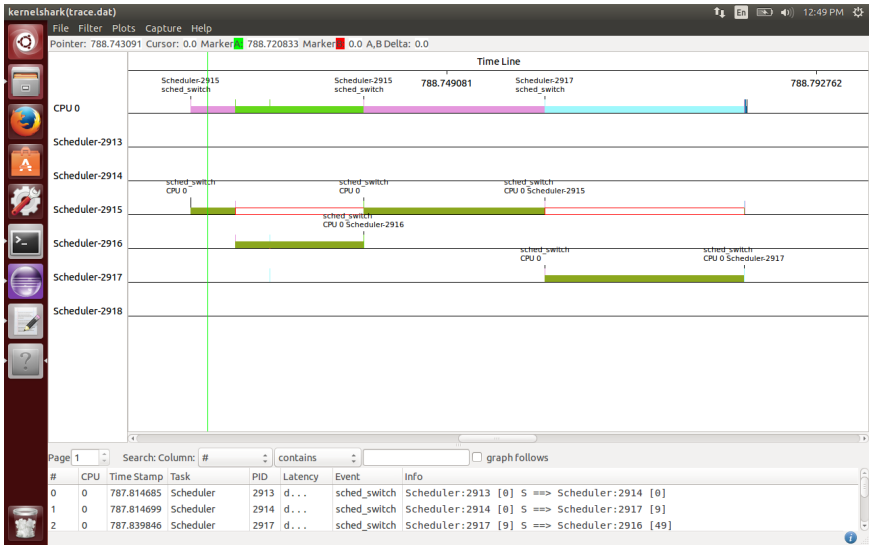

Figure 4: Priority Inversion

We see that even though the third job (Scheduler-2197) has the highest priority, it is not able to run, as it is waiting for a mutex locked by the first thread (Scheduler-2195), which is of a low priority. The first thread is preempted by the second thread (Scheduler-2196), which has a priority higher than the first thread, and lower than the third thread. The net result is that the third thread needs to wait for the second thread, even though the third thread has a higher priority than the second thread.

If we use priority inheritance, we observe that the second thread is preempted to allow the first thread to run, as soon as the third thread calls for a lock on the mutex. The screenshot can be seen below:

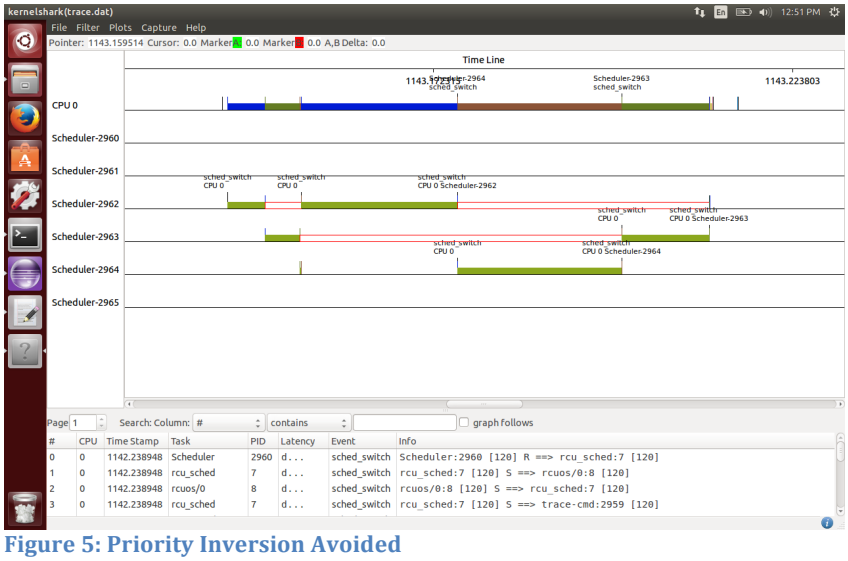

Figure 5: Priority Inversion Avoided