

### Assignment 1 Report

In order to verify that our program meets the scheduling requirements of the assignment we use trace-cmd to generate a .dat file on our host machine that can then be used to display useful information about the program's thread management on the kernelshark application. With the following command we are able to ensure that all the threads run on one CPU and that we record the sched\_switch and sched\_wakeup events:

```
sudo trace-cmd record -e sched_switch -e sched_wakeup numactl --physcpubind=0 ./homework1 0 <
testcase9.txt
```

On the Galileo board we configure ftrace to gather the same information and verify that the behavior is consistent on our target system. First, we use kernelshark to verify that our program is working correctly. We trace the program with the following input file:

```
S 10000
P 50 900 200 L3 300 U3 400
P 20 400 200 L3 300 U3 400
A 10 0 500
P 10 300 200 L2 100 L3 500 U3 100 L4 60 U4 200 U2 200
A 30 1 500
```

We then feed the resulting trace data into kernelshark and use it to generate graphs for the threads of our program, in this case called a.out. This generates the following graph:

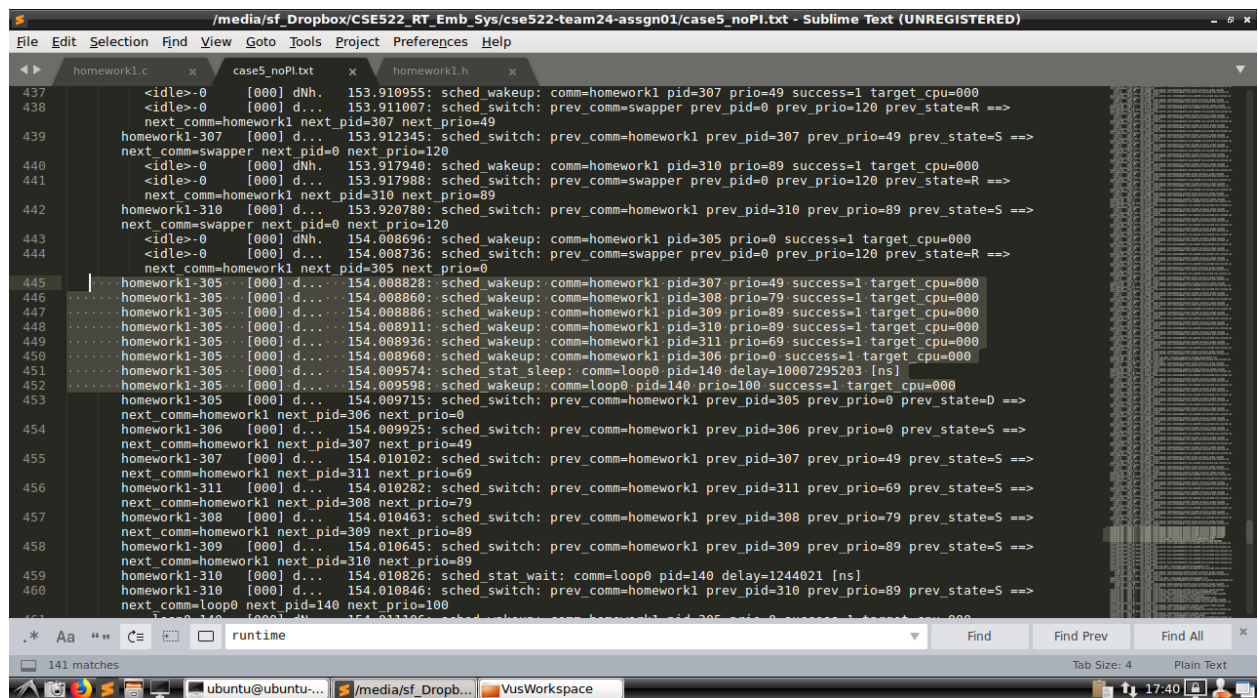


***Figure 1: Testing our program with Kernelshark***

As shown in Figure 1, we see that our program has 8 threads running. The first is the main which does some work at the beginning. The second is our timer thread which times the execution time and signals all other threads to terminate (gracefully) upon time up. This is followed by our mouse listener thread that reads from /dev/input/event2 which is the USB port the mouse is connected to on the Galileo Board and on our host machine. Then come the threads we created with our input text file:

- 1) Periodic task with a period of 900 ms
- 2) Period task with a period of 400 ms
- 3) Aperiodic task listening on event 0 (Left Mouse Button)
- 4) Periodic task with a period of 300 ms
- 5) Aperiodic task listening on event 1 (Right Mouse Button)

Notice that we can see each of the periodic tasks are being context-switched as they wake up according to their respective periods. The aperiodic tasks match with the mouse listener thread when the correct event is triggered for each thread. Therefore, we can verify that our program is performing as expected. Also notice that all threads wait for activation at the beginning, and we can see this barrier. Once the last thread arrives, then all of them begin execution at the same time. They also terminate execution gracefully upon time up. We also verify that this is true on the Galileo board with ftrace. First, we find the portion where the timer thread wakes up and signals all other threads that the time is up so they should look to exit as shown in figure 1a:



```

437 <idle>-0 [000] dNh. 153.910955: sched wakeup: comm=homework1 pid=307 prio=49 success=1 target_cpu=000
438 <idle>-0 [000] d... 153.911007: sched switch: prev_comm=swapper prev_pid=0 prev_prio=120 prev_state=R ==>
      next_comm=homework1 next_pid=307 next_prio=49
439 homework1-307 [000] d... 153.912345: sched switch: prev_comm=homework1 prev_pid=307 prev_prio=49 prev_state=S ==>
      next_comm=swapper next_pid=0 next_prio=120
440 <idle>-0 [000] dNh. 153.917940: sched wakeup: comm=homework1 pid=310 prio=89 success=1 target_cpu=000
441 <idle>-0 [000] d... 153.917988: sched switch: prev_comm=swapper prev_pid=0 prev_prio=120 prev_state=R ==>
      next_comm=homework1 next_pid=310 next_prio=89
442 homework1-310 [000] d... 153.920780: sched switch: prev_comm=homework1 prev_pid=310 prev_prio=89 prev_state=S ==>
      next_comm=swapper next_pid=0 next_prio=120
443 <idle>-0 [000] dNh. 154.008696: sched wakeup: comm=homework1 pid=305 prio=0 success=1 target_cpu=000
444 <idle>-0 [000] d... 154.008736: sched switch: prev_comm=swapper prev_pid=0 prev_prio=120 prev_state=R ==>
      next_comm=homework1 next_pid=305 next_prio=0
445 homework1-305 [000] d... 154.008828: sched wakeup: comm=homework1 pid=307 prio=49 success=1 target_cpu=000
446 homework1-305 [000] d... 154.008860: sched wakeup: comm=homework1 pid=308 prio=79 success=1 target_cpu=000
447 homework1-305 [000] d... 154.008886: sched wakeup: comm=homework1 pid=309 prio=89 success=1 target_cpu=000
448 homework1-305 [000] d... 154.008911: sched wakeup: comm=homework1 pid=310 prio=89 success=1 target_cpu=000
449 homework1-305 [000] d... 154.008936: sched wakeup: comm=homework1 pid=311 prio=69 success=1 target_cpu=000
450 homework1-305 [000] d... 154.008960: sched wakeup: comm=homework1 pid=306 prio=0 success=1 target_cpu=000
451 homework1-305 [000] d... 154.009574: sched stat sleep: comm=loop0 pid=140 delay=10007295203 [ns]
452 homework1-305 [000] d... 154.009598: sched wakeup: comm=loop0 pid=140 prio=100 success=1 target_cpu=000
453 homework1-305 [000] d... 154.009715: sched switch: prev_comm=homework1 prev_pid=305 prev_prio=0 prev_state=D ==>
      next_comm=homework1 next_pid=306 next_prio=0
454 homework1-306 [000] d... 154.009925: sched switch: prev_comm=homework1 prev_pid=306 prev_prio=0 prev_state=S ==>
      next_comm=homework1 next_pid=307 next_prio=49
455 homework1-307 [000] d... 154.010102: sched switch: prev_comm=homework1 prev_pid=307 prev_prio=49 prev_state=S ==>
      next_comm=homework1 next_pid=311 next_prio=69
456 homework1-311 [000] d... 154.010202: sched switch: prev_comm=homework1 prev_pid=311 prev_prio=69 prev_state=S ==>
      next_comm=homework1 next_pid=308 next_prio=79
457 homework1-308 [000] d... 154.010463: sched switch: prev_comm=homework1 prev_pid=308 prev_prio=79 prev_state=S ==>
      next_comm=homework1 next_pid=309 next_prio=89
458 homework1-309 [000] d... 154.010645: sched switch: prev_comm=homework1 prev_pid=309 prev_prio=89 prev_state=S ==>
      next_comm=homework1 next_pid=310 next_prio=89
459 homework1-310 [000] d... 154.010826: sched stat wait: comm=loop0 pid=140 delay=1244021 [ns]
460 homework1-310 [000] d... 154.010846: sched switch: prev_comm=homework1 prev_pid=310 prev_prio=89 prev_state=S ==>
      next_comm=loop0 next_pid=140 next_prio=100

```

*Figure 1a: Timer signals all threads that time is up*

However, we can see in figure 1b that threads terminate gracefully because some that are simply waiting exit right away while others that are running take slightly longer to call process\_exit as shown below:

*Figure 1b: Snippet of Graceful termination on Galileo Board*

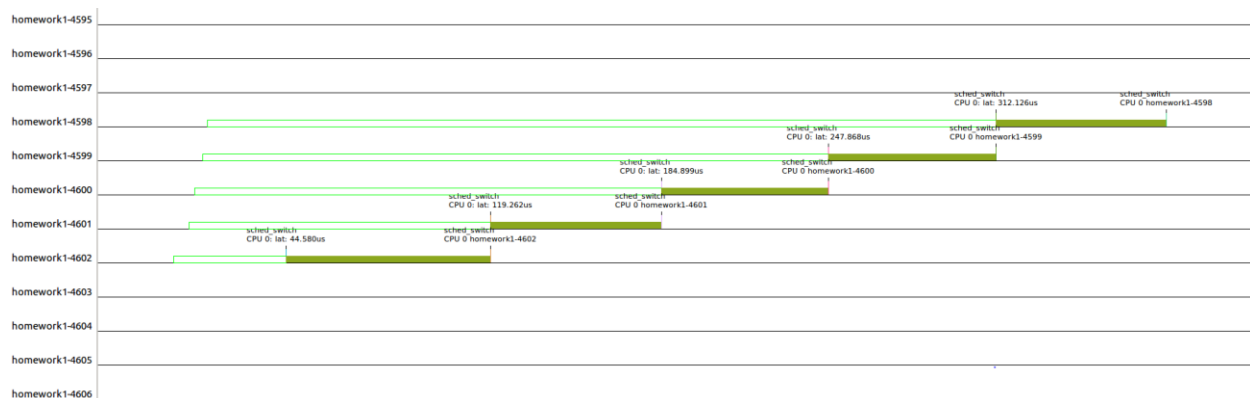
Next, we want to verify that the threads use FIFO scheduling protocol with the correct priorities. The easiest way for us to verify this was to produce print statements which can be seen upon program execution. However, we can further ensure this is the case by taking a closer look at the kernalshark graphs. Take the following input file for example:

```

9 10000
P 10 200 20000
P 30 200 20000
P 50 200 20000
P 70 200 20000
P 90 200 20000
A 20 0 20000
A 40 1 20000
A 60 0 20000
A 80 1 20000

```

We use this test case to generate another set of threads where the periodic thread's task bodies should overlap to see if they are being scheduled correctly. We can zoom into the graph to see what the threads are doing as shown below in figure 2.



Indeed we see the expected FIFO behavior according to the priorities given in the input file. We see that the five periodic tasks wake up at around the same time. However, they are scheduled based on their priority and time they came in. Task 5 has the highest priority of 90 and actually waits for another process running on the CPU but it is the first to be scheduled since it had the highest priority of the 5. Then they are scheduled in descending priority until they have all had a chance to execute which is what we were expecting for our scheduling policy. Therefore, our tasks are scheduled under the SCHED\_FIFO with the assigned priorities.

We then also test this on our Galileo Board and we can verify that the same behavior happens on the board. Figure 2a shows a part of the trace log where you can see threads have arrived at the same time but they are scheduled according to FIFO and depending on their priority.

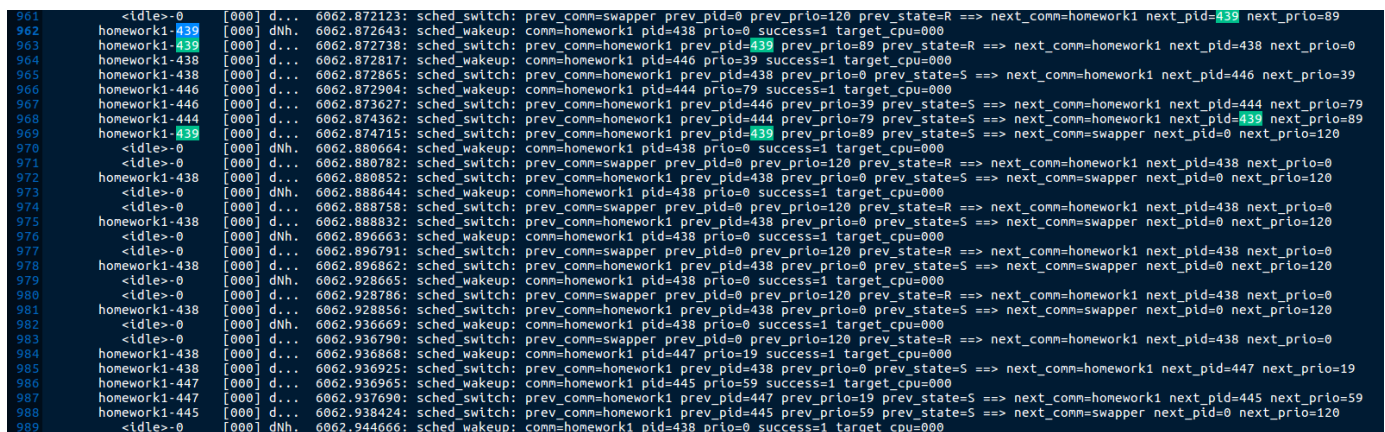


Figure 2a: FIFO Scheduling on Galileo Board

As shown in Figure 2a, there are cases where several threads are attempting to se the CPU at once. However, they are scheduled according to their priority. It is important to note that in the kernel priority actually goes from 0 – 100 with 0 being the highest and anything above 100 denotes a real-time user level scheduling. Therefore, the values in our program/input file need to be inverted and offset by one. So for example, a thread with priority 20 actually shows up in the log as  $(100-20) - 1 = 79$ . Therefore the lower this number the higher the actual priority. With that figured out, we see that indeed threads

of higher priority as scheduled first as they come it. We can see in figure 2a that a thread is schedule switched based on priority. For example, we see in line 967 that first the thread with the highest priority of 39 (which is actually 60) is scheduled, followed by the one with 79 and finally the one with 89, even though they did not wake up in that order necessarily. Therefore, we can confirm that our program also executes correctly on the board.

Additionally, this test case can also help us show that a mouse event will wake up all aperiodic threads waiting on that event as shown below in figure 3.

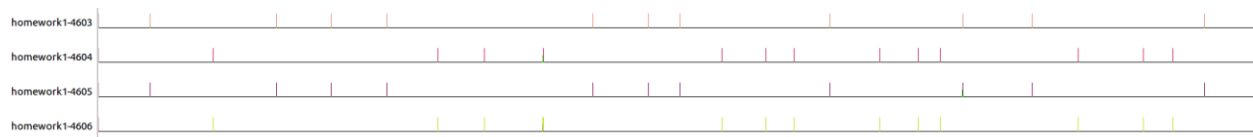


Figure 3: Mouse Events

Recall that the aperiodic tasks 6 and 8 were waiting on the left mouse event while tasks 7 and 9 were waiting on the right mouse release event. However, we see that when the mouse events are triggered all tasks waiting on the respective event are triggered simultaneously, thus the duplicated plots on figure 3.

Then, we look deeper to try to determine the behavior with priority inheritance (PI) enabled mutexes versus no PI mutexes. We test PI behavior with the following file:

```
S 10000
P 50 100 2000000 L3 3000000 U3 400000
P 10 200 2000000 L3 4000000 U3 400000
P 30 300 2000000 L2 3000000 U2 400000
A 20 0 500
A 40 1 500
```

Notice that we made the time spent in the task body longer and the periods and locks conflicting such that we could better observe this behavior. When we run our program with PI disabled we indeed notice that in some cases there is a delay to a task's execution caused by interference from lower priority tasks.

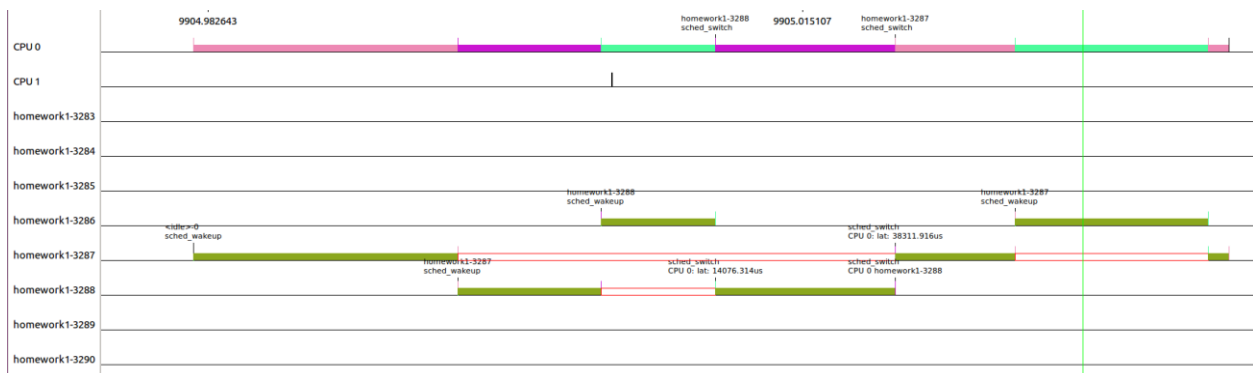
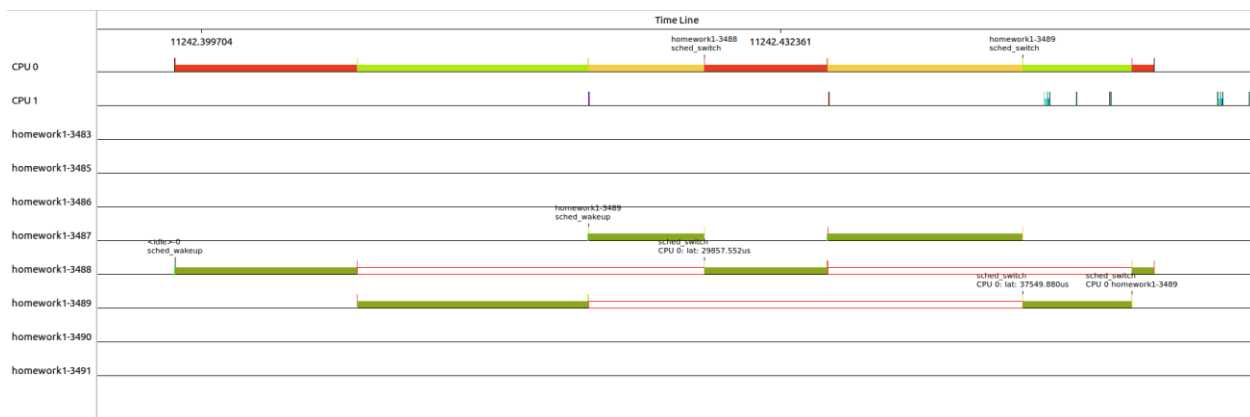


Figure 4: Priority Inheritance Disabled

Recall from our input file that there are 3 periodic tasks running. Task 1 has priority 50, task 2 has priority 10, and task 3 has priority 30. Task 1 and 2 both attempt to lock resource 3 while task 3 wishes to lock resource 2. The tasks plots from kernelshark are shown in Figure 4. We see that task 2 is running and is then pre-empted when task 3 of higher priority arrives. However, during this time task 2 has entered the critical section of its execution and thus acquired lock 3. Then task 1 with the highest priority arrives and pre-empts task 3 until it attempts to access resource 3 that task 2 has already acquired the lock for, therefore it is preempted by it. Task 1 is forced to wait on task 2, however, in that time task 3 wakes up and since it has a higher priority than task 2, it pre-empts that tasks and runs its execution. We then come back to task 2 and finally task 1 once task 2 releases the lock on resource 3. Only then can task 1 finish and then task 2 do the final iterations. This is an example of priority inversion where task 1 is forced to wait on a lower priority task because it has been blocked by a task with an even lower priority.

In order to prevent this, we can enable priority inheritance. This allows a blocking thread to inherit the priority of the thread it is blocking if it is higher. This ensures that the waiting thread will not wait on other tasks of lower priority leading to inversion. In fact, when we enable PI in our program and generate a trace, we cannot find an instance of priority inversion. Every time task 1 is blocked by task 2, task 2's execution will not be pre-empted by any of the other tasks because it has inherited task 1's priority of 50.



*Figure 5: Priority Inheritance Enabled*

As shown in figure 5, task 2 is executing first as in figure 2 and acquires the lock to resource 3. It is then pre-empted when task 3 of higher priority arrives. Then task 1 begins executing as it has the highest priority. However, now when task 1 arrives to the lock and is blocked by task 2 it gives its priority to task 2 to ensure it does not get jumped by some other task. Therefore, this time task 2 continues execution, not task 3, because task 2 has inherited task 1's priority of 50. Once task 2 releases the lock then task 1 can begin execution again without having to wait any unexpected time. Finally, task 3 finishes and then task 2 does the last of its operations which were not in the critical section, so these do have to wait for task 2 as it would be under its own priority of 10. Therefore, we only ever see task 1 wait on task 2 if task 2 had acquired the resource first. There is no instance where task 1's execution time takes an unexpected delay from other tasks jumping priorities. We also verify this behavior holds true on the Galileo board by running the ftrace function and collecting data for sched\_switch, sched\_wakeup, sched\_stat\_wait, process\_free and process\_frees events.