Linux Scheduler Profiling
CSE522S Lab 2

Sam Frank / sjfrank@wustl.edu
Ethan Vaughan / evaughan@wustl.edu
Erik Wijmans / erikwijmans@wustl.edu

October 22, 2016

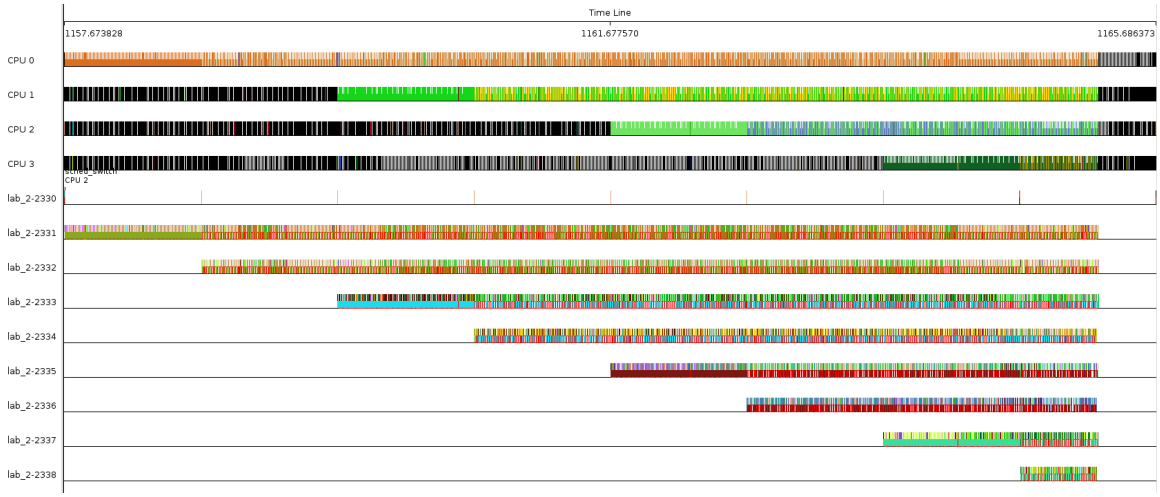

Figure 1: This figure show the kernelshark visualization of our barrier testing. As can be seen, all of the threads stop executing at the same time despite spawning at very different times.

1 Design Decisions and Process

We chose to build our profiling application in the Rust programming language. Rust is a systems-focused language with an emphasis on safe concurrency, and thus it was well-suited for this lab. It has many modern syntactical features, drawing inspiration from more functional programming languages.

In addition, we developed a script in Python to parse the output of `trace-cmd`, which allowed us to efficiently calculate certain runtime measures of our threads' execution. The script examines the `trace-cmd` output for a given interval of time and computes each thread's runtime proportion, as well as its average and maximum execution time before being preempted.

In total, we spent approximately 15 hours on this lab: 10 for gathering data and writing the lab report, and 5 for implementing the actual code.

2 Thread Barrier

In order to make the effect of the spin barrier immediately obvious, we added an artificial delay between the spawning of any two adjacent threads. We assumed that if the spin barrier was non-functional, each thread would end at a different time, as they all had equivalent workloads. In fact, we made our delay so long, and our workload short enough, that a newly spawned thread would finish before the next one was created.

Instead, we observed that all threads terminated at the same time, showing that our spin barrier functioned correctly. Figure 1 demonstrates this behavior, which was recorded via `trace-cmd`.

3 Effect of nice values under `SCHED_NORM`

To examine the effect of nice values on thread scheduling for `SCHED_NORM`, we measured the proportion of execution time that each thread on a core (high and low priority) received in a given interval

Nice Value	Observed	Expected
-5	88.5%	$\frac{1.25^5}{1.25^{-5}+1.25^5} = 90.3\%$
5	9.5%	$\frac{1.25^{-5}}{1.25^{-5}+1.25^5} = 9.7\%$

Figure 2: This table shows the results of our observed and expected runtime proportions by two threads with different nice values on the same core. The expected value is slightly higher than the observed as the expected value assumes that the two test threads are the only threads that run on that core.

of time. Then, we compared the observed proportions to those that we would expect given each thread's nice value. The expected runtime proportion of a thread is its weight divided by the sum of all weights. Under CFS, process weights are given by the formula:

$$weight = \frac{1024}{1.25^{nice}}$$

The expected and observed values for the runtime proportions are contained in Figure 2.

4 Behavior with real time priorities

When we switched to using a real time scheduling class, we frequently observed livelock on each core, resulting in a completely frozen raspberry pi. We hypothesized that the livelock was due to a high priority thread spinnning at the spin barrier before the low priority thread for that core had encountered the spin barrier. Because the low priority thread could not preempt the high priority thread, none of the threads could advance past the spin barrier.

We solved this issue by using two different spin barriers; one for the high priority threads and one for the low priority threads. This allowed the high priority threads to synchronize and begin their work without waiting for a low priority thread that would never run.

Using `trace-cmd`, we profiled the behavior of the program under the two real-time scheduling classes `SCHED_RR` and `SCHED_FIFO`.

4.1 `SCHED_RR`

Figure 3 visualizes the behavior of the program under `SCHED_RR`. On each core, the high priority thread runs to completion before the low priority thread gets a chance to run. This is the behavior we would expect with non-equal hard priorities. If the priorities were equal, we would see a round-robin effect between the two threads.

4.2 `SCHED_FIFO`

Figure 5 visualizes the behavior of the program under `SCHED_FIFO`. We observed the same behavior as `SCHED_RR`, where the high priority thread runs to completion before the low priority thread runs. The reason that the behavior is the same is because of the differential thread priorities–that is, there is no need to break a tie between two equal-priority threads.

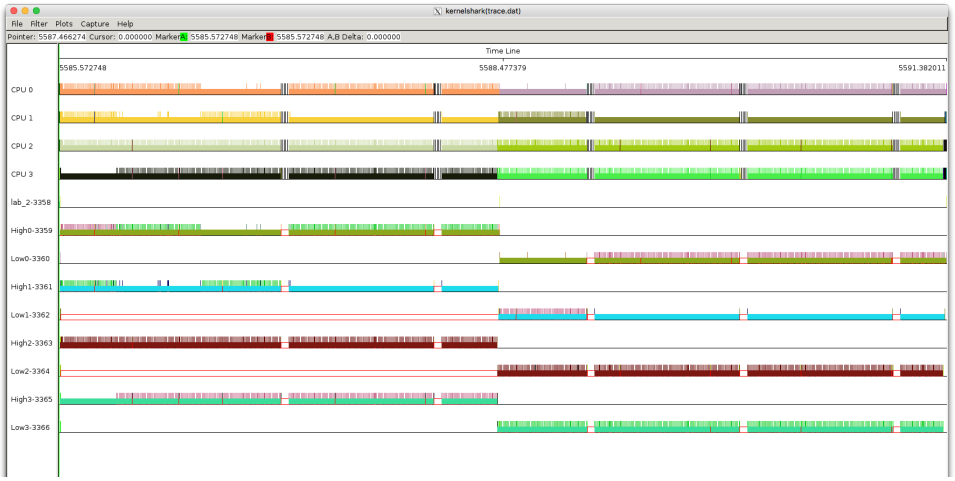

Figure 3: Kernelshark visualization of sched_switch events under `SCHED_RR`

Priority	Average	Maximum
15	3.7 ms	50 ms
5	0.013 ms	0.015 ms

Figure 4: This table shows each thread's average and maximum execution interval before being preempted under the `Sched_RR` policy. The average is calculated with outlier.

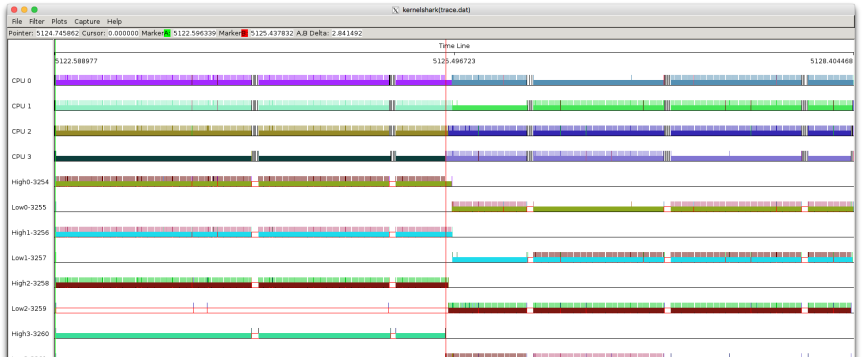

Figure 5: Kernelshark visualization of sched_switch events under `SCHED_FIFO`

For both scheduling classes, we measured each thread's average and maximum execution interval before being preempted. These values are documented in Figures 4 and 6. (Note: sample sizes are very small, so there is large variance)

Priority	Average	Maximum
15	3.7 ms	216 ms
5	0.03 ms	0.06 ms

Figure 6: This table shows each thread's average and maximum execution interval before being preempted under the `Sched_FIFO` policy. The average is calculated with outlier.

5 Behavior using four threads per core

5.1 SCHED_NORM

Figure 7 visualizes the behavior of the program under `SCHED_NORM` with four threads per core. As before, the high priority threads receive more execution time in a given interval than the low priority threads, but now within a given priority level, the threads receive the same runtime proportion. The calculations for the expected and observed runtime proportions are contained in Figure 8. For each priority, the expected runtime proportion is halved because there are now two threads running at that priority.

Figure 7: Kernelshark visualization of sched_switch events under `SCHED_NORM` with 4 threads per core

5.2 SCHED_RR

Figure 9 visualizes the behavior under `SCHED_RR` with four threads per core. As before, the high priority threads run to completion before the low priority threads get a chance to execute. But now with multiple threads at the same priority level, the round-robin behavior for tie-breaking is evident.

Nice Value	Observed	Expected
-5	44.75%	$\frac{1}{2}\frac{1.25^5}{1.25^{-5}+1.25^5} = 45.15\%$
5	4.77%	$\frac{1}{2}\frac{1.25^{-5}}{1.25^{-5}+1.25^5} = 9.85\%$

Figure 8: This table shows the results of our observed and calculated runtime proportions of four threads with two distinct nice values per core under the `SCHED_NORM` policy

Figure 9: Kernelshark visualization of sched_switch events under `SCHED_RR`

Thread	Average	Maximum
HighA	5.1 ms	7.6 ms
HighB	5.2 ms	7.5 ms

Figure 10: This table shows each high priority thread's average and maximum execution interval before being preempted under the `Sched_RR` policy.

5.3 SCHED_FIFO

With `SCHED_FIFO`, we needed to increase the number of spin barriers to 4, one for each family of threads. This is because once a thread reached the barrier, it would never yield and allow the other thread of equal priority on that core to reach the barrier. This would freeze the raspberry pi.

Figure 11 visualizes the very strange behavior of `SCHED_FIFO`. For long perioids of time, CPU cores will be idle and none of our test threads will be scheduled despite those threads being available to run.

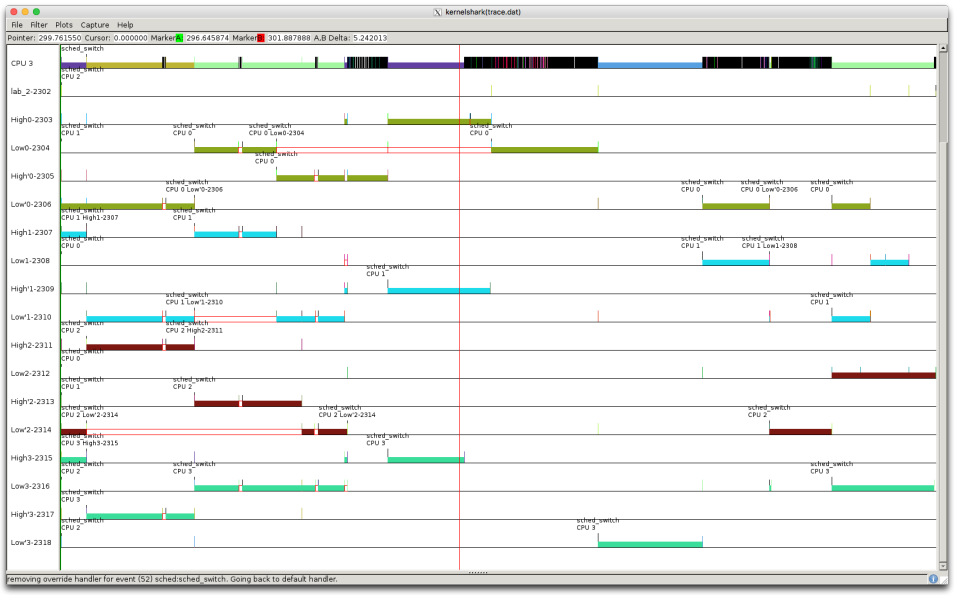

Figure 11: Kernelshark visualization of sched_switch events under `SCHED_FIFO`

Thread	Average	Maximum
HighA	300 ms	684 ms
HighB	35 ms	110 ms

Figure 12: This table shows each high priority thread's average and maximum execution interval before being preempted under the `Sched_FIFO` policy. As can be seen in Figure 11, sometimes the kernel processes that interrupt our process will stop running allowing one thread to run uninterrupted for an extremely long time.