Lab 4: Signal Generation

By Alec Newport [acn55] and Eldor Bekpulatov [eb654]
Thursday Lab - Section 403
November 21, 2019

Introduction

In this lab, we compared the performance of real-time and non-real-time embedded operating systems. To do this, we performed a number of tests both on the normal Raspbian kernel we've been using all semester and Raspbian 4.14.71 with the PreemptRT patch, which makes the operating system real-time. Real-time systems sacrifice increased complexity for more predictable process scheduling.

Part 1

For the first part of this lab, we characterized the latency of the Raspbian OS without the real-time patch. To do this, we downloaded and installed a program called Cyclictest which tracks program execution latency and generates histogram data. First, we tested the execution of four threads running under normal CPU load conditions (Figure 1). The peak was around bin 13, and there's a large variance in the latency. Then, we tested the same threads under 100% CPU load. To accomplish this, we ran a number of copies of sort_v1.c in the background (we started them using a bash script called sort_loop that loops through the run command). We had to modify this script to sort a larger array to ensure a 100% CPU utilization throughout the cyclictest execution. The results of this run are shown in Figure 2. The peak is in bin 11, which is about the same as under low-load conditions. The variance, however, is much smaller under high CPU load. This is likely due to the fact that the task scheduler is cycling through the available tasks in a more predictable fashion when the CPU is at 100% utilization. Therefore, tasks are executed in a more predictable amount of time.

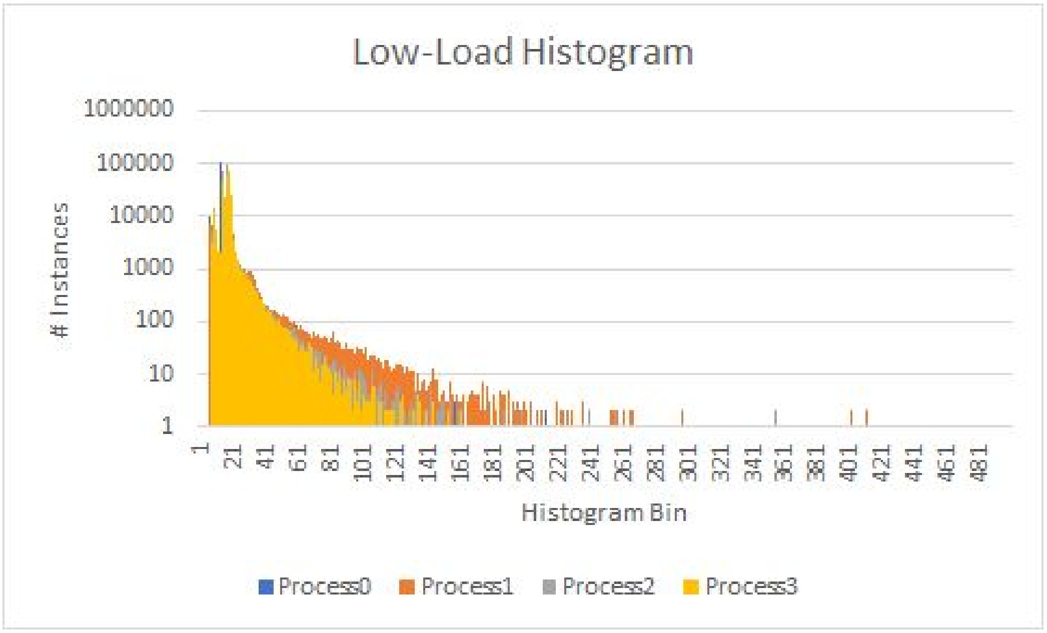


Figure 1: Raspbian Low Load Histogram

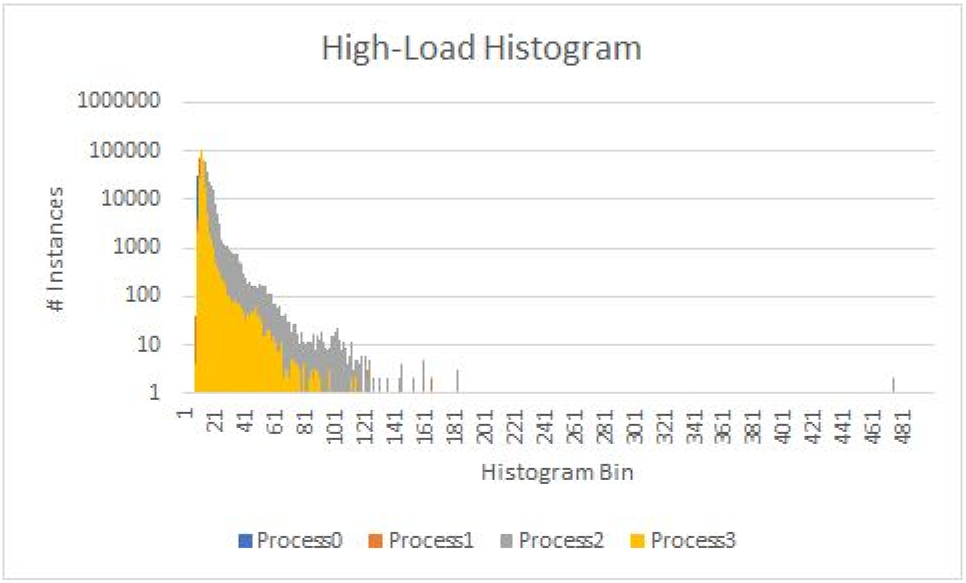
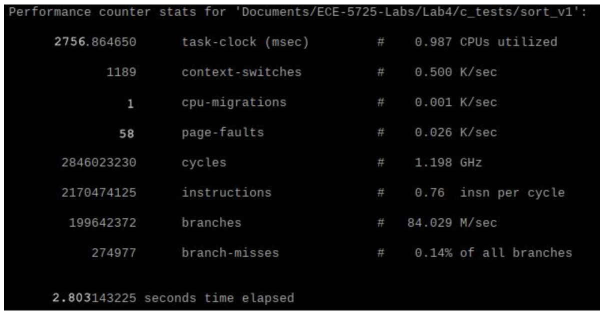


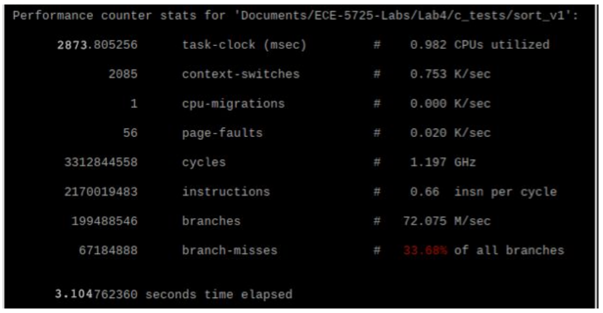
Figure 2: Raspbian High Load Histogram

Next, we used the same performance tool we've used in previous labs, Perf, to characterize the execution of sort_v1.c itself. This is a reasonably straightforward quicksort on an array. A function called comp compares two numbers in the array and returns 1 if the first element is greater than the second, -1 if the second is greater than the first, and 0 if they're equal. We ran two versions of the test, one with sorted and the other without sorting it. As expected, the sorted script ran a little faster, however it is important to note that sorted script reduced the number of branch misses in the execution of the program thus saving clock cycles. Those results are shown in Figure 3.



```
Performance counter stats for 'Documents/ECE-5725-Labs/Lab4/c_tests/sort_v1':

       2756.864650      task-clock (msec)         #    0.987 CPUs utilized
              1189      context-switches          #    0.500 K/sec
                 1      cpu-migrations            #    0.001 K/sec
                58      page-faults               #    0.026 K/sec
        2846023230      cycles                    #    1.198 GHz
        2170474125      instructions              #    0.76  insn per cycle
         199642372      branches                  #   84.029 M/sec
            274977      branch-misses             #    0.14% of all branches

       2.803143225 seconds time elapsed
```

Sorted

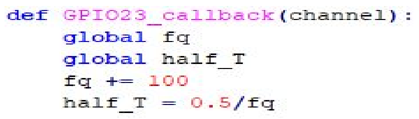


```
Performance counter stats for 'Documents/ECE-5725-Labs/Lab4/c_tests/sort_v1':

       2873.805256      task-clock (msec)         #    0.982 CPUs utilized
              2085      context-switches          #    0.753 K/sec
                 1      cpu-migrations            #    0.000 K/sec
                56      page-faults               #    0.020 K/sec
        3312844558      cycles                    #    1.197 GHz
        2170019483      instructions              #    0.66  insn per cycle
         199488546      branches                  #   72.075 M/sec
          67184888      branch-misses             #   33.68% of all branches

       3.104762360 seconds time elapsed
```

Unsorted

Figure 3: Raspbian Perf sort_v1.c Results

Finally, we generated square waves in both Python and C code and observed the maximum frequency we could accurately read on an oscilloscope. For the Python code, we adapter our blink.py script from an earlier lab. Its main loop simply toggles the value on the output pin once every half-period using a while loop. We also implemented two external buttons, one to exit the while loop and terminate the script, and one to increase the frequency so we didn't need to re-run the code every time we wanted to test a new value in the sweep (Figure 4).



```
def GPIO23_callback(channel):
    global fq
    global half_T
    fq += 100
    half_T = 0.5/fq
```

Figure 4: Frequency Step Callback

The results of the frequency sweep in Python are shown in Figure 5. The maximum frequency that is within 10% of its nominal value is about 700 Hz. This cap is so low because Python adds a lot of code around the script we write from libraries and other miscellaneous overhead.

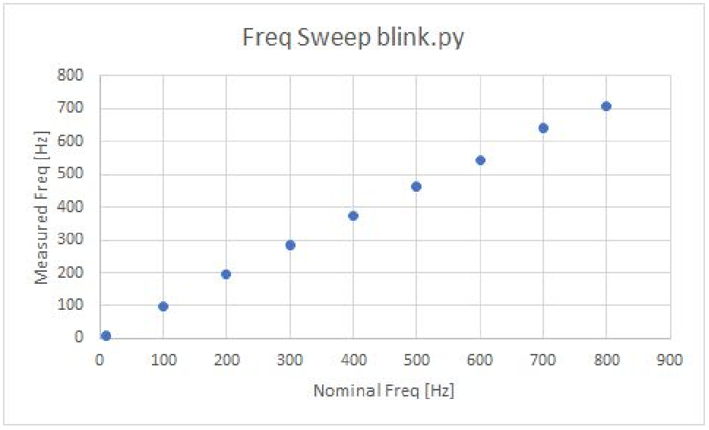


Figure 5: Python Frequency Sweep

The results from blink_v6.c are shown in Figure 6. We performed sweeps under both low and high CPU utilization. This required installing the C library called wiringPi to access the GPIO pins on the Pi from C code. For this script, we didn't bother writing in the button to increase frequency, so we manually wrote the frequency for each sweep point. Fortunately, this can be provided as an argument when running the code, so we don't have to recompile the C code every time. The maximum frequency that we were able to achieve within 10% was about 50kHz under low load and about 33kHz under high load. C has a lot less overhead than Python, which is why the timing is so much more accurate. At these speeds, the scheduler starts to have an effect, which is why the high CPU load reduces accuracy. The scheduler has to switch away from the script more often, so the timing is less accurate. Since we are using wiringPi library, it is important to note that delays under 100 microseconds (>10kHz) are timed using a hard-coded loop continually polling the system time, thus the system is under load will experience much reduced delay accuracy as seen in Figure 6 at 50 KHz. Delays over 100 microseconds (<10kHz) are done using the system nanosleep() function which uses an interrupt handler. Due to this, the accuracy of delays should not be affected if the CPU is under load or not. This can be observed in data points under 10 kHz in Figure 6.

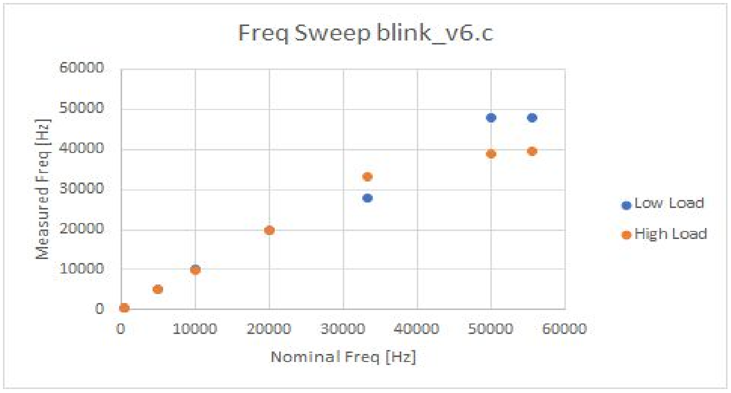


Figure 6: C Frequency Sweeps Under High and Low CPU Load

Part 2

The second step of this lab was to reinstall the kernel on the Raspberry Pi with the PreemptRT patch to allow for real time operation. There were three outlined methods of going about it: begin with a clean install of Rasbpian and compile configure and compile on the Rpi; begin with a copy of the current ('Lab 3') version of Raspbian and compile the modified kernel (4.14.91 with PreemptRT patch) on the Rpi, Cross compile the PreemptRT kernel on a Linux server, then, move the compiled kernel elements to the RPi SD card. We decided to go with the second option for the benefit of preserving the configurations done from Lab 1-3.

We initially started with Backing up our Kernel at its current state. Next, in order to see if we were constrained by memory in our SD card, we ran the **`df -h`** command. Although we were not constrained by space, we decided to run all the commands to remove applications like Wolfram and Supercollider. We also turned off the memory swapping from the Rpi BIOS. Next, we downloaded the Linux source files from github that took about 45 minutes to complete since we were using Wifi connection. To make sure we had available disk space for a kernel compilation we ran the **`df -h`** one more time. The figure shows that we had about 7.9 GB of free space available, which we assessed as plenty of space for what we're about to do

Next we downloaded the PreemptRT patch using the wget command from the provided link. Once complete, we made sure that all the dependencies for the kernel were installed. To do so, we navigated to the source folder of the Linux files, and ran **`sudo apt-get install bc`**. Next step was to actually compile the kernel with the PreemptRT patch which was going to take about 90 minutes. We backed up our SD card to ensure that we had a safe point to come back to.

Before we began compiling, the directions specified us to set up an environment variable called **`KERNEL=kernel7`**. Then we opened two terminal windows: one to run the compilation from the source folder; other to monitor the progress. On the first terminal we ran the following command:

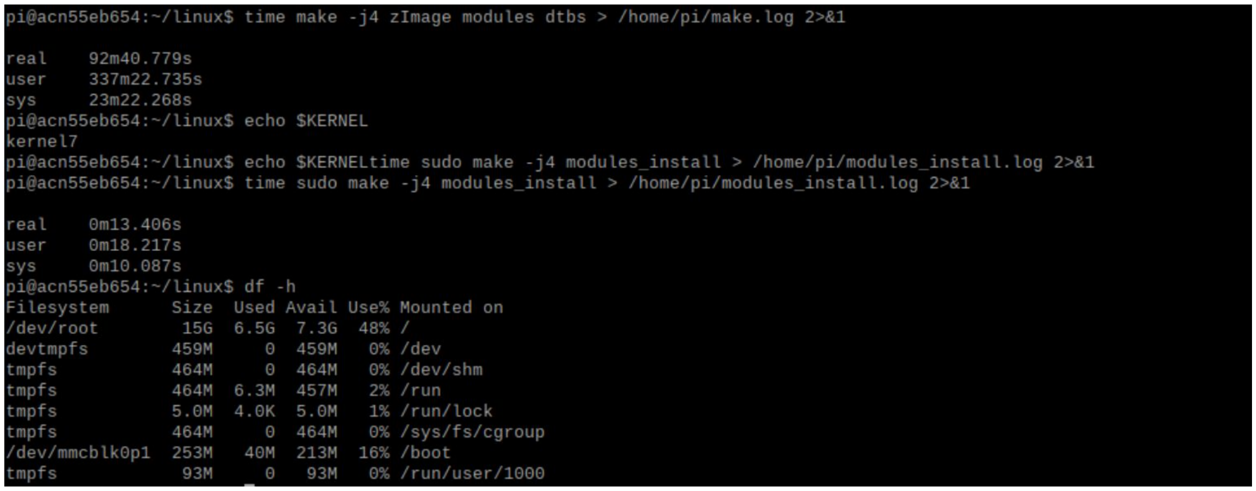
```
time make -j4 zImage modules dtbs > /home/pi/make.log 2>&1
```

The `time` command measures the compilation time. The `make` command compikes the zImage modules dtbs files and creates the corresponding ELF files. The **`-j4`** argument informs the make command to start 4 tasks which should use all four processors on the R-Pi. This is to speed up our compilation while taking advantage of parallelism. We wanted to log the output from the make command. Addition of **`2>&1`** allows us to redirect the stderr outputs into stdout and eventually into the file specified by the path **`/home/pi/make.log`**. Since outputs were concatenated to the end of **`make.log`** file, we used **`tail -f make.log`** to read the last lines of the log file.

Once complete, we ran the make command on the modules_install file:

```
time sudo make -j4 modules_install>/home/pi/modules_install.log 2>&1
```

The combined compilation took about 92 minutes. We also ran the **`df -h`** command to see the amount of space used by the compiled kernel. Compiled kernel was about 0.6 GB in size. Figure 7 shows the output of compilation.



```
pi@acn55eb654:~/linux$ time make -j4 zImage modules dtbs > /home/pi/make.log 2>&1

real    92m40.779s
user    337m22.735s
sys     23m22.268s
pi@acn55eb654:~/linux$ echo $KERNEL
kernel7
pi@acn55eb654:~/linux$ echo $KERNELtime sudo make -j4 modules_install > /home/pi/modules_install.log 2>&1
pi@acn55eb654:~/linux$ time sudo make -j4 modules_install > /home/pi/modules_install.log 2>&1

real    0m13.406s
user    0m18.217s
sys     0m10.087s
pi@acn55eb654:~/linux$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  6.5G  7.3G  48% /
devtmpfs        459M     0  459M   0% /dev
tmpfs           464M     0  464M   0% /dev/shm
tmpfs           464M  6.3M  457M   2% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           464M     0  464M   0% /sys/fs/cgroup
/dev/mmcblk0p1  253M   40M  213M  16% /boot
tmpfs            93M     0   93M   0% /run/user/1000
```

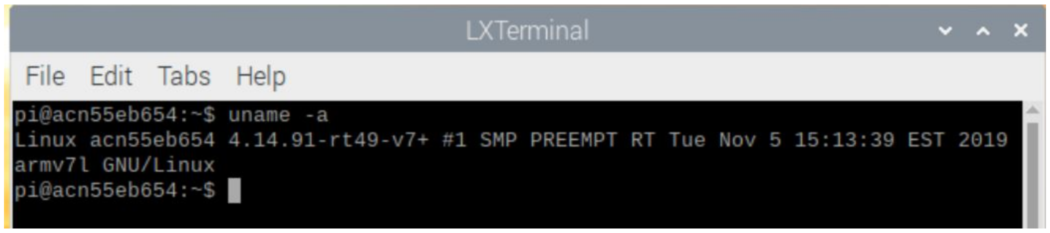
Figure 7. Compiling the kernel

Once we had confirmed that the **`make.log`** and **`modules_install.log`** files had matched up with the ones provided to us, meaning they had finished without any errors, we were confident that compilation was successful. We backed up our SD card at this point. Last steps included simply copying compiled kernel binaries to the **`/boot/`** folder. Before doing so, we rebooted and checked the version of kernel using uname -a command to verify that we were still running the old version of the kernel.

Finally we ran the following commands below:

```
$ sudo cp arch/arm/boot/dts/*.dtb /boot/
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
$ sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
$ sudo scripts/mkknlimg arch/arm/boot/zImage /boot/$KERNEL.img
```

Note that the copy commands move compiled results from a variety of locations in the /linux directory to the /boot location of the current Linux kernel. The final command replaces the original kernel image with the newly compiled Preempt-RT kernel. In order to boot in the new kernel we had to reboot the RPi. Fortunately, we hadn't lost our RPi configurations and we didn't have to go through the setup as we did in Lab 1. Although not required, we sought it fit to backup the SD card here. Figure 8 shows the RPi booting in the new kernel.



```
pi@acn55eb654:~$ uname -a
Linux acn55eb654 4.14.91-rt49-v7+ #1 SMP PREEMPT RT Tue Nov 5 15:13:39 EST 2019
armv7l GNU/Linux
pi@acn55eb654:~$
```

Figure 8. Booting in PreemptRT Kernel

Part 3

The final part of this lab involved running all of the same tests we run on the initial Raspbian kernel on the new PreemptRT kernel. We started with the Cyclictest runs. The results of the low-load run are shown in Figure 9, and those of the high-load run are shown in Figure 10. In the low-load run, the first thing we noticed was the multiple peaks in the distribution. There seem to be large peaks for all four processes around bins 8, 14, and 17, with smaller peaks around bins 34 and 45. This is likely due to the fact that there are multiple, distinct priority levels for the background running processes. When tasks with higher priority than 90 arrive, they must be dealt with before running the Cyclictest thread, leading to a new peak. The fact that the curve is narrower than that for the Raspbian run (Figure 1) demonstrates the fact that a real-time scheduler is better at quickly scheduling relatively high-priority tasks such as this and allowing them to execute quickly before context switching. The same narrowing can be seen in the high-load run compared to the Rasbian run (Figure 2). The multiple peaks aren't seen, however, likely because the high CPU load locks the scheduler in to a set schedule for the complete run. So these processes are almost always finished in the same amount of time.

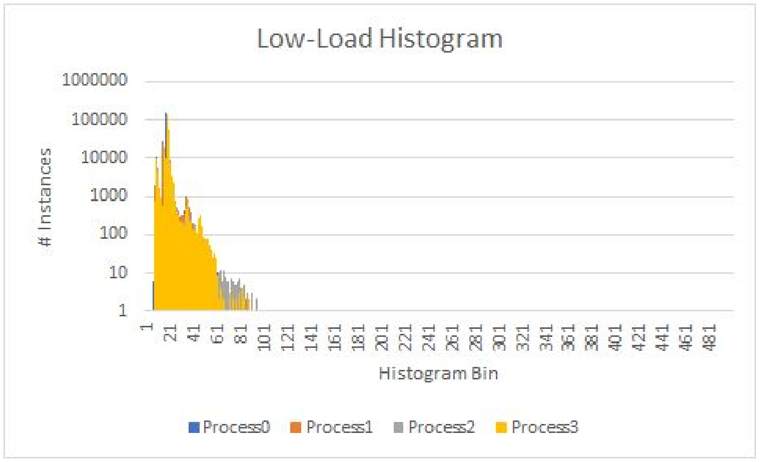


Figure 9: PreemptRT Low Load Histogram

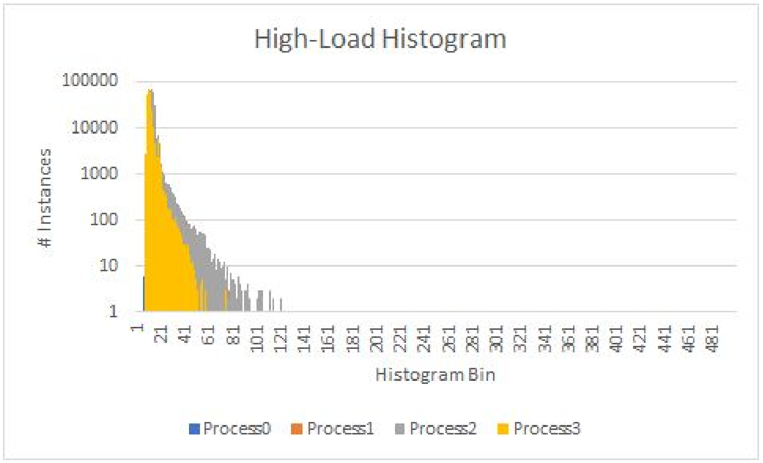
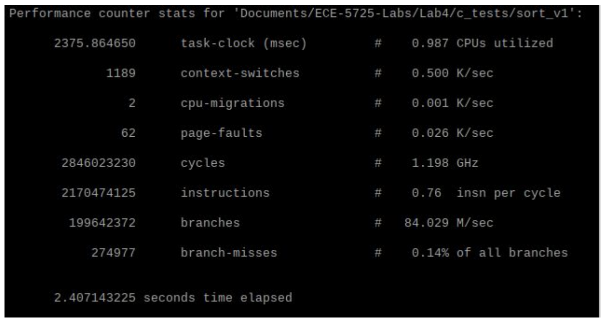


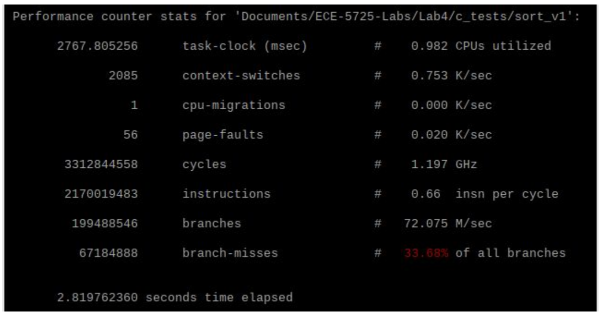
Figure 10: Preempt RT High Load Histogram

Next we re-ran Perf on the sort_v1 script. We noticed a slight improvement in performance, but it wasn't significant. As before, the sorted script ran slightly faster, and its branch-misses were much lower. Aside from that there were no notable differences. The results are shown in Figure 11.



```
Performance counter stats for 'Documents/ECE-5725-Labs/Lab4/c_tests/sort_v1':

      2375.864650      task-clock (msec)         #    0.987 CPUs utilized

             1189      context-switches          #    0.500 K/sec

                2      cpu-migrations            #    0.001 K/sec

               62      page-faults               #    0.026 K/sec

       2846023230      cycles                    #    1.198 GHz

       2170474125      instructions              #    0.76  insn per cycle

        199642372      branches                  #   84.029 M/sec

           274977      branch-misses             #    0.14% of all branches


      2.407143225 seconds time elapsed
```

Sorted

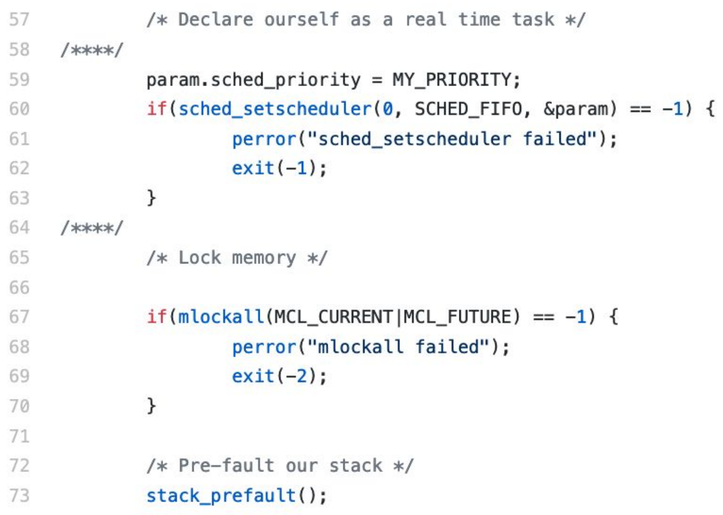


```
Performance counter stats for 'Documents/ECE-5725-Labs/Lab4/c_tests/sort_v1':

      2767.805256      task-clock (msec)         #    0.982 CPUs utilized

             2085      context-switches          #    0.753 K/sec

                1      cpu-migrations            #    0.000 K/sec

               56      page-faults               #    0.020 K/sec

       3312844558      cycles                    #    1.197 GHz

       2170019483      instructions              #    0.66  insn per cycle

        199488546      branches                  #   72.075 M/sec

         67184888      branch-misses             #   33.68% of all branches


      2.819762360 seconds time elapsed
```

Unsorted

Figure 11: Raspbian Perf sort_v1.c Results

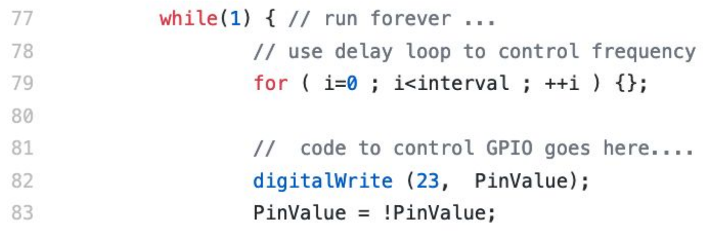
Finally, we executed frequency sweeps of square waves on a GPIO pin. Before doing any type of frequency sweep, we had to use modified version of the blinky code to allow for priority scheduling. The modifications were comprised of several pieces. First, we declared some constants such as priority, stack size. Next, we assigned a scheduling policy to the FIFO scheduler that will essentially allow our processes to hog the CPU. Lastly, in order to fully make this process a real-time process, we locked the memory to avoid VM page faults and ‘prefault’ the stack, which sets the entire stack for this process to zero, so that a future random-access of a stack variable will not cause a stack fault to add additional delays. Prefaulting will lock the process's virtual address space into RAM, preventing the section memory that will be accessed by the process from getting swapped. Figure 12 displays the section of code that accomplished this task.



```
        /* Declare ourself as a real time task */
/****/
        param.sched_priority = MY_PRIORITY;
        if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
                perror("sched_setscheduler failed");
                exit(-1);
        }
/****/
        /* Lock memory */

        if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
                perror("mlockall failed");
                exit(-2);
        }

        /* Pre-fault our stack */
        stack_prefault();
```

Figure 12. Modification to the blinky_v6 code

The final modification is the code structure that generates the square wave. This code uses a for loop as a delay mechanism rather than nanosleep. The for loop keeps the code ‘resident’ when running using sched_FIFO in the RT kernel. Figure 13 shows the loop logic where we are simply counting until the interval value is reached then toggling the GPIO pin. Theoretically, if our CPU is able to reach IPC of 1 and 100 instructions per iteration (worst case interval=50) , then we can toggle the GPIO at the frequency of at least 10 MHz.



```
        while(1) { // run forever ...
                // use delay loop to control frequency
                for ( i=0 ; i<interval ; ++i ) {};

                //  code to control GPIO goes here....
                digitalWrite (23,  PinValue);
                PinValue = !PinValue;
```

Figure 13. Square Wave Generation Logic

We initially started with large intervals, and slowly reduced it. In the previous kernel we were able to reach a knee at about 33 kHz. However in the realtime verizon, we could not compare nominal frequency to output frequency, but rather we had to compare some arbitrary value called interval to the output frequency. For large intervals, the nominal and the output frequency were inversely proportional, but however for interval values less than 50, we discovered a “knee” in the graph. This can be seen in the log-log graph depicted in Figure 15. Interval value 50 corresponded to an output frequency of ~7 MHz. This is much larger frequency value compared to our previous 33 kHz. In the previous kernel our output frequency capped off at 50 kHz but with RT kernel, we were able to achieve max frequency of 9.3 MHz. This is by far much more accurate and capable signal generator script relative to the previous versions. Figure 14-15 shows the interval sweep of RT kernel with different scales. Also, since our program is occupying one of the 4 processors at all times, all the incoming processes were simply scheduled for the remaining three processors. This meant that loading the CPU did not affect the performance of our prioritized process.

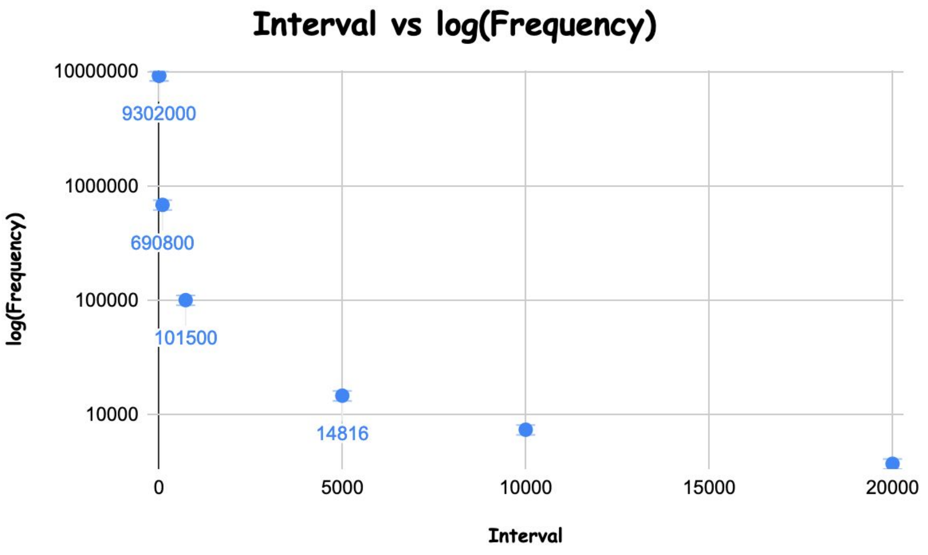


Figure 14. Interval Sweep with Linear X-scale

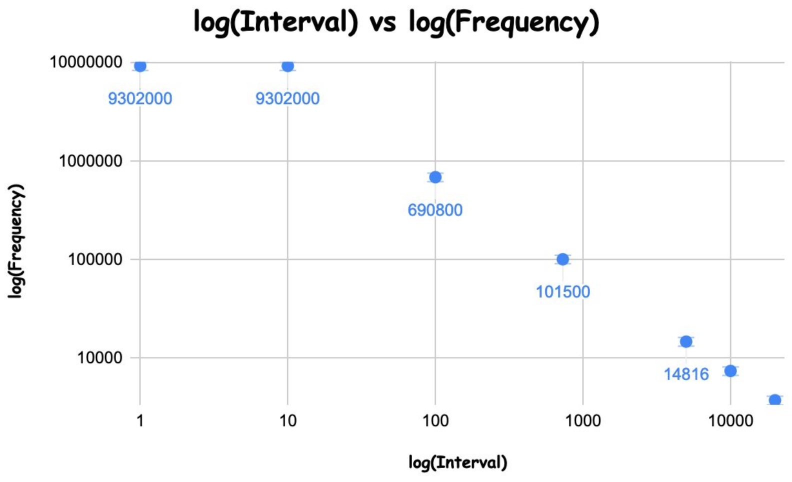


Figure 14. Interval Sweep with Logarithmic X-scale

Conclusion

In conclusion, we compared the performance of real-time and non-real-time embedded operating systems. To do this, we performed a number of tests both on the normal Raspbian kernel we've been using all semester and Raspbian 4.14.71 with the PreemptRT patch, which allows for fifo_scheduling with predefined priority. Using this ability and other attempts to minimize the response time for processes, we scripted tasks that enables the operating system to act as real-time processor. Using these tools we were able to achieve much more responsive processes, and much more accurate delays.

Code

Code stored on server at /home/Lab4/acn55_eb654_Lab4
http://wiringpi.com/reference/timing/