Lab 4
Lab Section: Monday

November 08, 2020

Jonathan Nusantara, Jiaxuan Su
jan265, js3596

Introduction

In this lab, we will be exploring and learning about Preempt RT kernel patch. In order to best understand the improvements made by Preempt RT kernel patch, we will first perform some experiments with the current installed Raspbian. These experiments involve running python and C signal-generation programs and recording the performance statistics with tools like cyclictest, perf, and piscope. Then, the new kernel will be built with Raspbian 5.4.72 and the PreemptRT patch 5.4.70. The same experiments using signal-generating programs will be performed to see the improvements in performance.

Design and Testing

Week 1:

Quick preparation before starting the lab:

This lab involves replacing the kernel that is on the SD card which will erase all data that is stored there. As a result, the first thing to do is to perform a backup of the SD card. Then, the files from the previous labs are uploaded to a cloud storage like Google Drive. Finally, since this lab is a bit more complicated, we had first read the required readings for the week to gain more insights about what will be performed during the lab. The readings are available in the Canvas page of the ECE 5725 course.

Performance measurement using cyclictest:

The first thing that we would want to measure is the latency time. Latency is time duration or delay between when the task is available/has arrived to be executed and when the task actually starts to be executed. Unfortunately, we could not measure latency using the perf tool. A tool that we can use is cyclictest, which would first need to be downloaded and installed to the Pi. It is loaded from GitHub. Below are the things that were performed:

```
git clone
git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests
.git cd rt-tests
make all
sudo make install
sudo cyclictest --help
```

With the above commands, the cyclictest will be fully installed. In terminal, "*`cyclictest --help`*" was run to view how to use the cyclictest tool and the meaning of the different flags.

Next, this tool is tested to measure the latency of the system. This is done by running:

```
time sudo cyclictest -p 90 -n –m -t4 -l 10000
```

Notice the different flags used above after “cyclictest”. This means that the command above will run threads at priority of 90, will use the nanosleep timer (-n), will lock memory (-m), and will run 4 threads (-t4) for 10000 loops. The result of the cyclictest is as below:

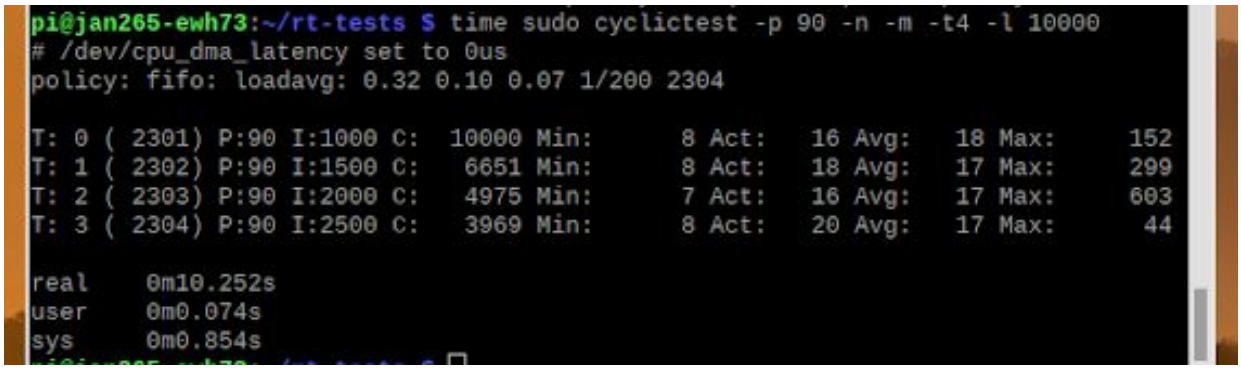

Figure 1 - Cyclictest result with loop of 10000 with unloaded Pi.

From the result above, we can see that 4 threads are used. For each thread, we can see the minimum, average, and maximum latencies. The test runs for 10.252 seconds. We can see that the maximum latency was 603ms in thread 2. All of the threads have an average latency at around 17-18ms. The term “unloaded Pi” means that there is no other program that is run when performing this cyclictest measurement.

Now, we will explore what may happen if we test it with 300000 loops. This time we will record the result to 500 histogram bins. The command below is executed:

```
time sudo cyclictest -p 90 -n –m -h 500 -t4 -l 300000
```

The result is as below:

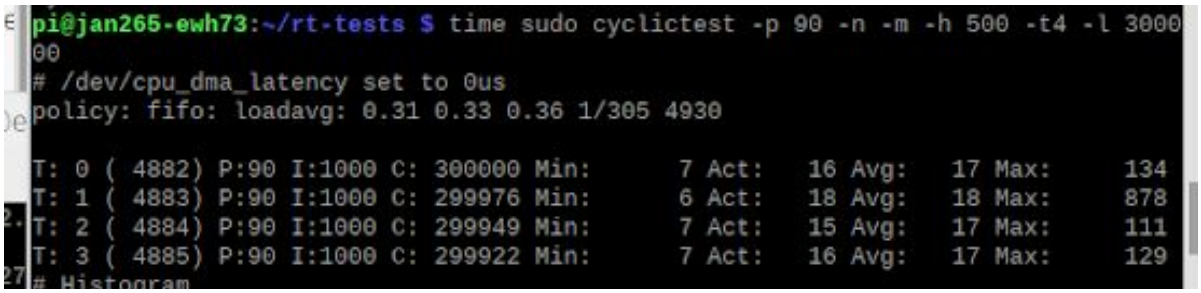

Figure 2 - Top cyclictest result with loop of 300000 with unloaded Pi.

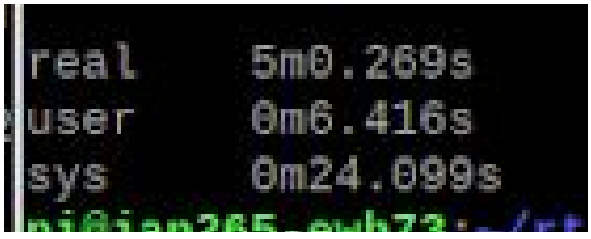

Figure 3 - Bottom cyclictest result with loop of 300000 with unloaded Pi.

From Figure 2 and 3, we can see several changes compared to the previous command run. It now takes around 5 minutes to execute as we now have a loop of 300000. Additionally, although the min and average values of the latencies of the 4 threads stays around the same value, we can see an increase in the maximum latency value from 603 to 878. This means that there is now a longer maximum latency. In order to have a better view of this, a histogram is plotted based on the result recorded:

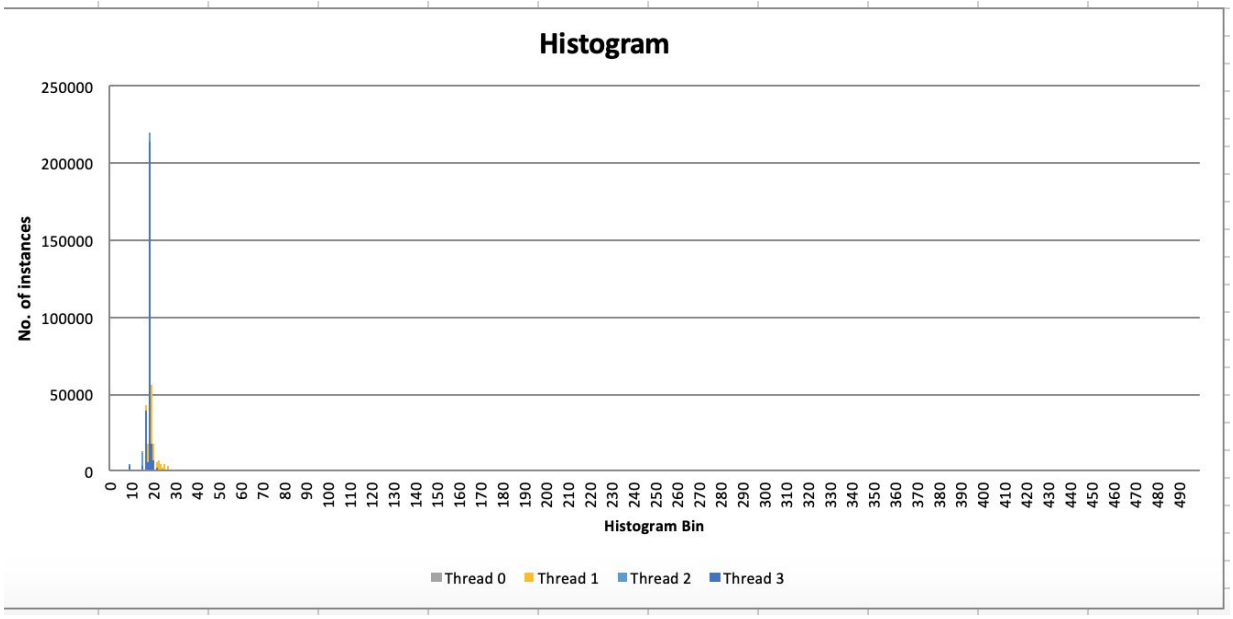

Figure 4 - Histogram bin of unloaded cyclictest.

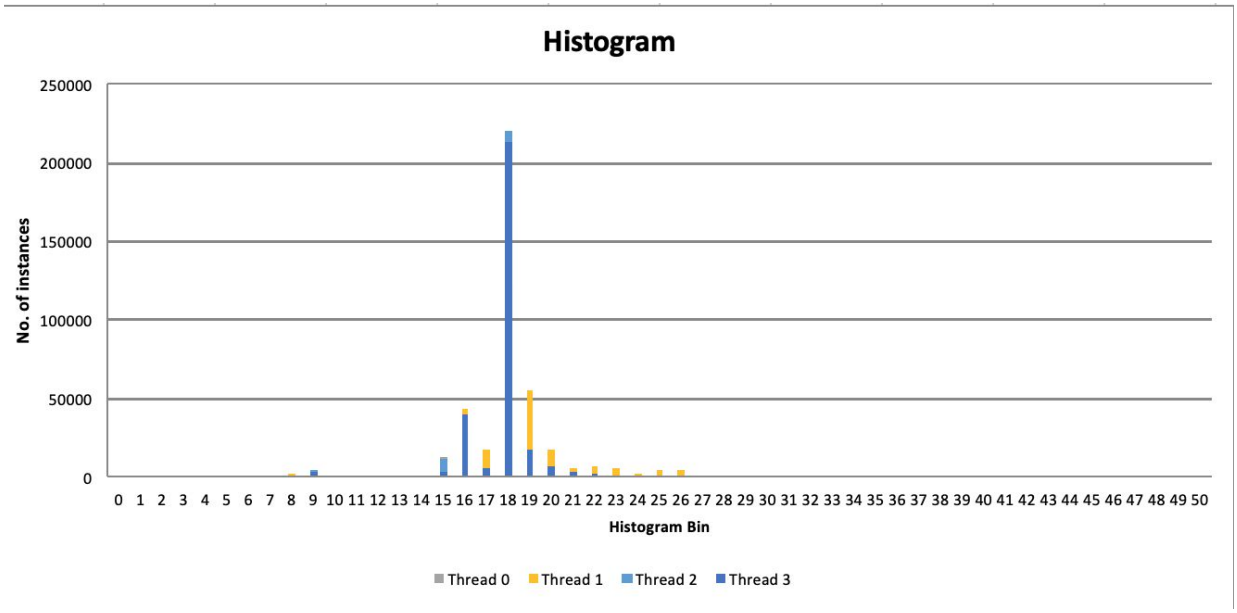

Figure 5 - Histogram bin of unloaded cyclictest up to bin 50.

From Figure 4, when the graph is plotted for all 500 histogram bins, it is difficult to get any information because there are a lot of empty bins. It is again plotted with bins 0 to 50 in Figure 5. Each histogram bin corresponds to a latency value in ms. We can see that out of the 300000 loops executed, the latency value tends to be in the latency of 18ms, which is the latency of around 220000 occurrences (more than half of total occurrences). The next most occurrences are in the bins next to it, between bin 16ms to 20ms. From the plot, we can see that there is a very low variance in maximum latency. This shows that with the low variance, the amount of latency is quite predictable. In order to further visualize the variance, a plot with a logarithmic scale is created:

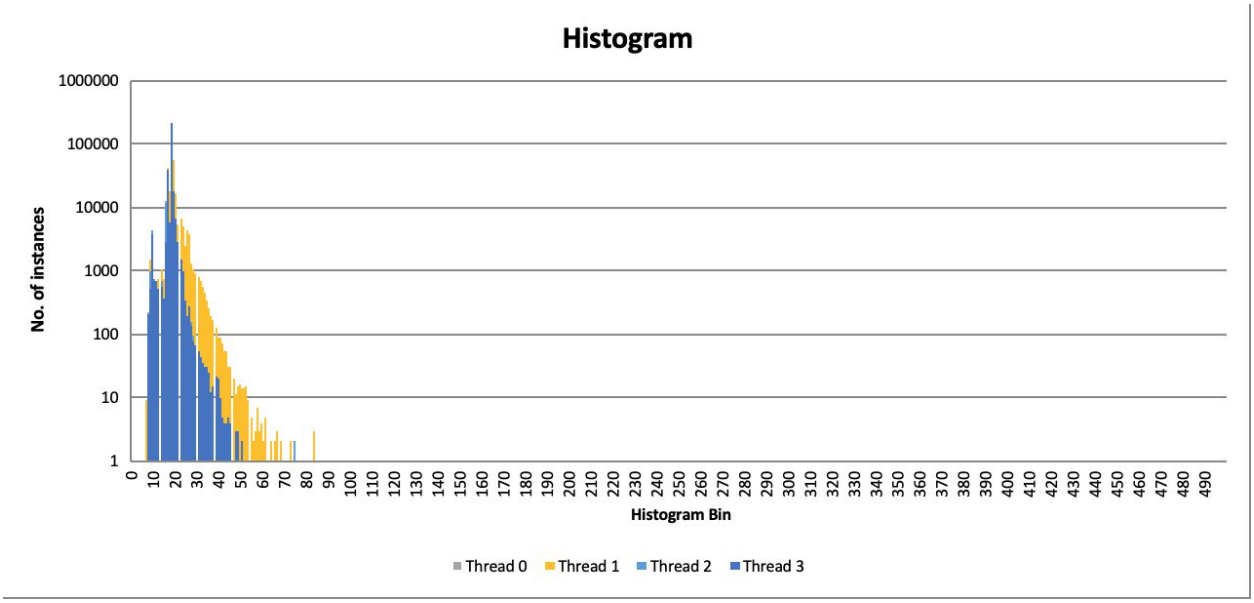

Figure 6 - Histogram bin of unloaced cyclictest with log scale.

The log scale provides a better visualization as it will scale up the lower values.
We will use the graph above to compare the variance with the other plots (when Pi is loaded or when using Preempt RT patch).

Next, we will try to run the cyclictest when the Pi is loaded, which means that a cpu-intensive task is run in the background that will use 100% of the CPU. A program called sort_v1.c is available in this directory:
`/home/jfs9/lab4_files_f20/c_tests/sort_v1.c`
What this program does is that it will create an array of random numbers and will traverse each index to see if the value is bigger than 128. The content of it is not important, but we know that this will use 100% of the CPU. In order to make this program run for a long time, it is modified so that it will loop around for "`20000000000`" times, which will be a very long time (way more than 5 minutes).
A snippet of the "sort_v1.c" program is as below:

```
for (unsigned i = 0; i < 20000000000; ++i)
```

```
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)   //  add to sum only if element > 128
                sum += data[c];
        }
    }
```

The c code is first compiled with the command below:

```
gcc –std=c99 –o sort_v1 sort_v1.c
```

Then, an executable is created "./sort_v1". So this time, we execute ./sort_v1 in one terminal window then run the command below in another terminal window:

```
time sudo cyclictest -p 90 -n –m -h 500 -t4 -l 300000
```

The result is as below:

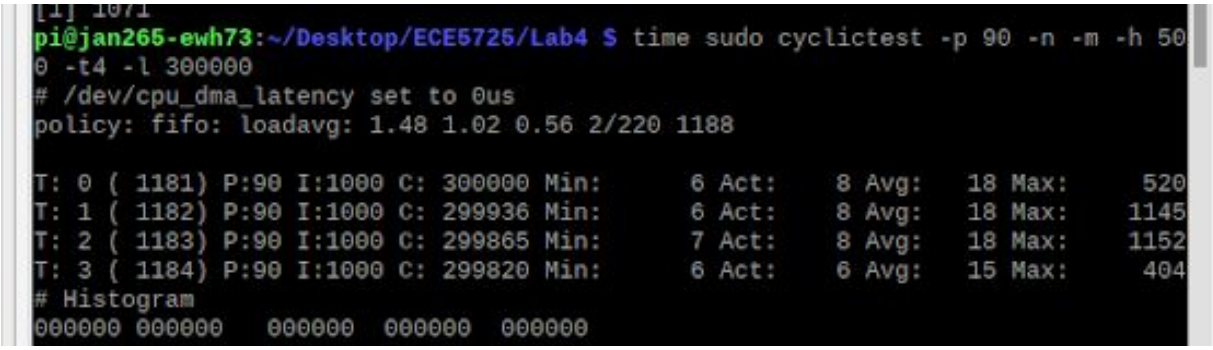

Figure 7 - Top cyclictest result with loop of 30000 with loaded Pi.

Comparing the result in Figure 7 to Figure 2, the most important data that we would like to compare is the maximum latency. We can see that the maximum latency is 1152 by thread 2, and the next one would be 1145 by thread 1. This is more than the value 878 that is recorded when Pi is unloaded. From this data and observation, we can see that when Pi is loaded, there will be an increase in maximum latency. This means that when we are running several programs together and when some of them are CPU-intensive, we have to be aware of how we may have a higher maximum latency, which will affect how executions and tasks are scheduled.

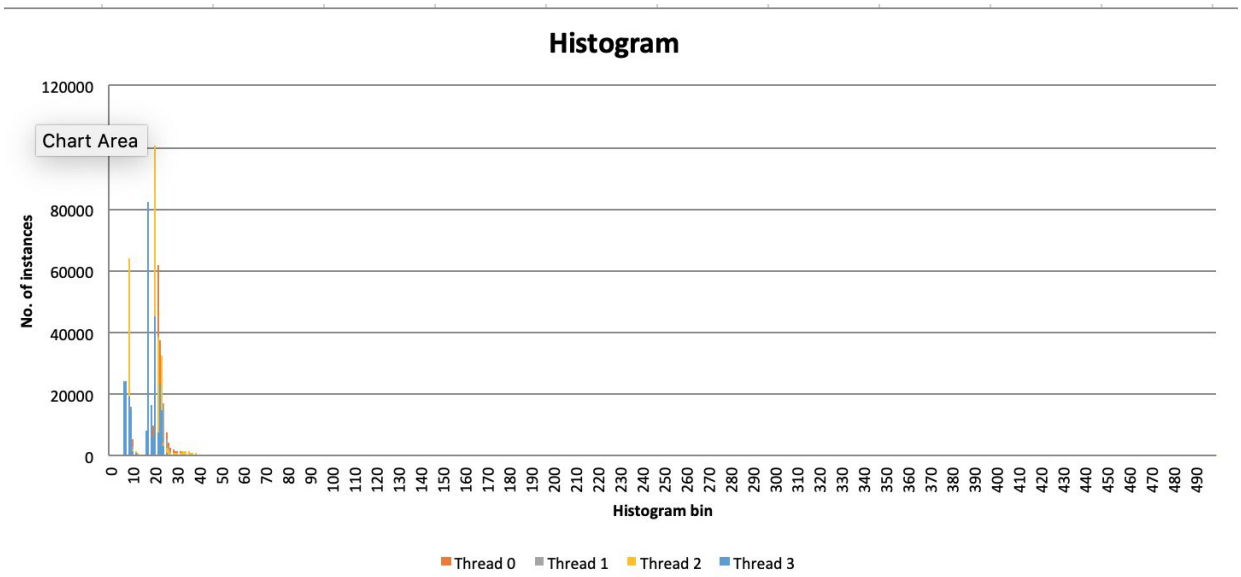

Figure 8 - Histogram bin of loaded cyclictest.

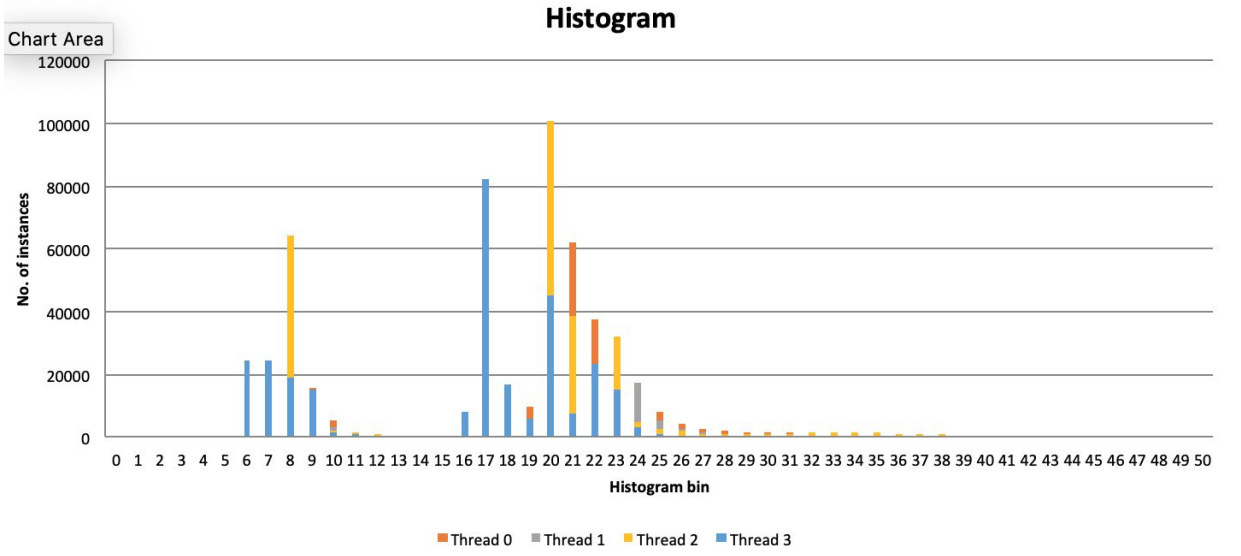

Figure 9 - Histogram bin of loaded cyclictest up to bin 50.

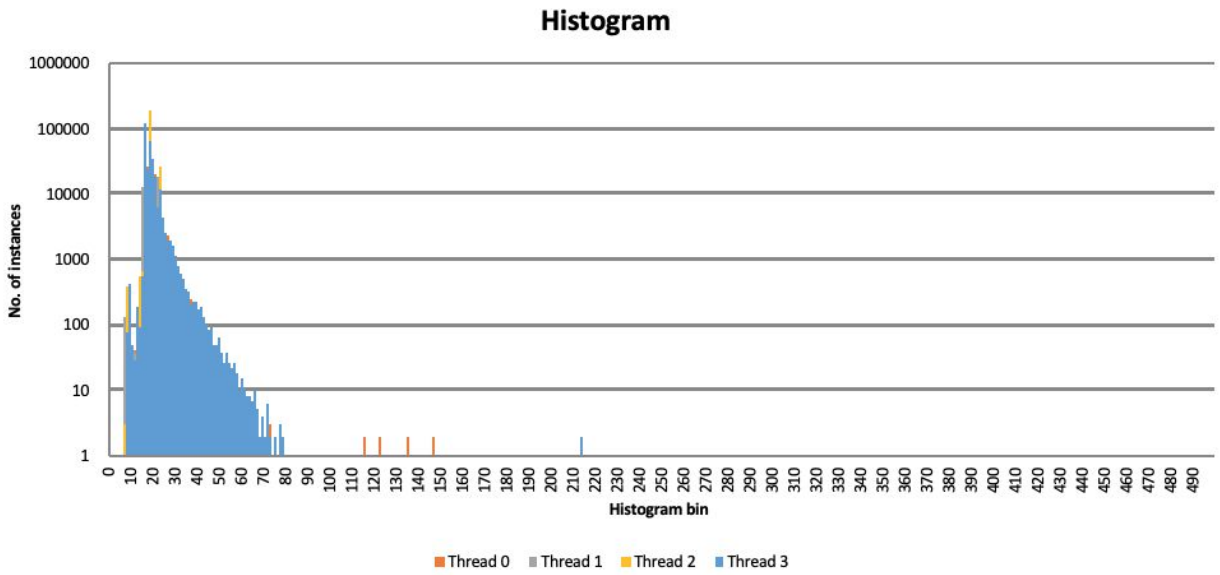

Figure 10 - Histogram bin of loaded cyclictest with log scale.

Similar to the unloaded Pi, the program ran for 5 minutes. From Figure 8, we can immediately see that there is a larger variance in the loaded Pi test, compared to the unloaded Pi test. We can see that most occurrences would have the latency of 20ms, which is quite close to the result in unloaded Pi. However, with the higher variance, the amount of latency is less predictable. We can now see latency as low as 6ms, which did not occur in unloaded Pi result. In Figure 10, we can also see how there are several occurrences where the latency is in the range of hundreds, which did not happen in Figure 6. As a result, this is something that we have to be aware of when working with a loaded Pi, as scheduling is affected by latencies. The amount of latency in a loaded Pi is less predictable.

During the loaded Pi test earlier, in order to make sure that the “sort_v1.c” program is using 100% of the CPU, it is being analyzed in “htop”:

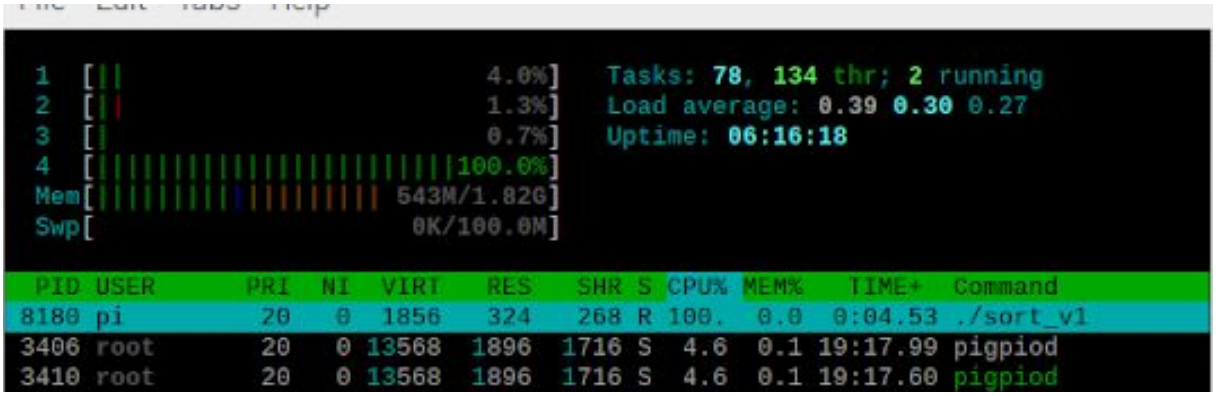

Figure 11 - Result of htop when running cpu-intensive task.

From the figure above, we can see on the line highlighted blue that the CPU is used at 100%. This means that the “sort_v1.c” program is running the way we expect it to. If the CPU usage is to decrease by a lot, this may mean that the program has ended and we will have to re-run that program.

From this section, we have learned how running a cpu-intensive task may affect the latency for other programs that are running. It will increase the maximum latency that can happen, which is something that we should be aware of. We also learned how the variance of the plot is higher when running loaded Pi. We have recorded important measurements and will be compared to when we are using a Preempt RT kernel. This section is pretty straightforward and we did not face any major challenges.

Performance measurement using perf:
Next, we will be testing the performance of the “sort_v1.c” program by using the perf tool. The perf command below is then executed:

```
sudo perf_4.9 stat ./sort_v1
```

This will display the recorded statistics as below:

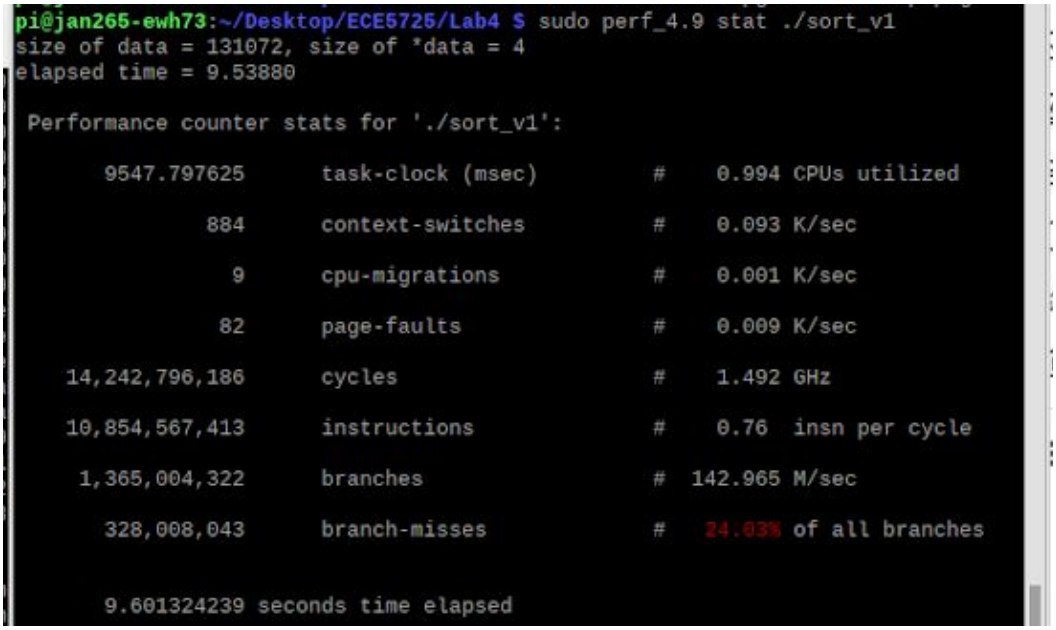

Figure 12 - Performance statistics of sort_v1.c without quick sort.

Then, the command is again run, but this time a sort is first being done before entering the main loop that will sum the index in the array that has a value more than 128. This can be done by uncommenting this line:

```
qsort(data, sizeof(data)/sizeof(*data), sizeof(*data), comp);
```

What this line does is that it will sort the array in numerical order. The program is again ran and the statistics is as below:

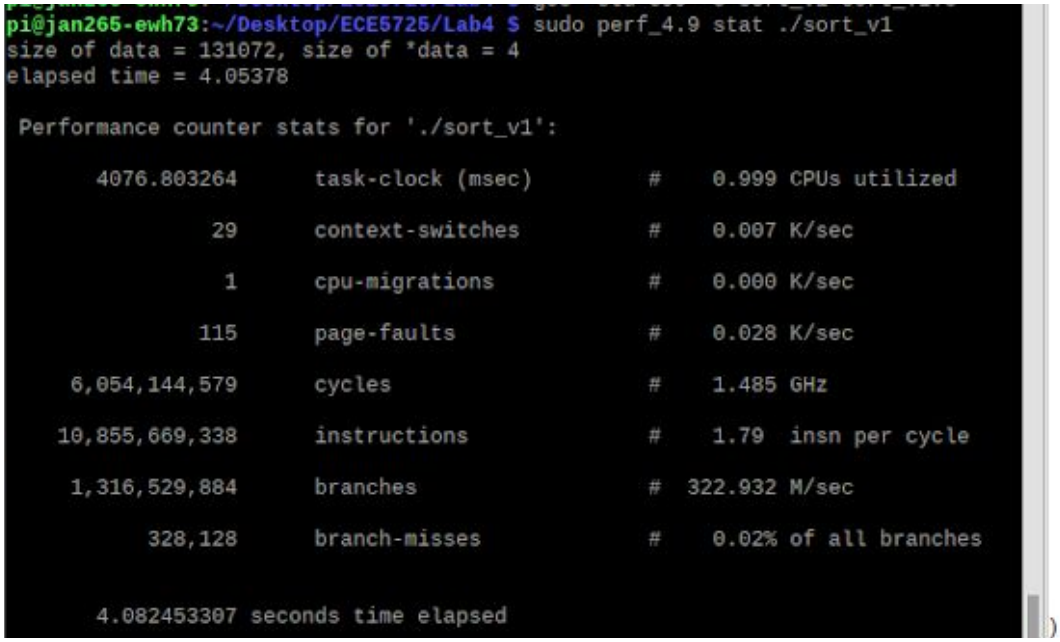

Figure 13 - Performance statistics of sort_v1.c with quick sort.

Comparing the two results above, we can see that the program is more efficient and runs faster when quick sort is first done. We can see that when sorting is done, the task clock will be less by half, the number of context-switching is way less, there is only 1 cpu-migration, more than half of the amount of cycles, and significantly less branch misses. Due to this improvement in efficiency, the time required to run the program is cut by half.

From this test, we learned that it is better to perform the sorting first as it helps in the efficiency of the program, which is proven by running the perf tool. Now here is a little discussion on why the program ran faster, despite adding an extra step of sorting the program. This happens because of the way linux has an instruction pipeline that includes Fetch, Decode, Execute, Memory, and Write, which goes step by step. In the Execute step, it will check if the number in the array index is bigger than 128 or not. The branch predictor will make a prediction on what the next step is. If it turns out to be wrong, the Fetch and Decode step that is previously executed would be for waste and this is called a branch miss.

So looking at the figures above, we can see that there were around 25% of branch misses when the array is not sorted, and this cost the program a lot of time. This is because since the array is unsorted, it is often the case that the branch predictor makes a wrong branch prediction and thus branch misses happened. However, after performing the sorting, what will happen is that for the early parts of the array, the predictor will have a consistently right prediction, because its prediction is actually based on the history of previous results. Since the previous results in the early parts of

the array are always less than 128, then it will predict that the current number is less than 128. Because of this, there is much less branch misses of 0.02%. This is why despite using an extra step of sorting, the program ended up to be faster because of the significantly less branch misses.

Overall, we did not face any difficulties in this section as it is straightforward. However, it takes a while for us to understand why this phenomena above has happened.

Performance measurement of square wave in python and C:
Next, we will check out the difference between a python and C when running a similar program. We have two similar programs in python and C that may be used to blink an LED in a circuit. This program has the capability to have the frequency to be adjusted. Since this program will generate a square wave, we will analyze the produced square wave in piscope. We will keep increasing the frequency to see what is the maximum frequency each program can produce accurately. His means that the difference of frequency between the value supplied in the program and the one recorded needs to be within 10%.

The connection that is used is from the previous Lab3, where we have a circuit that looks like this:

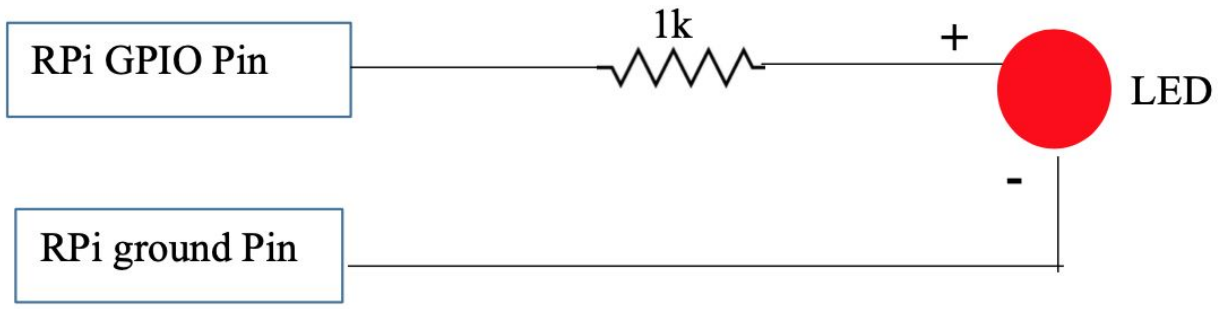

Figure 14 - Schematic diagram of LED connections to RPi.

We are using GPIO 26 to be connected to the LED. Now that the circuit is ready, we can run the program.

Firstly, we are testing using the python program. We are running the “blink.py” program that we created in the previous lab, in which it will run the PWM signal based on the frequency that we have set. Before starting the python program, in another window we will execute “sudo pigpiod” and “./piscope” in the piscope directory to activate the signal monitoring. Then, we will run the program. An example of the frequency measurement is shown below:

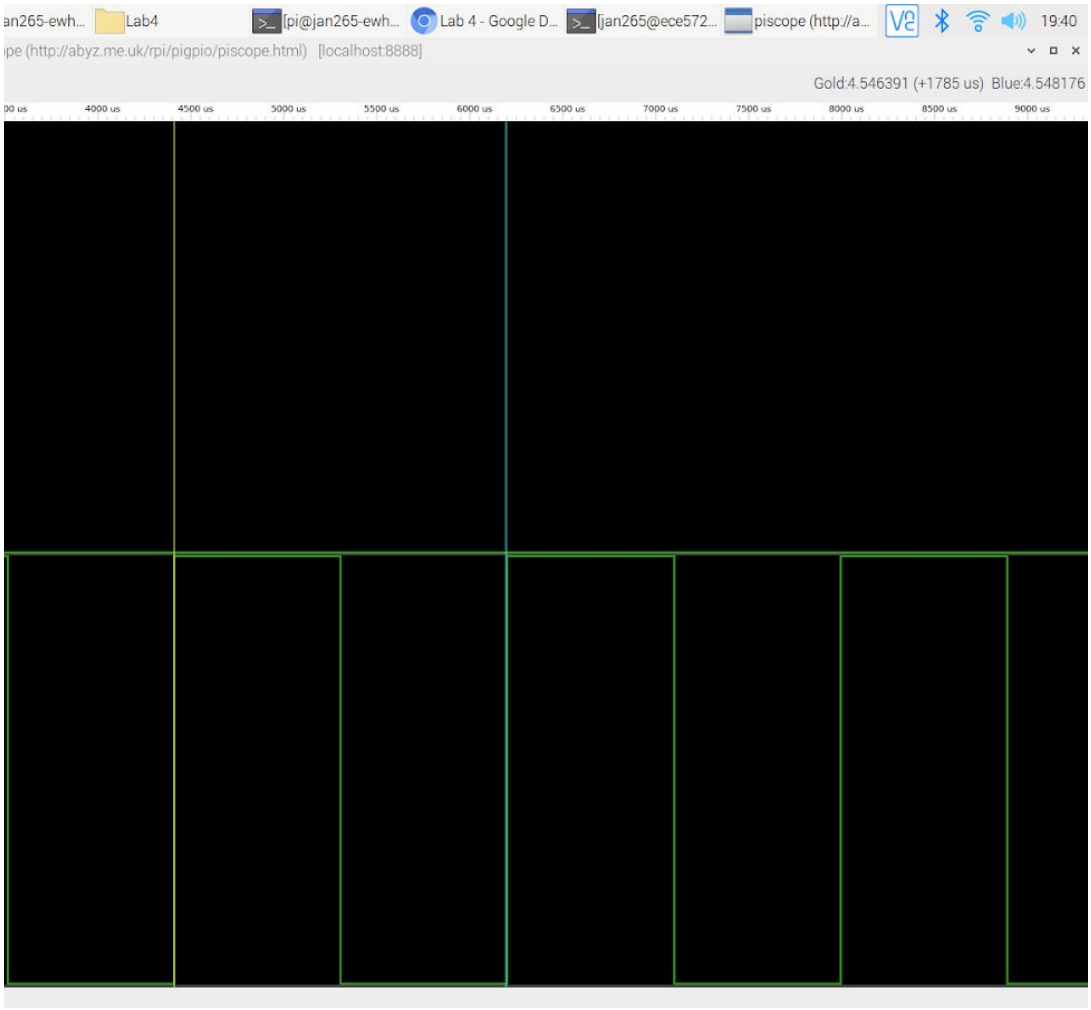

Figure 15 - PiScope result of python program at 600Hz.

In piscope, we can use the blue and yellow cursor to measure the period of a single signal. Then, we can see period measurement on the top right (measured as 1785 microseconds in this example). This translates to 560Hz. This measurement was taken when 600Hz is set in the program. Their difference is calculated to be around 6%, which is still under 10% so the signal produced is good. In this example, we carefully made measurements incrementally from 50Hz to 1000Hz, and at each point we will compare with the result in piscope. These results are recorded in an excel sheet. From this experiment, we have found that the maximum frequency that will still be accurate is 900Hz.

Next, we performed the same exact measurements, but this time with the “sort_v1.c” program running in the background. This is to test on what will happen when the Pi is loaded and that CPU is running at 100%. The cpu-intensive program is running the entire time we are making the frequency measurements. The same measurements are

performed and are recorded in an excel sheet. From this experiment, we have found that the maximum frequency that will still be accurate is 900Hz. This means that it has a similar result when Pi is not loaded and “sort_v1.c” is not running.

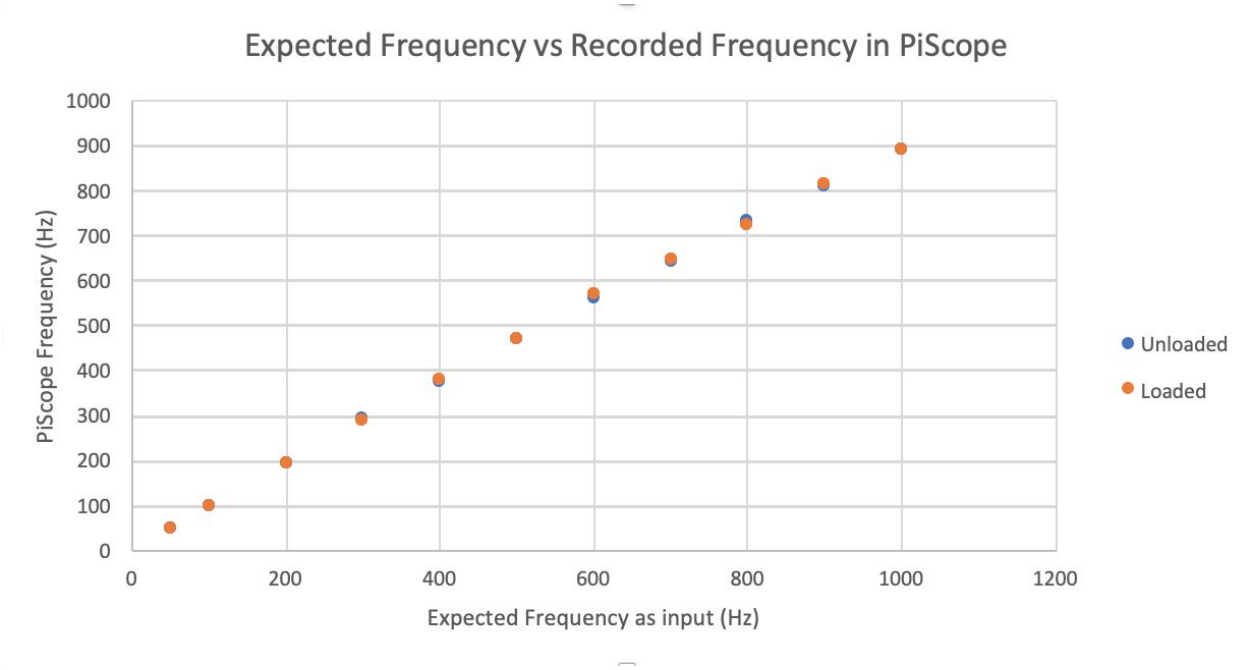

Figure 16 - Frequency plot of blink.py program.

From the plot above, we can see that at each frequency, both instances of the python program running (when Pi is loaded or unloaded) provides a similar performance. Not only does each point is almost at the same location, but they both have the same maximum frequency of 900Hz. At 1000Hz, the error between the frequency in the program and the actual frequency recorded in piscope is more than 10%.

Next, a similar experiment (both loaded and unloaded) is done for the blink program in c “blink_v7.c”. This c program is provided in the lab4 directory:

```
/home/jfs9/lab4_files_f20/
```

In this c program, we can also set the frequency that we would like to pass. Before executing the program, we would have to compile the program using the command below:

```
gcc -Wall -pthread -o blink_v7 blink_v7.c -lpigpio -lrt
```

Now that we have the executable ready, we can run the experiment. One thing to note that this c program works differently with the piscope program. It will not run like it normally would with a python program. We would first need to kill the pigpiod by running “sudo killall pigpiod” or else the C program will not run. Then, we first execute the c program “./blink_v7” and then start the piscope.

We performed the same measurements as we did for the python program. What we soon realized is that the c program can actually produce much higher frequency. We ended up testing the program by incrementing the frequency at 10000Hz for every stage. When Pi is unloaded, the maximum frequency based on our experiment is 50000Hz. The frequency recording at each stage is saved in our excel spreadsheet.

Then, for the last time, we performed this frequency measurement again but with the cpu-intensive program running in the background. This is to see if the maximum frequency will now decrease. The “sort_v1.c” is run during the entire experiment, which will make the CPU run at 100% as verified when running “htop”. The result that we have found is that the maximum frequency is now at 40000Hz.

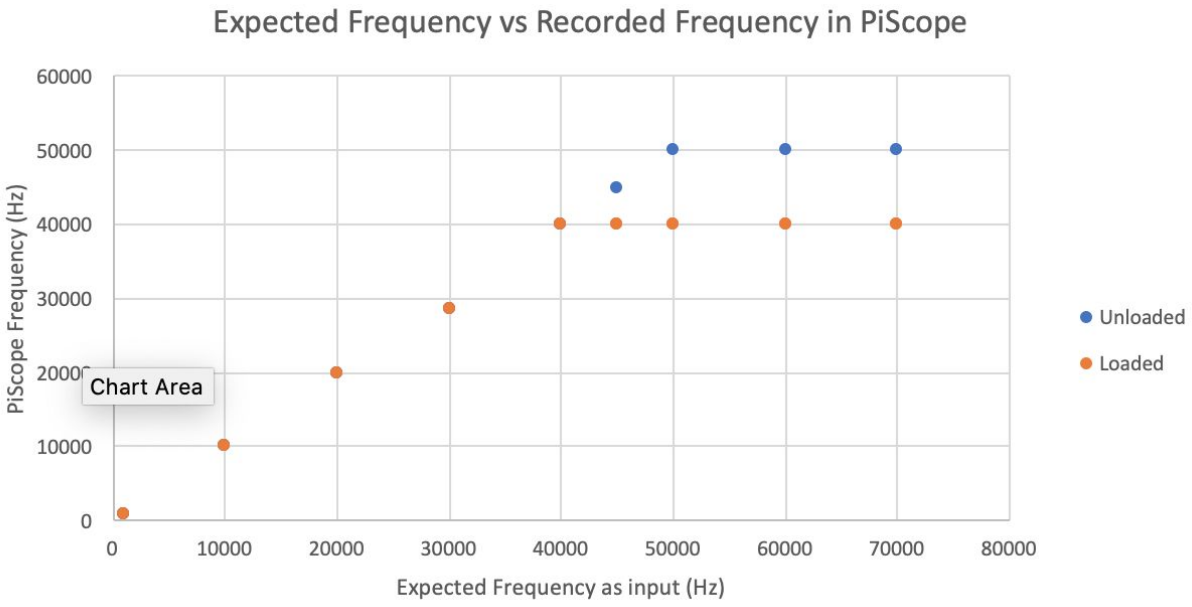

Figure 17 - Frequency plot of “blink_v7.c” program.

Looking at the figure above, we can see that the maximum frequency for loaded Pi is at 40000Hz. So although the program is set to output a higher frequency, the Pi just cannot do that. This is because we are running the CPU-intensive task in the background, which is already using 100% of the CPU. Another analysis that we have made is that the maximum frequency in C is much higher, which means that Python is much slower. This shows that Python just cannot run the program faster. This is because python is an interpreted language. This means that for each line in Python, there are much more CPU tasks and instructions that need to be done, compared to a single line in C code.

In this section, we found out about two major things. Firstly, python is much slower than C. So, when we want a program to run as quickly and efficiently as possible, using C will be preferred. Another thing is that running another program that is cpu-intensive may decrease the performance of the other programs running in Pi, as shown in the C-program experiment. Using this data, we will be comparing it with the results using Preempt RT kernel to see if there will be any improvements.

This section is pretty straightforward. However, the main challenge is that using piscope to measure frequency is very slow as we would need to use the cursor to measure the period, then convert it to frequency. Using an oscilloscope that can provide the frequency measurement will be much faster.

Building the Preempt RT Kernel:

In this section, we will be building and compiling the Preempt RT Kernel. In order to build a kernel, we need a Linux source file and the preempt RT patch file. These two files from two different repositories will be combined together to a new kernel. The Raspbian kernel that we will use is version 5.4.72 while the Preempt RT patch file is version 5.4.70-rt40. These two have been proven to work together, and as a result we will be using it. Unfortunately, this process involves a lot of different steps. Each of the steps will be discussed below, however, to give a brief overview, there are the list of things that need to be done:

1. Download the Linux source files
2. Download the PreemptRT patch
3. Merge the patch with the Linux source
4. Configure the Linux code
5. Compile the Linux kernel
6. Copy the kernel to the SD card and boot the R-Pi

First of all, we backed up our SD card to have the back-up from Lab3 and the first few sections of Lab4. Also, the ".bashrc" modifications from Lab3 are also removed as we no longer want to start the program on every reboot.

Next, we would have to free some space in the RPi SD card. This is because this compilation process needs a lot of memory. Below are the few things that we can do:

- Remove large files, such as videos or games. Removing unneeded files is also recommended (large or small)
- Run "`sudo apt-get remove --purge wolfram-engine`" to remove wolfram-engine

- Run "`sudo apt-get remove --purge supercollider*`" to remove the supercollider music processing software.
- Disabling swapping on in the kernel by running the following commands:
  ```
  sudo dphys-swapfile swapoff
  sudo dphys-swapfile uninstall
  sudo update-rc.d dphys-swapfile remove
  ```
  This will remove any "dead sectors" that Linux has saved in reference to the SD card, in which it refers to the sections in the SD card in which Linux was not able to access previously. This may help increase the available memory.
- Finally, run the following three commands:
  ```
  sudo apt-get autoclean
  sudo apt-get clean
  sudo apt-get autoremove
  ```

After cleaning the memory, we should check the current available memory by running "`df -h`" in Terminal.

```
pi@js3596:~ $ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  3.1G   11G  23% /
devtmpfs        805M     0  805M   0% /dev
tmpfs           934M     0  934M   0% /dev/shm
tmpfs           934M  8.5M  926M   1% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           934M     0  934M   0% /sys/fs/cgroup
/dev/mmcblk0p1  253M   55M  198M  22% /boot
tmpfs           187M     0  187M   0% /run/user/1000
pi@js3596:~ $
```

Figure 18 - The available memory after cleaning up some files.

Then, the next step is to download the Linux source from the github repository. First, we should go to the directory "/home/pi". Then, the command below is executed:

```
time git clone --single-branch --depth 1 --branch
'rpi-5.4.y'https://github.com/raspberrypi/linux.git
```

This download may take a while depending on the internet speed. If you are using a mobile hotspot, it unfortunately takes about one hour. In order to verify that we downloaded the right version, in the "/home/pi/linux" directory, we can run "head Makefile" and it will show that we have the version 5.4.72 (or 5.4.73) as in the following figure.

```
[pi@js3596:~/linux $ head Makefile
# SPDX-License-Identifier: GPL-2.0
VERSION = 5
PATCHLEVEL = 4
SUBLEVEL = 73
EXTRAVERSION =
NAME = Kleptomaniac Octopus

# *DOCUMENTATION*
# To see a list of typical targets execute "make help"
# More info can be located in ./README
```

Figure 19 - New Linux source version.

Now that we have downloaded the Linux source, we will confirm that we are not missing any dependencies. We ran run the command below:

```
$ sudo apt install bc bison flex
$ sudo apt install libncurses5-dev
$ sudo apt install libssl-dev –t buster
```

Next, we download the Preempt RT patch and put it directly in the Linux source directory. This is done by going to “/home/pi/linux” directory, then executing:

```
$ wget
https://www.kernel.org/pub/linux/kernel/projects/rt/5.4/patch-
5.4.70-rt40.patch.gz
```

This will put the downloaded files right in that directory. Then, we backup the SD card again. In this checkpoint, we will have a clean and working copy of the Linux source that we can come back to if needed. Then, now that we have confirmed the versions of the Linux source and the Preempt RT patch, we can merge them. This is done by executing the command below in the linux directory:

```
/$ zcat patch-5.4.70-rt40.patch.gz | patch -p1 >
/home/pi/patch.log 2>&1
```

There is an output log called “patch.log” in the “/home/pi” directory. We compared this output with another “patch.log” provided in Canvas to confirm that it is good and with no errors.

Next, we will configure the kernel. The following commands are executed in the linux directory:

```
$ make clean
$ make mrproper
```

What the commands above did is to clean up any previously compiled files in the linux directory. Next, we will rename our kernel to kernel7l (read as kernel-seven-ell). This is extremely important as we would not want to wrongly name our kernel. We can rename it and check if we successfully renamed it:

```
$ KERNEL=kernel7l
$ echo $KERNEL
```
Then, the following command is ran to make sure that we have the default configuration to set up the RPi. It will create several ".config" files that are used by the menuconfig command. The command is:
```
$ make bcm2711_defconfig
```
Then, we will make some additional configurations by running the following (still in linux directory):
```
make menuconfig
```
The command above will lead you to a configuration page, with instructions on top. There are two things that need to be configured. First, we should head to 'General Setup, then to 'preemption model', then we activate 'RT Fully Preemptable kernel'. Also, check if 'preemption model' exist in 'Kernel Features'. The second thing is to head to 'General Setup', then to 'Timers subsystem', then select 'High Resolution Timer Support'. Finally, we save the configurations with the default name.

Now that the Linux source and preempt RT patch has been merged, and we have made the required configurations, the kernel is ready to be compiled. Before that, we ran "`df -h`" to make sure that we have enough memory:

```
[pi@js3596:~/linux $ df -h                                                    ]
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  6.2G  7.5G  46% /
devtmpfs        805M     0  805M   0% /dev
tmpfs           934M     0  934M   0% /dev/shm
tmpfs           934M  8.5M  926M   1% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           934M     0  934M   0% /sys/fs/cgroup
/dev/mmcblk0p1  253M   55M  198M  22% /boot
tmpfs           187M  4.0K  187M   1% /run/user/1000
```

Figure 20 - The available memory right before compiling the kernel.

The compilation will take around 60 minutes. Since all of the cores in the Pi will be used, it will be best to not run anything else to prevent slowing down the RPi. At this point, we also make sure our Kernel name one last time.

We compiled the kernel by running "`$ time make -j4 zImage modules dtbs > /home/pi/make.log 2>&1`" in one terminal window in the Linux directory, which will perform the compilation itself using the "make" command. Notice that a "-j4" flag is used, which means that the compilation will be done in all of the 4 cores to make it run as fast as possible. In another terminal window, we ran "$ `tail -f make.log`", and it is used to keep track of the compilation progress. After the compilation completes, we compare our "make.log" with the Professor's "make.log" that is in the class folder: /home/jfs9/lab4_files_f20.

Once we have verified that everything is going well, we run the following in the linux directory:

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

This will perform the modules installation and run for several minutes. We made another comparison involving the “modules_install.log” that is in the pi directory with a similar file from Professor in the class folder.

At this step, this is the amount of available space that we have:

```
[pi@js3596:~/linux $ df -h                                              ]
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  6.9G  6.8G  51% /
devtmpfs        805M     0  805M   0% /dev
tmpfs           934M     0  934M   0% /dev/shm
tmpfs           934M  8.5M  926M   1% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           934M     0  934M   0% /sys/fs/cgroup
/dev/mmcblk0p1  253M   55M  198M  22% /boot
tmpfs           187M  4.0K  187M   1% /run/user/1000
```

Figure 21 - “df -h” execution result after compiling kernel.

Once we are done with no errors, this means that we now have a compiled preempt kernel in the SD card. However, it has not been installed yet and will be done in the next lab section. In this section, everything is pretty straightforward and we did not face any challenges. However, we had to be extremely careful because there is a huge impact we have made even the slightest mistakes. In this section, we basically learned about what are the different steps of compiling a kernel with our own version of a patch. We would need to make sure that we have enough memory, then we merge the Linux source and the patch, and then we will perform the compilation that will take quite a while.

Week 2:

System setup:

Continuing the work from last week, we have a Preempt RT kernel that has been compiled but has not been installed. Before continuing with the installation, we made sure that we have made a backup so that if something goes wrong we can restore this current checkpoint in which we have a compiled kernel ready to be installed again.

Next, we will proceed with installing the Preempt RT kernel. Firstly, we go to the “home/pi/linux” directory and pass the following command to set the kernel name:

```
$ KERNEL=kernel7l
$ echo $KERNEL
```

Then, we run “uname -a” to check the current kernel version. The output is as below:

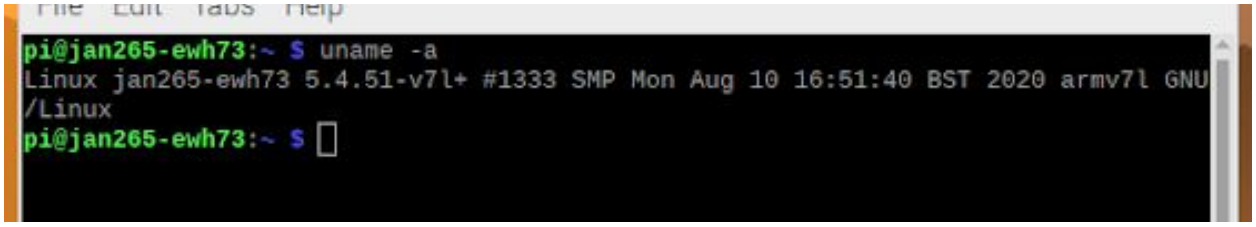

Figure 22 - Output of “uname -a”.

We can see that we have linux version 5.4.51 currently running. Now, we would like to upgrade it to the kernel version that we have just compiled. Before that, we run the following commands to put elements of the new kernel to the corresponding locations. These elements are the compiled results that exist in the “/linux” directory and that we would like to move to the “/boot” directory to be boot.

```
$  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 cp arch/arm/boot/zImage /boot/$KERNEL.img
```

Notice that in the last line, the command is on how the kernel image that we have just built will replace the original kernel image that we are currently using. Now, we can reboot the system.

Upon rebooting, our Pi and its SD card will now have the new kernel running and launched. There is a chance that the “raspi-config” of the Pi would need to be re-setup, although that is not the case for us. We made sure that our Pi is currently running the new version by running “uname -a” again and getting the following output:

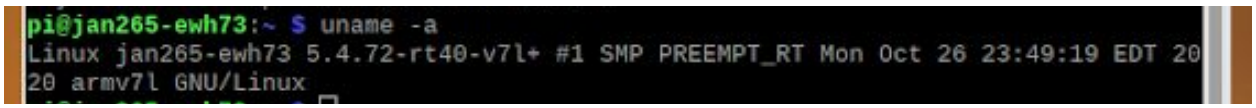

Figure 23 - Output of “uname -a” showing Preempt kernel.

Notice how it is running version 5.4.72 with a Preempt RT patch. This means that we have successfully installed this Preempt RT kernel. With this new kernel, there is a chance that we may have to reinstall several tools such as cyclictest or perf_4.18. This will be verified in the next section. With this Preempt RT kernel, there is an issue with

the USB driver that may cause RPi to freeze, so the following line needs to be added to the “/boot/cmdline.txt”:

```
dwc_otg.fig_enable=0 dwc_otg.fig_fsm_enable=0
dwc_otg.nak_holdoff=0
```

Note that everything is typed in a single line as this is extremely important. Since “cmdline.txt” is loaded upon reboot, we need to reboot our system again.

Now, we have a fully installed Preempt RT kernel that is ready to be tested for its performance. Before that, the SD card is backuped to make sure we have this copy where Preempt RT patched kernel has already been installed. This section of the lab is very straightforward, but we need to be careful and to perform backups at several steps.

Performance measurement using cyclictest:
In this lab, we will again perform cyclictest, in which we wil compare the results of the performances with the results using the previous non-PreemptRT kernel. Note that cyclictest may immediately not run as it may have not been installed due to “/usr/bin” being updated with the new kernel. Although after testing it, it is not the case for us. The following command (similar to last week) was ran:

```
sudo cyclictest -p 90 –m –n –t4 -l 10000
```

From the command above, we got the following:

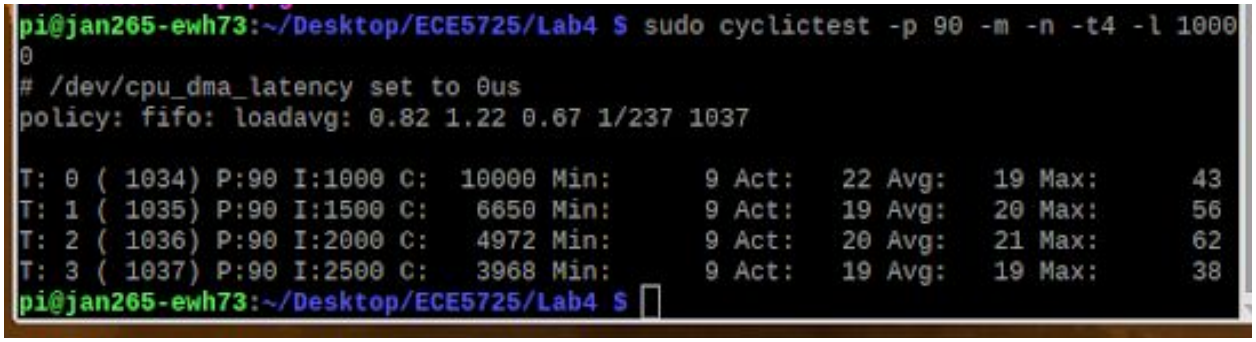

Figure 24 - Cyclictest result for 10000 loops with Preempt RT.

We can immediately see the improvement in performance, in which the max latency is significantly different. With Preempt RT, the max latency is 62, while before we have a max latency of 603. So this is our first proof that Preempt RT will allow a much lower max latency. We will proceed to see the max latencies when there are much more loops and when the system is loaded.

We will test on what will happen when there are 300000 loops being done by running the command below:

```
sudo cyclictest -p 90 -m -n -t4 -l 300000 -h 500
```

The result of the latencies information and the histogram plot is as follows:

```
pi@jan265-ewh73:~/Desktop/ECE5725/Lab4 $ sudo cyclictest -p 90 -m -n -t4 -l 3000
00 -h 500
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 1.78 2.20 1.89 1/242 1171

T: 0 ( 1165) P:90 I:1000 C: 300000 Min:      9 Act:   27 Avg:   20 Max:      72
T: 1 ( 1166) P:90 I:1000 C: 299972 Min:      9 Act:   19 Avg:   20 Max:      72
T: 2 ( 1167) P:90 I:1000 C: 299937 Min:      9 Act:   19 Avg:   20 Max:      58
T: 3 ( 1168) P:90 I:1000 C: 299910 Min:      9 Act:   19 Avg:   20 Max:      63
# Histogram
```

Figure 25 - Cyclictest result with 300000 loops with unloaded Pi

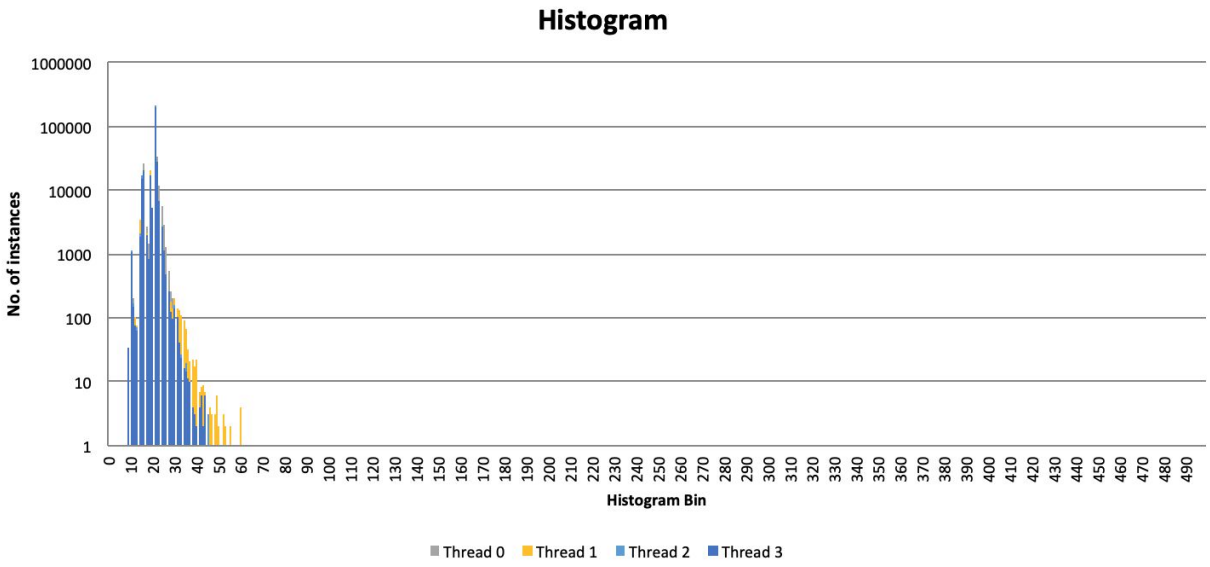

Figure 26 - Histogram bin of unloaded cyclictest with log scale.

Next, the “sort_v1.py”, which we have edited to run for a very long time (so we do not have to monitor that it is still running using htop), was run in the background. This is again used to keep the CPU running at 100%. The cyclictest command is again run to see if there will be any changes to the latency results:

```
pi@jan265-ewh73:~/Desktop/ECE5725/Lab4 $ sudo cyclictest -p 90 -m -n -t4 -l 3000
00 -h 500
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 3.63 2.87 1.81 2/230 1079

T: 0 ( 1069) P:90 I:1000 C: 300000 Min:      7 Act:    9 Avg:    9 Max:      40
T: 1 ( 1070) P:90 I:1000 C: 299975 Min:      6 Act:    8 Avg:    8 Max:      47
T: 2 ( 1071) P:90 I:1000 C: 299951 Min:      7 Act:    9 Avg:    9 Max:      55
T: 3 ( 1072) P:90 I:1000 C: 299926 Min:      7 Act:    9 Avg:    9 Max:      43
# Histogram
```

Figure 27 - Cyclictest result with 300000 loops with loaded Pi

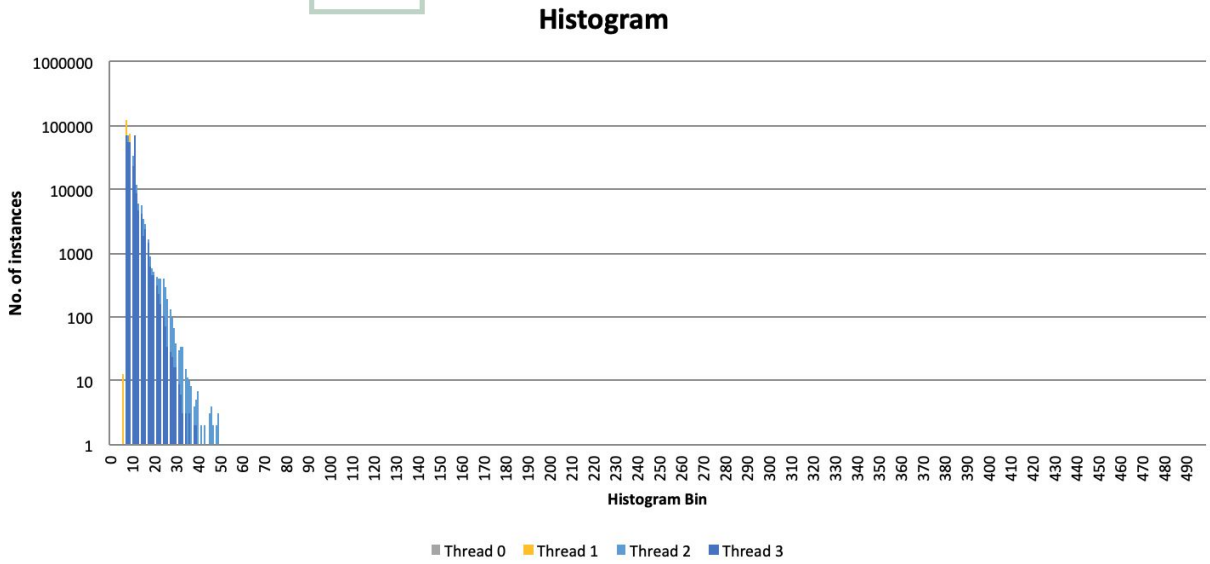

Figure 28 - Histogram bin of loaded cyclictest with log scale.

First, we compared the histogram result of the unloaded and loaded cyclictest in Preempt RT kernel. When Pi is unloaded, we can see more of a Gaussian distribution. The latency value with most occurrences is at about 23ms.

First, we compared the result of the max latencies of the unloaded and loaded cyclictest in Preempt RT kernel. Interestingly, the max latency when the Pi is loaded is slightly lower than when it is unloaded, of value 55 against 72. However, this value is so small that it does not really mean something significant. The program is already running as efficiently as possible and the max latency is already hovering at a value that is as low as it can get. Then, we compared the histogram result of the two scenarios. When Pi is unloaded, we can see more of a Gaussian distribution. The latency value with most occurrences is at about 23ms. On the other hand, when Pi is loaded most latency value is at around 10ms, and the occurrences linearly decrease as the latency bin gets bigger than 10ms. We can also see that there is less variance of latency values when Pi is loaded. This may be because when Pi is loaded and a task with a higher priority arrives (priority 90), the higher priority task will preempt and quickly be executed first. As a

result, in most occurrences the latency tends to be small and they are also more predictable (less variance). On the other hand, when Pi is unloaded, there is no other task waiting and so there is no urgency, and thus the latency value is more varied.

The significant changes that we again spotted is when these results are compared with non-preempt kernel. In the previous kernel, the max latency for unloaded and loaded pi was 878 and 1152, and this number became 72 and 55 with the preempt kernel. We can see how Preempt RT is making the scheduling run efficiently and thus having much less latencies. When comparing the histograms, we first noticed a slight difference during the unloaded Pi scenario between the previous kernel and the Preempt RT kernel. There is slightly less variance in Preempt RT kernel, in which the previously have histogram values up to 80s, but the Preempt RT kernel has histogram/latency values up to the 60s. There is not much difference because there is not really a background process running (except the linux processes) and there is not really preemption involved that allows a performance improvement. However, when comparing the loaded Pi scenario between the previous kernel and the Preempt RT kernel, we can clearly see the difference. The histogram in the previous kernel goes up to the latency values of 80, while in Preempt RT kernel it only goes up to the 50s. We can also clearly see less variance in the histogram data. This shows the latency improvement that Preempt RT can bring to the table when the Pi is busy or loaded. By allowing higher priority tasks to preempt lower priority tasks, it allows a generally lower and more predictable latency value, which is something that we want.

Looking at the results above and comparing with the previous kernel, we have learned why one would actually want to use a Preempt RT kernel patch. Preempt RT improves the scheduling latency significantly and so the maximum latency that could possibly happen is drastically lower. This is convenient for users as the latencies are now more predictable. The main reason that this happens is because Preempt RT makes the kernel to be fully preemptible. This means that everytime a kernel locks, we can preempt it. This gives a better monitoring of time during a process and enhances process scheduling. Some of the common problems in Linux such as priority inversion are now solved by how interrupts in Preempt RT are now handled with threads or that memories are locked immediately, and in how it allows priority inheritance.

We did not face any problems in this section and we can clearly see the difference of the results here compared to the previous kernel. It helps us to understand what is happening with Preempt RT.

Performance measurement using perf:

Next, we will be testing the performance of the “sort_v1.c” program by using the perf tool. The perf command below is then executed:

```
sudo perf_4.9 stat ./sort_v1
```

This will display the recorded statistics as below:

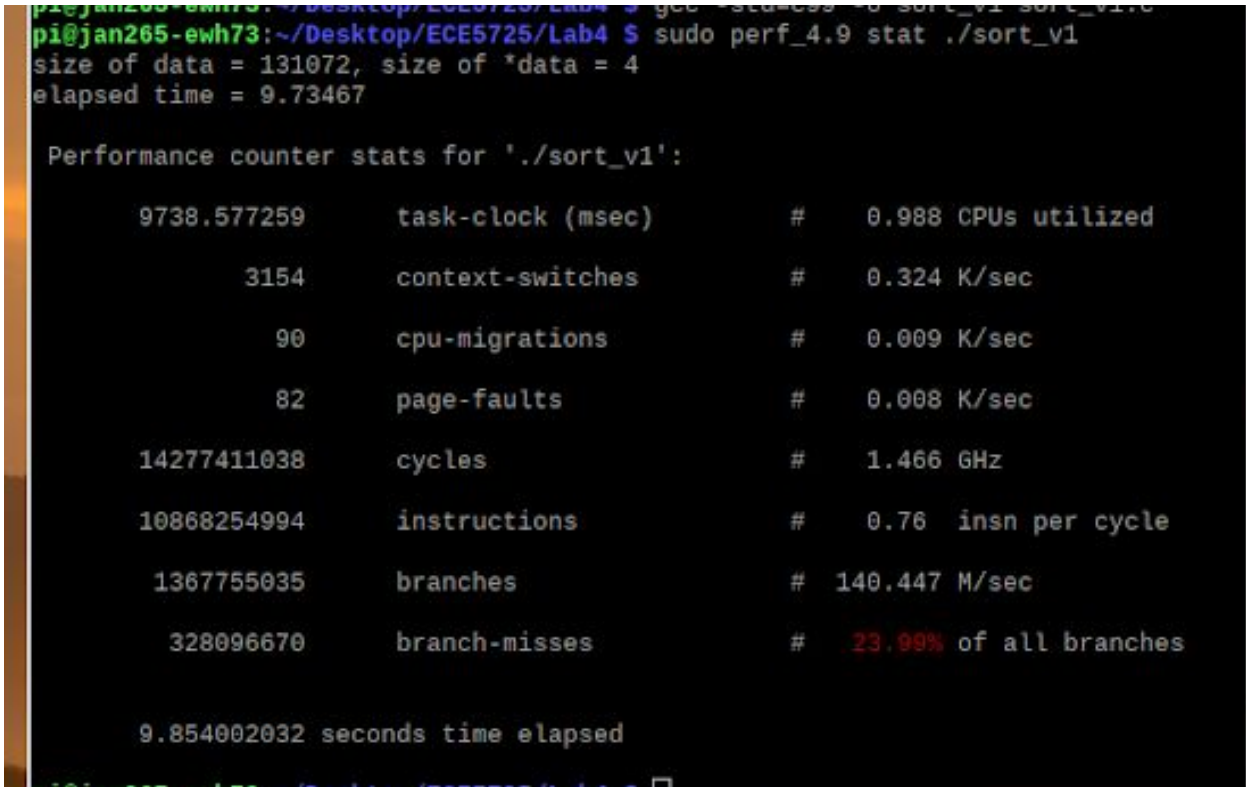

Figure 29 - Perf result of “sort_v1.c” without quick sorting performed.

Then, the “sort_v1.c” is modified to again include the quick-sort to the array (similar to what we did before). The perf command is again executed:

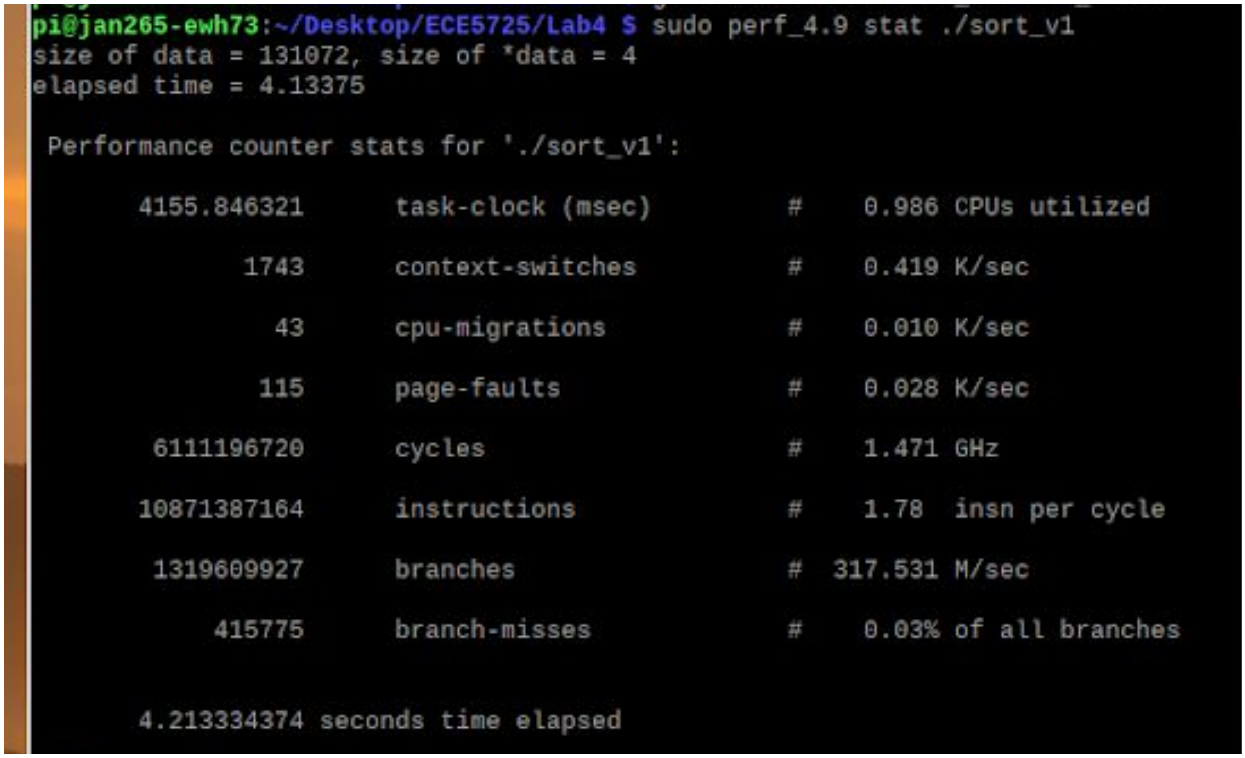

Figure 30 - Perf result of “sort_v1.c” with quick sorting performed.

Here, we noticed that the process again ran faster when the array is first sorted. This is because of the same reason as explained in the previous section when this experiment was run in the non-preemptRT kernel.

Next, we compared the results above to the result of the previous kernel version. There actually is not that much of a difference between the two kernel results. However, what we can see consistently is that the program is actually running 0.2 seconds slower in the PreemptRT kernel compared to the previous kernel in both scenarios (unsorted or sorted). Since the “sort_v1.c” is running at priority 0, this may have meant that processes running at priority 0 in PreemptRT actually run slightly slower.

Wrapping up this section, we have learned that a process at priority 0 may actually run slightly slower when using a Preempt RT patch. This is certainly something to take note of and that we can verify in the next sections ahead. This section went smoothly and no challenges faced.

Square Wave Tests:
In this section, we will perform the square wave tests, which will test on the maximum frequency that a program can generate now that we have a Preempt RT kernel installed. Since we now have a Preempt RT, we should modify the C program that we

previously used for the square wave generator. The program that we will be running is called “test_rt_v16.c”, which is a modified version of “test_rt_skel_v16.c” that is provided in the course server. The code itself has configurations that we can make because we are using a Preempt RT kernel. In the program, we actually modified several things to allow us to blink the LED and to read the GPIO pin. Below, we will discuss the things that are set up in this C program that does not exist in the C square wave generator program that was previously used.

The first configuration made is in setting a priority. Since we can now preempt, we set this program’s priority to 49 because we have learned in class that this would allow us to preempt to most processes (but not too high that it may preempt critical processes such as mouse or keyboard readings). The code is as below:

```
#define MY_PRIORITY (49)
```

Next, we also set up the stack size that the program can use:

```
#define MAX_SAFE_STACK (8*1024)
```

Then, we also set the scheduling policy to “Sched_FIFO” with a priority of 49. We are using “Sched_FIFO”, which is now available to us with PreemptRT kernel, because of several reasons. It has a First In First Out scheduler with non-zero priorities available, it does not run with time-slices, it allows preemption to other processes like batch or idle tasks, and also allows thread to be preempted by higher priority. The code is as below:

```
param.sched_priority = MY_PRIORITY;
if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
```

Next, we also lock the memory to avoid VM page faults and also prefault the stack to prevent a future random-access of stack variable from causing a stack fault that causes a delay. The code is as below:

```
mlockall(MCL_CURRENT|MCL_FUTURE) stack_prefault();
```

Compared to the previous C code that uses nanosleep for a delay between LED blinks, this program will use a for loop to keep the program running as we are using “sched_FIFO” scheduling policy and will need to keep the program running.

Finally, we made several changes to allow the program to blink the LED and have piscope read the GPIO. The code that is first added is in initializing the GPIO pin:

```
gpioSetMode(26, PI_OUTPUT);  // set GPIO 13 = OUTPUT
```

Then, the code below is added to have the GPIO switch between on and off after delays:

```
gpioWrite(26, PinValue);  // Send signal on pin 13
PinValue = PinValue ^ 1;  // Flip the value of the output pin
```

For reference, the full code is added in the Appendix section below.

The program is then compiled:

```
gcc -Wall -pthread -o test_rt_v16 test_rt_v16.c -lpigpio -lrt
```

Then, the program is ready to be experimented. In this program, we are able to pass a value of the interval directly to when calling the program:

```
./test_rt_v16 2500
```

Piscope was run after running the program to see the period and frequency measurements. We found out that the interval will represent the period of the wave, although it is not scaled perfectly. When 2500 is passed, it translates to 25000ns of period, which is 40000Hz in frequency. So when the interval is decreased, we noticed in piscope that the frequency will increase.

Next, similar to what we performed when using non-preempt RT kernel, we experimented on how high of a frequency can the C program produce a stable square wave. The experiment was done with unloaded Pi and loaded Pi (by running the sort_v1.c program). We swept through various frequencies and we are able to get the plot below:

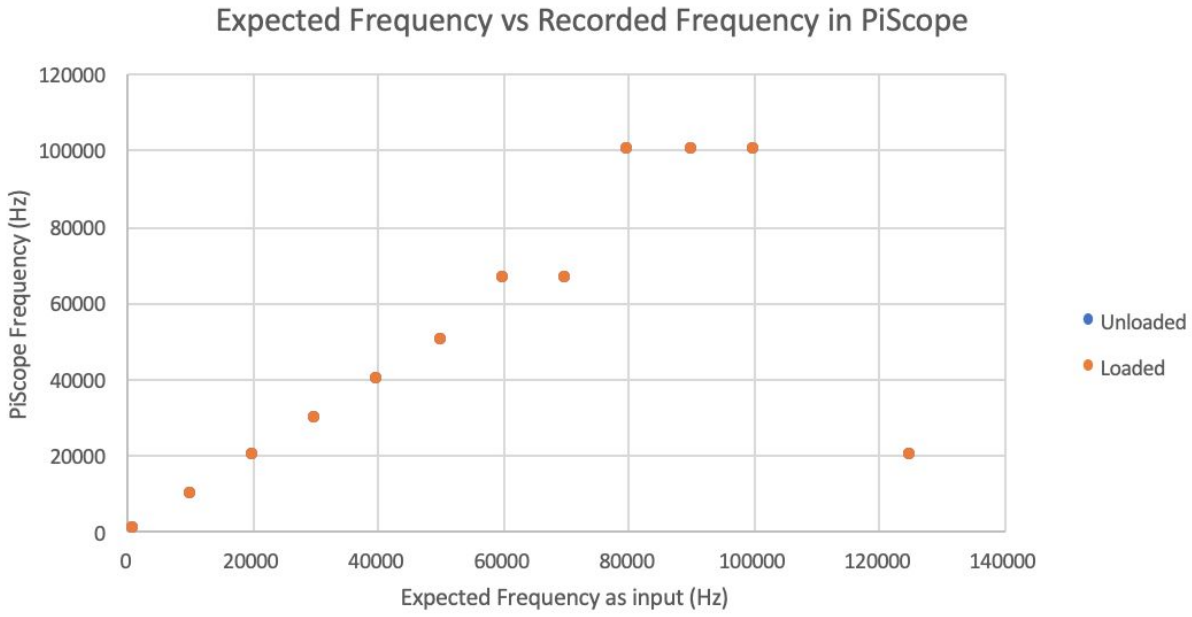

Figure 31- Frequency plot of “test_rt_v16.c” program in Preempt RT kernel.

The first thing to note is that the result of piscope is a bit flawed. What we noticed was that the smallest period increments was 5 microseconds. We can see how when the input frequency is at 60kHz and 70kHz, the recorded frequency in Piscope is 66kHz,

which corresponds to a period of 15 microseconds. Another thing is that when the input frequency goes beyond 50kHz, the displayed/measured square wave is a bit inconsistent (shown below). As a result, this affects the measured frequency in piscope. This is all because piscope is a software, unlike using a physical oscilloscope.

However, we were able to confirm that in both scenarios, they were able to stably output 100kHz square waves. We have a screenshot below that shows that when Pi was loaded, the program was able to generate a stable square wave at 100kHz:

Figure 32 - Piscope result when the program passes 100kHz during a loaded Pi.

In the figure above, we can see that the square wave has a period of 10 microseconds, which is 100kHz. However, we can also notice the shortcoming of piscope, in which the square wave in the middle of the plot is actually wider. This is what the discussion above was referring to, in which the output plot of square waves is often inconsistent when it has a frequency of higher than 50kHz. This inconsistency happens around every 10 cycles.

During our experiment, we also found that piscope cannot display square waves when frequency is above 100kHz. This causes us to not be able to actually find out what the upper bound frequency is when using Preempt RT kernel. The plot below shows what happened when a frequency of 125kHz was passed:

Figure 33 - Piscope result when the program passes 125kHz during a loaded Pi.

Concluding our findings above, despite the shortcomings of piscope, we were able to learn that using Preempt RT kernel allows us to have a higher upper bound frequency, as it can go beyond 60kHz (the upper bound frequency of C program with previous kernel). This fact was then confirmed by Professor, in which when using an oscilloscope, in the two scenarios Professor was able to produce frequencies up to the MHz range. This shows the dramatic improvement in performance and speed that Preempt RT kernel is able to provide. Looking at the changes that this program has, this performance improvement is because of how it now has a higher priority and also running a “Sched_FIFO” scheduling policy, and this allows us to preempt most of other processes and thus allowing this program to run faster. By being able to run faster, the program is able to output a square wave of a higher frequency.

In this section, the main challenge that we faced was when using piscope. We are quite confused about the output that we got from piscope because of how inconsistent the results were. Using piscope is certainly less ideal than using an oscilloscope. However, we are still able to learn how using a Preempt RT kernel and increasing the priority of a program can allow it to run more efficiently and at a greater speed.

Final square wave tests:

In this final section, we explored the features of the Preempt RT kernel, specifically the scheduling algorithm and the priority of the different tests that we ran. Firstly, we look into the “test_rt_v16.c” program. We first ran “test_rt_v16.c” to generate a stable square wave at the default interval/period. Then, we ran the following command “`ps-alef | grep test_rt_v16`”. The output is as below:

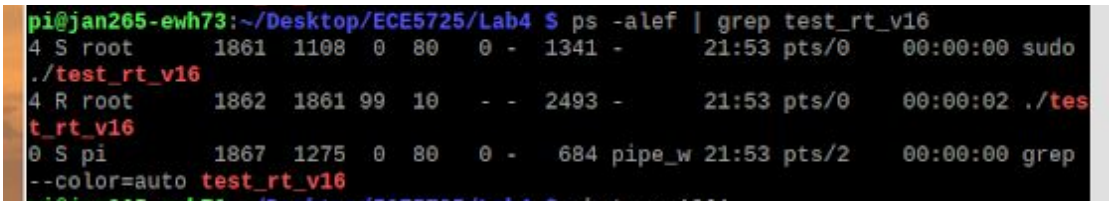

Figure 34 - Process of test_rt_v16.c.

From the output above from the ps command, we can see that there are a total of two processes related to the program. The first entry shows the program name with “sudo”, and the second entry shows when just the program process is running (without “sudo”). The fourth column shows the PID (process ID) and the fifth column shows the PPID (parent process ID). On the second entry, we can see that the PPID of this process is the PID of the first entry. This means that the first entry with the sudo is the parent, which will then call the second entry to run. Next, we ran “chrt” command to get the real-time scheduling attributes of a process. The command “chrt –p 1861” and “chrt –p 1862” were run as below:

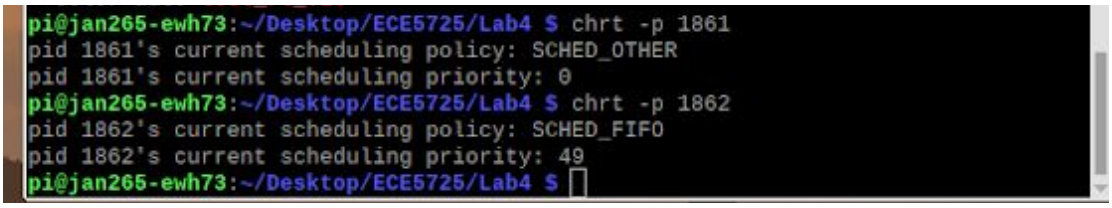

Figure 35 - Real-time attributes of test_rt_v16.c.

From the figure above, we can see that the sudo command will have a scheduling policy of “Sched_other” and a priority of 0. As a result, it will be easy for the process to be preempted by other tasks (such as that runs on sched_other or higher priority). On the other hand, the main program (second entry) is running “Sched_FIFO” at priority 49, which is what was declared in the program itself. Looking at this result, this confirms our earlier findings during the experiment on how the program is able to run quickly and with a better performance. This is because of the scheduling policy and priority that allows it to preempt others.

Next, we will look at the scheduling policy and priority of the “sort_v1.c” program. First, that program is run in the background. Then, “./test_rt_v16 2500” is ran. Similar to the findings from our earlier experiment, the “sort_v1.c” does not affect the square wave

generated in “test_rt_v16.c”. We will find that our by running the “ps–alef | grep sort_v1” and “chrt –p 1908”:

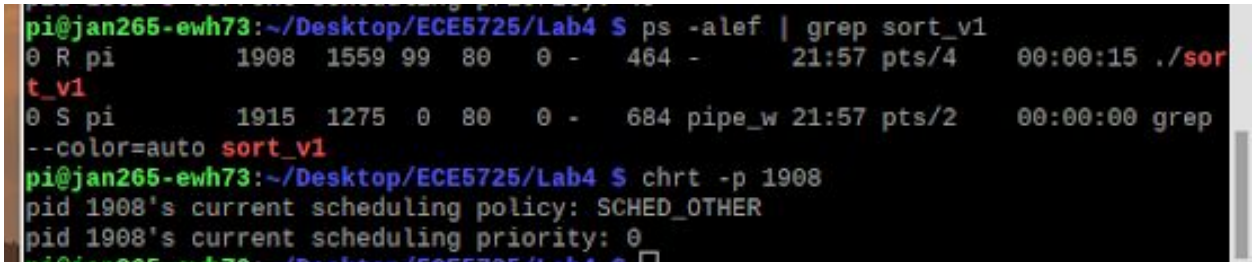

Figure 36 - Scheduling attributes of “sort_v1.c”.

We can see that the “sort_v1.c” program has a scheduling policy of “sched_other” and priority of 0. Since at the same time the “test_rt_v16.c”, which has a higher priority and scheduling policy of “sched_fifo”, is also running, the “test_rt_v16.c” program will preempt “sort_v1.c” (that has a lower priority). As a result, the “sort_v1.c” does not impact the square wave generated as it is getting preempted. This shows the benefit of using Preempt RT kernel, in which we can choose the priorities of different tasks or processes. This will allow tasks with higher priorities to not be interrupted by tasks of lower priority in their scheduling.

Finally, we ran “blink_v7.c”, which is the original LED blink C program that was previously run when testing the previous kernel. Then, we ran `“ps-alef | grep blink_v7”`, `“chrt -p 1918”` and `“chrt -p 1919”`:

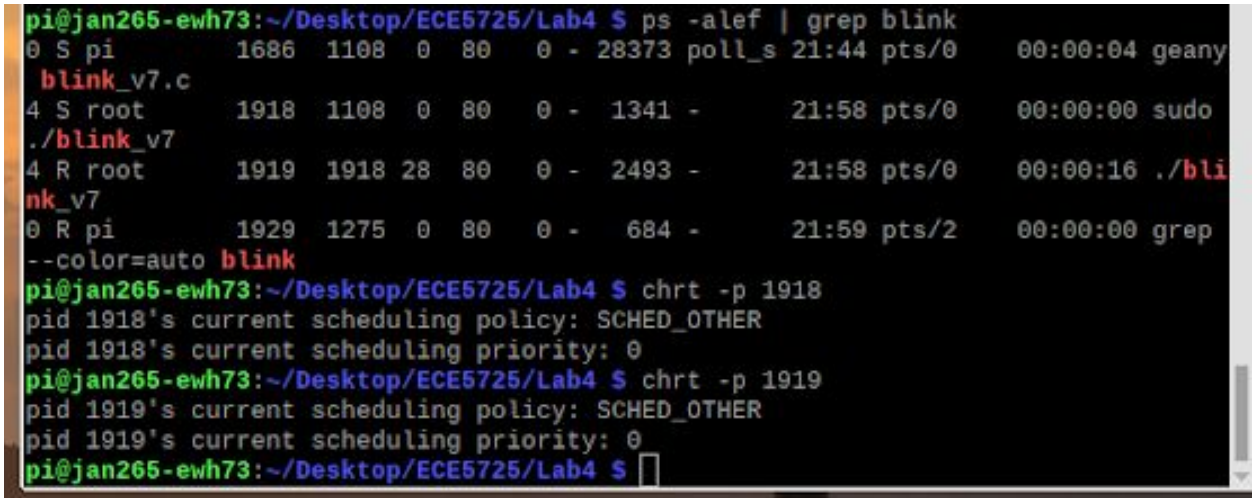

Figure 37 - Scheduling attributes of “blink_v7.c”.

In the figure above, we can see that there is the “sudo” process and the program’s process. Based on the PID and PPID, we can also see that the “sudo” process would call for the process of the program itself. Based on the attributes, we learned that both processes have the scheduling policy “sched_other” and priority of 0. This explains why the square wave in “blink_v7.c” will be affected by the “sort_v1.c” program. Since both are using “sched_other”, they will basically use round robin scheduling. Both of them

also have priority of 0. As a result, the square wave in “blink_v7.c” is affected by the sorting program. With the round robin scheduling algorithm, the task execution will be switched between each other and the program will slow down.

Last but not least, we explored whether we will have the same upper bound frequency when running “blink_v7.c” program with this preempt RT kernel. We used similar frequencies to what we used in the previous kernel. Below is the plot that we received:

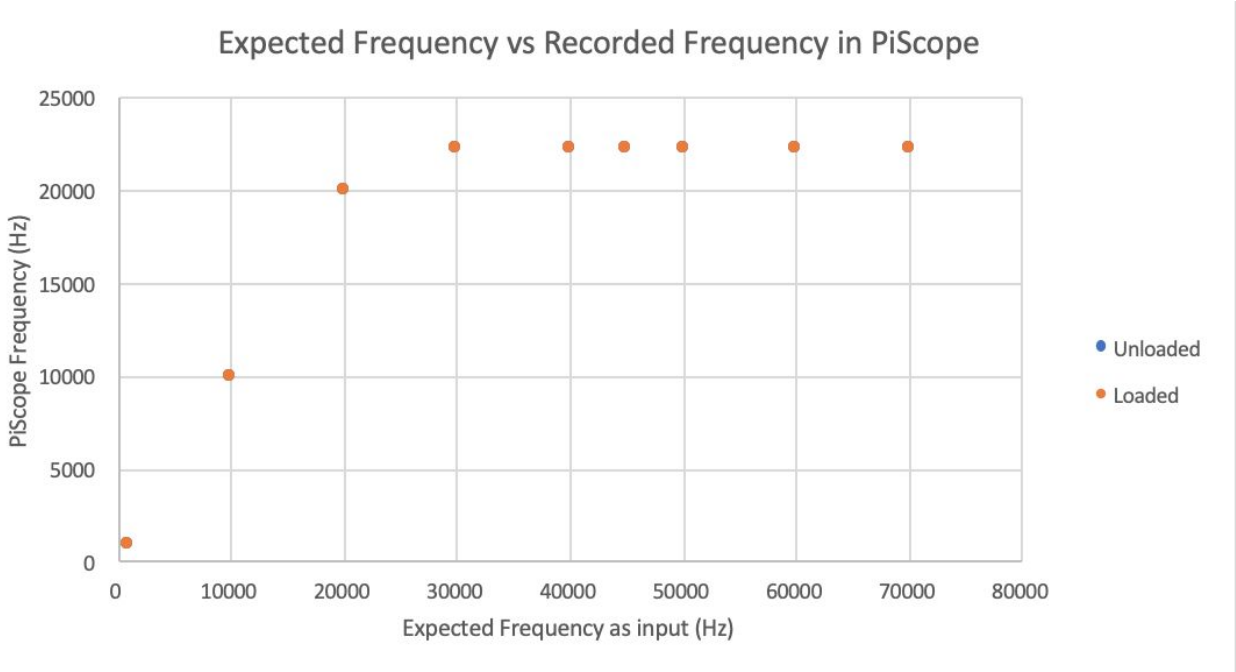

Figure 38 - Plot of “blink_v7.c” running different frequencies in preempt RT kernel.

In the plot above, we can see that the upper bound frequency above is 22kHz. This is surprising because we expected it to be able to produce at least 40kHz square waves. We get the piscope plot as below when running the program for 50kHz square wave:

Figure 39- Piscope plot showing 22kHz plot when program is supposed to be running at 50kHz.

Although this is somewhat unexpected, we may have found out a reason for it. From the perf experiment a few sections ago, we have learned that the “sort_v1.c” program is running slower in the Preempt RT kernel compared to the previous kernel. Additionally, “sort_v1.c” and “blink_v7.c” have similar scheduling policy and priority. As a result, it may be the case that in preempt RT kernel, priority 0 tasks will be running more slowly because they may be preempted by other higher priority tasks. As a result, the C program cannot generate a square wave of a higher frequency than 22kHz. After asking Professor Skovira, he mentioned that this is logical reasoning.

We have included a scheduling attribute summary below:

Program	Schedule policy	Schedule priority
test_rt_16.c	Sched_FIFO	49
sort_v1.c	Sched_Other	0
blink_v7.c	Sched_Other	0

Based on the table above, we can see how test_rt_16.c will be prioritized when all three programs were run altogether due to its scheduling policy and priority.

Wrapping up this section, we have learned about the different scheduling attributes that are involved between the programs that we used. We learned how having a higher priority or different scheduling policy may help boost the program's performance. Additionally, we have also learned on how a priority 0 in preempt RT kernel may actually run slower, compared to the previous kernel. Overall, this section is pretty challenging because there are a lot of new topics learned. However, doing these experimentations greatly help us to understand these topics.

Results:

In the first week of the lab, we first tested the performance of the pre-compiled kernel. The latency of the system with and without load was successfully tested by using the Cyclictest. The latency was determined to be less predictable and tend to have larger value when the system was loaded with CPU-intensive tasks. Then we were able to measure the performance of two different versions of sort_v1.c program by using the Perf utilities. We also successfully measured the performance of the square wave generation in Python and C codes on free GPIO pins. The GPIO output was measured with PiScope. The blink program was found to have a higher stable frequency in C code which was around 50kHz when system was unloaded and 40kHz when system was loaded, while in Python the maximum frequency was around 900Hz under both conditions. After the performance measurements, we correctly downloaded, configured and compiled the new Linux source files and PreemptRT patch. The installation of the PreemptRT kernel was verified to be successful by checking the log files created during the compilation process. All measurements and compilations were performed as expected in the first week of the lab.

In the second week of the lab, after verifying the correct version of the PreemptRT kernel, we first tested the performance of the new kernel with the Cyclictest. The average latency was found to be higher than the old kernel when the system was unloaded but significantly lower when the system was loaded. The maximum latency was lower under both conditions. The latency was also found to be more predictable in the PreemptRT kernel. Correctly performing the Cyclictest, we then did the Perf test for the two versions of sort_v1.c program in the new kernel. The test gave a similar result as the test performed in the old kernel. After the Perf test, the performance of square wave generation of the PreemptRT kernel was tested with a different C code. The highest stable frequency acquired was 100kHz, which is higher than when using the previous kernel. We could possibly get an even higher frequency but measurements were limited by PiScope capabilities. We have also explored the features of the PreemptRT kernel by looking into the priorities of different programs. The blink program was tested again to check the performance in the PreemptRT kernel. The highest stable frequency found was lower than the value found in the old kernel possibly because of

how Preempt RT kernel deals with priority 0 tasks. Finally, we explored the different scheduling attributes of the programs that we have used and on how it affects the performance of the program.

Overall, all of the goals and tasks of the lab were achieved. The only thing that did not work was the PiScope, in which it could not make frequency measurements above 100kHz. Other than that everything went well. The lab checkouts were done either during the lab section or the weekend office hour.

Conclusion

In this lab, we first learned how to explore the performance of the system by measuring the latency with Cyclictest and performed the measurements to the non-PreemptRt kernel. Then, the performance of the system was further measured by running Perf utility and using PiScope to determine the square wave generation ability on free GPIO pins. Following the instruction, we installed the PreemptRT kernel and did the same performance measurements to the new kernel. By comparing the results of two different kernels, we were able to have a better understanding of the PreemptRt kernel. Everything performed as expected in this lab and all goals were achieved. However, we also experienced some issues. The biggest issue we encountered was measuring the square wave generated on the GPIO pin with PiScope. Due to the inconsistency of the PiScope, we had a hard time determining the upper limit of the frequency. While sometimes the PiScope was able to display a stable square wave, sometimes at the same frequency it failed to display the square wave as expected. The PiScope was limited to measure the frequency up to 50kHz. Beyond 50kHz, the measurements of the PiScope became very unreliable and limited. As a result, the measurements could be extremely confusing sometimes and we were unable to fully explore the performance of the PreemptRT kernel. While the PiScope measurements could be confusing, the lab instruction was quite clear and easy to follow. The only suggestion for the future labs would be to have an alternative to PiScope that is more reliable.

All of our code has been uploaded to the ece5725 server:

`/home/Lab4/M_jan265_js3596_4`

APPENDIX

Test_rt_v16.c

```
//
// jfs9 10/24/15
// v2 Add wiringPi to toggle output
// v5 10/22/18 - add frequency display
// v6 10/22/18 - loop not nanosleep
```

```
// v16 10/24/2020 - convert from wiringpi to pigpio
//          clean up some unused vars and code
//

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>
#include <pigpio.h>

#define MY_PRIORITY (49) /* we use 49 as the PRREMPT_RT use 50
                            as the priority of kernel tasklets
                            and interrupt handler by default */

#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
                                   guaranteed safe to access without
                                   faulting */

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

void stack_prefault(void) {

        unsigned char dummy[MAX_SAFE_STACK];

        memset(dummy, 0, MAX_SAFE_STACK);
        return;
}

int main(int argc, char* argv[])
{
        struct sched_param param;
        int interval = 500000000; /* long delay */
        int PinValue = 0;  // used to toggle output pin
        int i;


       if ( argc>=2 && atoi(argv[1] ) >0 ) { // if positive argument
          interval = atoi(argv[1]);
       }
       printf ( "Interval = %d ns\n", interval);

        gpioInitialise();   // initialize pigpio
        gpioSetMode(26, PI_OUTPUT);  // set GPIO 13 = OUTPUT

        /* 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();

         while(1) {

               for ( i=0 ; i<interval ; ++i ) {  /// use delay loop to
control frequency
                   // interval 1200 = about 60k Hz
               }

               //  code to control GPIO goes here....
               gpioWrite(26, PinValue);  // Send signal on pin 13
               PinValue = PinValue ^ 1;  // Flip the value of the output
pin
   }
}
```