Lab 4

Outline

In Lab4, we will be exploring the PreemptRT kernel patch. Investigation into performance begins with the current installed version of Raspbian using some simple C-based signal generation techniques. Next, a new kernel will be built using the Raspbian 5.4.72 source and the corresponding PreemptRT patch, version 5.4.70. Once the patched kernel is running, we will explore signal generation using Real Time techniques provided by the patch.

In a bit more detail, we will follow the steps:

- On the current system:
  - Install utilities for managing GPIO in C
  - Implement a square wave signal in C
  - Measure the performance of the C code
- Install a new kernel, patched with PreemptRT
  - Download the Linux source files
  - Download the PreemptRT patch
  - Merge the patch with the Linux source
  - Configure the Linux code
  - Compile the Linux kernel
  - Copy the kernel to the SD card and boot the R-Pi
- On the new system
  - Install utilities for managing GPIO in C
  - Run the square wave code to check this install
  - Convert this system to a real-time code
  - Test performance of the real-time system

There are several new packages to install along the way. Most are straightforward. The Linux patch application is a bit complex; however, these techniques are good to know for general knowledge in altering a Linux kernel; something that will be common in embedded systems.

The following packages will be installed throughout steps in the lab:

- Cyclictest
- Perf
- pigpio
- kernel source
- Preempt-RT patch

The following will be used to verify system performance:

- Perf using the sort programs
- The Cyclictest utility
- Measurement of signals using the PiScope

Canvas links

- Cyclictest
- Pigpio
- pigpio C code example
- github link for Linux source branches
- RaspberryPi.org link for kernel building
- PreemptRT how to build applications, code skeleton

Before we start

In this lab, we will be replacing the current kernel on the SD card. This will ERASE everything on the card. There are important tasks you should perform before you do anything else:

1. **As a first step, Backup your SD card** using the instructions on Canvas. This will preserve the entire Lab 3, non-preempt kernel image so that you can restore it as needed.
2. As an additional safety measure, use your favorite cloud / USB stick copy / ece5725_f20 server method to save your programs from previous labs. Having these in a handy location will make them easy to restore onto the newly formatted card, as needed.
3. There are a number of links in this weeks' reading in Canvas. These readings contain a lot of information relative to Lab 4. Please take some time to read these selections as this will save you a lot of time on lab procedures.
4. The following Lab instructions have been developed from a variety of links included in Canvas and a number of experiments in compiling the RT kernel. When you review the links, you will notice the various items included in these instructions.

Part 1: Verify current system performance

The following will be used to verify system performance:

- Perf using sort programs and square wave programs introduced in Lab 4
- The Cyclictest utility we have been discussing in class
- Measurement of signals using the oscilloscope

Performance measurement using cyclictest

Download, install, and run Cyclictest on the current system

Cyclictest is loaded from github. Canvas includes a link to the cyclictest page; please reference this page for complete instructions. The short form is included below:

```
pi@RPi3-jfs9$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
pi@RPi3-jfs9$ cd rt-tests
pi@RPi3-jfs9$ make all
pi@RPi3-jfs9$ sudo make install
pi@RPi3-jfs9$ sudo cyclictest --help
```

Once cyclic test is installed, test it by running *cyclictest --help*

Check the latency of the system by running:

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

This command runs threads at priority = 90, uses the nanosleep timer, locks memory, runs four threads, for 10000 loops. After you record these results, re-run the command with the following options:

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

This command runs threads at priority = 90, uses the nanosleep timer, locks memory, records 500 histogram bins, runs four threads, for 300000 loops. This should run in about 5 minutes.

Record the details of the histogram and convert the histogram into a chart in Excel to capture a record of the maximum latencies. What do these results show about latency of the system?

You should also test the behavior of cyclictest while running cpu-intensive tasks on the RPi cores. Note that sort_v1.c (in the next section) is a good cpu loading test.

Run 20 – 30 copies of sort_v1 in the background and repeat the command:

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

Use htop to make sure the cores remain busy for the duration of the cyclictest run. You might need to restart sort_v1 programs during the run. You might also consider modifying sort_v1 to run a bit longer.

Later in the Lab, you will compare these histograms of cyclictest on the loaded and unloaded RPi against the same tests under the RP-PREEMPT kernel.

Performance measurement of sort.c

From the ece5725-f20 server, download:

```
/home/jfs9/lab4_files_f20/c_tests/sort_v1.c
```

have a look at the code with your editor of choice to understand the code

Use the following command (the C compiler, gcc) to compile the source:

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

Note: the –std=c99 is used for sort_v1 to support older C calls. This flag is not generally used for other C programs. Once the file has been compiled, run the executable sort_v1 using perf:

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

Record all the results.

Modify sort_v1.c to include a sort on the large array prior to entering the loop to sum results. Hint: LOOK at the sort_v1.c code carefully to add the sort function. Compile and run the altered file. Run the executable using perf. Record all the results. In your lab report, please describe the difference in the performance measurements relative to system architecture.

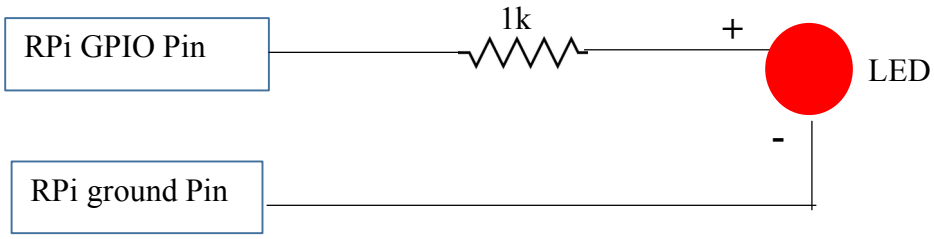
Performance measurement of square wave in python and C

Quick summary:

- Use blink.py to generate a stable square wave in python..
- Use blink_v7.c to generate a stable square wave in c.
- Run blink_v7.c again with system loading and check the stable square wave

Design a python program, blink.py, to output a high/low pulse on a free GPIO pin. The pin you select will be used for the duration of Lab4 so insure that it is a free GPIO pin that may be safely configured as an output. Note that a PWM motor control pin from Lab3 would be an excellent choice (the LED may still be connected!)

As in Lab 3, one useful method for testing is to attach an LED (with appropriate 1k resistor) to this pin. For low frequency signals, the LED will blink when the software is running correctly. As functions in the lab will vary from Python to a few flavors of C, on two different kernels, the low speed LED check is a good way to quickly verify program correctness. From Lab 3, recall the example LED circuit:



Once you have blink.py running, use PiScope to monitor output pin and, decreasing the period of the generated wave, determine (roughly) where a stable square wave may be maintained. Use the PiScope pause functions and two cursors to measure the generated period which you can then convert to frequency. Determine the upper limit for the frequency that can obtained with the blink.py program. If the frequency set in the blink.py program matches (**within 10% of the expected frequency**) what PiScope reports, the generated wave is stable. You should expect to reach an upper bound for the frequency reported by PiScope (this will be somewhat subjective). Plan to plot a number of points for this experiment (for example, expected frequency versus frequency reported by PiScope). Note; please see Appendix C, below, for a discussion of PiScope.

Once you have determined a stable value for blink.py, generate a similar square wave program in C.

IMPORTANT:

Using the pigpio C library, you can now work on the square wave application. Plan to use the same output pin as you used for the Python code. When using the pigpio library, you will have to make sure to indicate this to the C compiler. There is a link on Canvas with sample 'blink' code. Here is a snip:

```
#include <stdio.h>
#include <pigpio.h>

#define LED     13  // GPIO pin 13 (pigpio uses BCM numbers)

int main (void)
{
  gpioInitialise();  //  Setup pigpio

  gpioSetMode(LED, PI_OUTPUT);  // set GPIO 13 as an Output

  for (;;)
  {
    gpioWrite (LED, 1);     // LED On
    gpioDelay (500000);     // 500 mS delay
    gpioWrite (LED, 0);     // LED Off
    gpioDelay (500000);      // 500 ms delay
  }
  return 0;
}
```

Notes:

- pigpio used BCM numbers for GPIO. So, pin 13 is GPIO pin 13 as we have been using in previous labs; no special setting in pigpio to set this numbering system.
- The 'gpioDelay' call in pigpio accepts integers only
- Please check the lab4_files_f20 files on the server. blink_v7 shows a version of blink which uses a nanosleep function; this should give better results for all frequencies.

To compile a C program using the pigpio library, use the command:

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

Once you have the code running and the LED blinking, PiScope test to (roughly) determine a minimum setting for the period in the C code to generate a stable square wave. Note that if the frequency reported by PiScope is within 10% of the expected

frequency, consider this a stable result. Plan to plot a number of points for this experiment (for example, expected frequency versus frequency reported by PiScope)

Next, create a python or C code designed to use 100% of a single cpu (floating point computations are good candidates for using a lot of cpu time… like sort_v1.c!). Once you have this running, launch it at least 4 times and verify (using htop) that all cpus are 100% used.

Once the system is loaded, run the two blink tests again on a loaded system; run the loaded systems for both blink.py and blink_v7.c. Are you able to generate a square wave as in the first performance experiment? Adjust the period as needed to generate a stable square wave, generating a second graph of these results.

Part 2: Build the PreemptRT kernel

There are a number of steps in this section. If you follow the installation plan all will be well; skip a step or wander from the instructions at your own peril! As noted above, the basic steps to creating a new Linux kernel include:

- Download the Linux source files
- Download the PreemptRT patch
- Merge the patch with the Linux source
- Configure the Linux code
- Compile the Linux kernel
- Copy the kernel to the SD card and boot the R-Pi

There is a link on Canvas to instructions for installing kernel changes. This is a reference for the detailed instructions that follow. A few notes about the process:

- There are two main elements to forming the kernel: The Linux source files and the PreemptRT patch files. The idea is to combine these two source repositories in order to compile a new kernel.
- There are a number of versions of each of these packages. We have found specific versions that work. We will be using a version of the most recent Raspbian kernel that we know is stable with the PreemptRT patch. The versions we will be using are:
    - Kernel files: 5.4.72
    - PreemptRT patch: 5.4.70-rt40
- There are a number of methods for creating the PreemptRT kernel:
    1. Begin with a clean install of Raspbian and compile on the Rpi
    2. Begin with an SD card loaded with a copy of the current ('Lab 3') version of Raspbian and compile the modified kernel (5.7.72 with PreemptRT patch) on the Rpi
    3. Cross compile the PreemptRT kernel on a Linux server (Ubuntu, for example). Once complete, move the compiled kernel elements to the RPi SD card
- We have been most successful with (2), compiling using the 'Lab 3' version of Raspbian and compiling on the Rpi. The advantages for this method:
    - All work is done on the Rpi platform
    - We save a number of steps, including transferring files from a server to the RPi
    - This method preserves all of the work done so far in Labs 1-3

- However, there are some disadvantages
    - The biggest disadvantage is the kernel compile time on the RPi (60 minutes). This is the big motivation to go to a cross-compile platform; more memory and a faster processor lead to faster compile times.
    - Because of the limited space on the SD card, we need to manage free space to allow the compilation to complete.

There are techniques to overcome these disadvantages. The alternative compile methods are available if you would like to try them as well. As long as we can achieve the goal of a compiled PreemptRT kernel, alternative paths are acceptable.

Next are the step by step instructions for compiling the Kernel patched with Preempt-RT.

Step 1: Initial Setup and Download Linux Source Files

We are now ready to begin creation of the patched kernel. Refer to the Canvas links for several methods used for the installation of a new kernel. The steps below detail the path used to build the kernel on the Rpi. Please execute the steps below in exact order.

Start with your saved image of the Lab 3 SD card. Remove the modifications to .bashrc so that the kernel boots to the piTFT without running any Lab3 programs.

If you haven't already:

BACKUP the Lab3 SD Card

This is important as the work on the Preempt-RT kernel will replace the working Lab3 kernel. You will need a backup to restore to the non-preempt kernel.

Free Space on SD Card

You may want to free as much space as possible on the SD card for the upcoming downloads and compilation. This is less critical than in the past since we have switched from 8 GByte to a 16 GByte SD card. However, it is good practice to keep the SD card trim because we will be loading A LOT of C code for the compile process.

- Remove large mp4 videos, games, and experimental applications. (1.7G free)
- Make sure wolfram-engine is removed:
    - sudo apt-get remove --purge wolfram-engine
- Remove supercollider (music processing software) (1.9G free)
    - sudo apt-get remove --purge supercollider*

- Turn off swapping on in the current kernel (see Canvas links in References / SD Card Management) (2.5 G free)
- Complete with the final set of commands: (2.9G free)
    - sudo apt-get autoclean
    - sudo apt-get clean
    - sudo apt-get autoremove

A few commands for checking storage:

df –h

Checks the free space in the filesystem

dpkg –get-selections

Shows all installed applications

du –sh *

Details the storage used in a particular directory. For example, you can use 'cd /var' followed by 'du –sh *' which will detail storage in the /var directory

Using these methods, you should able to free enough space to enable subsequent downloads and compilations as detailed below. Continue to check, and record, SD space as you move forward with Lab 4. This is especially important before you begin the long compilation run (more on this a bit later).

Next load the appropriate source code for the base Linux kernel:

```
user@RPi ~$ cd /home/pi    # the home directory for the pi user
```

DO NOT cut and paste the following command. Additional control characters may be introduced during cut-and-paste and, although the command may appear correct, it will not execute correctly!

```
user@RPi ~$ time git clone --single-branch --depth 1 --branch 'rpi-5.4.y'
https://github.com/raspberrypi/linux.git
     #  The above git command clones the branch 'rpi-5.4.y'
     #     to a local directory named 'linux' under the /home/pi directory.
     # 'clone' commands copy code from git repository to a location on a
     #      local machine (Rpi, in this case)
```

Sample command output plus an indication of remaining space once Linux 4.4.y C source code is loaded on to the SD card:

```
pi@RPi-jfs9:~ $ time git clone --single-branch --depth 1 --branch 'rpi-5.4.y'
https://github.com/raspberrypi/linux.git
Cloning into 'linux'...
remote: Enumerating objects: 1864, done.
remote: Counting objects: 100% (1864/1864), done.
remote: Compressing objects: 100% (262/262), done.
remote: Total 6762345 (delta 1696), reused 1602 (delta 1602), pack-reused 6760481
Receiving objects: 100% (6762345/6762345), 1.85 GiB | 3.95 MiB/s, done.
Resolving deltas: 100% (5659001/5659001), done.
Checking out files: 100% (65177/65177), done.

real      17m31.304s
user      20m59.997s
sys 2m45.607s
```

Note above example timing was measured without the '--depth 1'. Using this git option, download time for the code should improve.

```
pi@RPi-jfs9:~ $ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  6.5G  7.3G  47% /
devtmpfs        805M     0  805M   0% /dev
tmpfs           934M     0  934M   0% /dev/shm
tmpfs           934M  8.6M  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
```

I downloaded the above code while the RPi was connected to WIFI. Time to download was between 10 and 15 minutes (again, without the –depth 1 option….)

To check the version of the code you have downloaded, run:

```
$ cd /home/pi/linux
$ head Makefile
```

The top of the Makefile should indicate the level you just downloaded. The release should be 5.4.72

Make sure any missing dependencies are installed by running:

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

Step 2: Download the PreemptRT Patch and merge with kernel source

Get a copy of the PreemptRT-patch, directly into the 'linux' source directory, by running:

```
user@host $ cd /home/pi/linux

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

Important: Once you have confirmed that you have the correct version of the linux source, and have downloaded the patch file:

Backup the SD card

This will preserve a 'clean' copy of linux source in case there are issues with later install steps.

After you backup the SD card, apply the patches to the linux source:

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

Notes: The –p1 option is 'patch p-one'….not 'patch p-ell'
Output from this command will be placed in '/home/pi/patch.log'
A sample 'patch.log' is saved in Lab4 links

After this command runs error-free (a few seconds) patches have been applied to the original kernel source. Check /home/pi/patch.log to make sure there are no errors.

Step 3: Configure the kernel

Begin by making sure the install will begin with a clean make environment. Note that all commands are run in /home/pi/linux unless indicated otherwise:

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

The above two commands remove any previously compiled files from the /linux directory

Give the kernel image the correct name (kernel7l). The second command insures that the default configuration for the R-Pi is created.

```
$ KERNEL=kernel7l   # Setting to compile the RPi 4 kernel
$ echo $KERNEL   # to check environment variable setting
kernel7l          #  kernel7 will be returned if setting is correct
$ make bcm2711_defconfig   # use make to build the RPi4 config file
```

Notes: KERNEL=kernel7l # the KERNEL environment variable is set to
kernel7l (kernelSeven-ell)
NOT kernel17 (kernelSeventeen)
kernel7
kernel71 (kernelSeventyOne)
… or any other variation!

The 'make bcm2711_defconfig' command creates a skeleton .config file to be used by the menuconfig command.

Run make menuconfig to check/change settings for the upcoming compilation

```
$ make menuconfig
```

menuconfig allows you to select a LARGE number of options for the kernel configuration. Look through the menu that pops up; note the navigation tips at the top…you can use help at any point to get more information on any selection in the menu. Please look through the menu and make the following selections (if they are not already set):

- In 'General Setup, 'preemption model', select 'RT Fully Preemptable kernel'
  Note, that 'preemption model' could also appear in the 'Kernel Features' section

- In 'General Setup', 'Timers subsystem', select 'High Resolution Timer Support'

As the instructions indicate, please save your changes, in the default location (which is the .config file), to complete menuconfig.

Step 4: Compile the patched, configured kernel

A few notes before you begin compilation:

- At this point, a lot of new source code has been loaded onto the SD card. Check the available free storage on the card using 'df –h'. In my experiment, I began with about 7.5 Gbytes free at this point.
- Once you run the initial make command, below, it will run for about 60 minutes on the Rpi. During this time, the Rpi should remain powered up (!) and should be undisturbed until the compilation is complete. The best solution would be to run this between week1 and week2 lab sessions.
- Note that the results of the make command are saved in /home/pi/make.log. A sample make.log is saved in the Lab4 link on the class server for comparison.
- The make command will use all of the cores on the RPi. Do not run anything else on the RPi (like web browsers, full screen editors, etc) when running the make command

The first command is the long compilation of the Linux kernel. The 'tail' command will indicate progress as the compilation is underway.

Week1 Checklist
Show the TAs a copy of the following

- blink.py and blink.c at the highest recorded frequency when the Rpi is unloaded
- blink.py and blink.c at the highest recorded frequency when the Rpi is loaded
- saved data from cyclictest on the original kernel
- perf results for the two sort programs
- the linux source directory at the correct version with the correct patch file
- a copy of patch.log
- display the results of 'df –h' to show free space on the SD card
- Backup files: Lab3 system backup

Stop HERE for Week 1 Lab session. The following commands should be run between Week1 and Week2

This is the break between Lab4, week1 and Lab 4, week 2. The second week of Lab 4 will begin with testing the function of the PreemptRT kernel. If you end week 1 at this point, you will be taking the Rpi home and performing the compile step between week 1 and week 2 lab sessions. Please plan accordingly…..

If you are compiling outside of the lab, don't forget the following important steps:

- Check the value of the $KERNEL environment variable before you begin.
- Make sure you run commands in the /home/pi/linux directory.
- The following commands should help:

```
$ cd /home/pi/linux
$ KERNEL=kernel7l
$ echo $KERNEL   # to check environment variable setting
kernel7l        #  kernel7l will be returned if setting is correct
```

The first compile step (below) will run for about 60 minutes on the RPi. Please plan to run this step between the Week 1 and Week 2 Lab 4 sessions.

For the compilation, run the following commands**:**

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

In a second window:
    $ tail –f make.log   // to watch progress

Note: This command will take about 60 minutes to run.  Once it has completed,
plan to check the make.log to see if there were any errors during compilation
(check the sample 'make.log' saved in /home/jfs9/lab4_files_f20 on the class
server.)
```

```
Before you run the next command:
• Check the file make.log for any warnings or errors
• Make sure the $KERNEL environment variable is still set
```

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

The –j4 argument informs the make command to start 4 tasks which should use all four processors on the R-Pi. The first 'make –j4 zImage modules dtbs' command performs the bulk of the compilation and will take about 60 minutes on a R-Pi4.

After completing both commands, note the remaining free space on the SD card:

```
pi@RPi3-jfs9:~/linux $ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        15G  7.1G  6.9G  51% /
devtmpfs        460M     0  460M   0% /dev
tmpfs           464M     0  464M   0% /dev/shm
tmpfs           464M   13M  452M   3% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           464M     0  464M   0% /sys/fs/cgroup
/dev/mmcblk0p1   43M   22M   21M  52% /boot
tmpfs            93M     0   93M   0% /run/user/1000
```

The second make command will run in a few minutes. Results will be saved in the modules_install.log (see the sample in the Lab 4 link for comparison)

IMPORTANT: **Backup the SD card at this point.**

If the log files show that the above operations occurred without errors, the system will still be configured with the original, Lab 3 kernel. In addition, the newly compiled PREEMPT RT kernel is saved to the card but HAS NOT yet been completely installed.

Lab 4, Week 2

IMPORTANT: If you completed the compilation, make sure to **Backup the SD card at this point**. If the log files show that the above operations occurred without errors, the system will still be configured with the original, Lab 3 kernel. In addition, the newly compiled PREEMPT RT kernel is saved to the card but HAS NOT yet been completely installed.

After backup, reboot the SD card and continue with the remaining installation instructions.

Run: uname –a

Record results for the current kernel. At this point, the kernel should show the Lab 3 version (5.4.51, on my lab system). After kernel installation, you will use uname –a to confirm the updated kernel version.

```
pi@RPi-jfs9:~ $ cd linux
pi@RPi-jfs9:~/linux $ KERNEL=kernel7l
pi@RPi-jfs9:~/linux $ echo $KERNEL
kernel7l
pi@RPi-jfs9:~/linux $ uname -a
Linux RPi-jfs9 5.4.51-v7l+ #1333 SMP Mon Aug 10 16:51:40 BST 2020 armv7l GNU/Linux
```

The following commands are used to place elements of the new kernel in the correct locations:

```
$ cd /home/pi/linux # make sure you are in the linux source directory
$ KERNEL=kernel7l   # Make sure the $KERNEL environment variable is set
$ echo $KERNEL     # to check environment variable setting
$ kernel7l          #  kernel7 will be returned if setting is correct
$
$ 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
```

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.

Once these commands are run, a reboot should bring up the new PreemptRT kernel.

Step 5: System Setup

At this point, the new kernel has been compiled and written to the SD card. On reboot, the newly compiled kernel will be launched

Note that you **might** have to perform the initial raspi_config steps from Lab1. This step depends on your initial starting point. If you discover that you need to run raspi-config, please refer to the Lab1 writeup on Canvas if you need some hints on how to perform the initial setup. The section from Lab 1 describing raspi-config is included in an appendix.

Part 3: Check performance of the PreemptRT system

OK, at this point, you have a shiny new PreemptRT kernel running on the R-Pi. Now what? First, check to make sure that the PreemptRT kernel is indeed running:

```
pi@RPi-jfs9:~ $ uname -a
Linux RPi-jfs9 5.4.72-rt40-v7l+ #1 SMP PREEMPT_RT Sat Oct 24 16:20:20
EDT 2020 armv7l GNU/Linux
```

Note that the kernel indicated is 5.4.71 and shows that the Preempt RT kernel is running

If you started with the 'Lab3' base, you code from past labs should still be in place. Try some examples to make sure everything continues to run. If you have a 'Test Plan' in place including some of your favorite codes, this would be a great time to run the test plan

Since we have a new kernel, we need to install a few things in order to check performance:

- For Python. Rpi.GPIO should be pre-installed so we are OK here.
- Perf, Cyclictest, and pigpio should still be installed.
- If you began with a basic Linux system (instead of 'Lab3'):
    - Have a look at Part 1 of this lab to install cyclictest.
    - From Lab2 instructions, install perf_4.18
    - If you have saved your code to a cloud/usb location, load all of your code back onto the system.

One last configuration step:
Because of an issue with a USB driver, some of the testing on the PreemptRT kernel may cause the Rpi to freeze. Please apply the following changes to eliminate the impacts to testing. Add the following to the end of the file /boot/cmdline.txt

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

Note DO NOT use an 'enter' on this line anywhere….just let characters on the line wrap.

An example of cmdline.txt with the changes:

```
pi@jfs9-RPi3:~ $ cat /boot/cmdline.txt
dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=/dev/mmcblk0p7
rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait fbcon=map:10
fbcon=font:VGA8x8 splash plymouth.ignore-serial-consoles dwc_otg.fig_enable=0
dwc_otg.fig_fsm_enable=0 dwc_otg.nak_holdoff=0
```

Once these changes are in place, reboot the RPi

Important: **Backup the SD card. At this point, the Preempt RT patched kernel is installed.**

After the backup is completed, you can reboot the Raspberry Pi with the new Preempt-RT kernel. Just to review what was accomplished:

1. The 5.4.72 Linux source code was loaded to /home/pi/linux
2. The 5.4.70 Preempt-RT patch was downloaded and applied to the source code
3. The modified kernel code was compiled to generate a new Linux kernel image
4. The new Linux kernel image was installed, along with necessary support code and system modifications.

At this point, the new Linux Kernel is up and running and backups were made at each development step.

Since compilation is completed, the /home/pi/linux directory will not be used beyond this point (in step 4, above, the result of all this work, the new Linux kernel was installed). As an optional step, you may remove the /home/pi/linux directory which will free up considerable space on the SD card. Note that you can recover this directory by loading a previously backed-up version of your Lab4 work.

Performance of the new System

Once all of these tools and codes are loaded. Let's move on to performance of the system.

First, try cyclictest:

Note: since a new kernel was installed, /usr/bin has been updated. Check the cyclictest instructions from Lab4, week1 if you must reinstall cyclictest in the new system.

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

*A*dd 'dash h' to display the histogram of the run

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

Again, note the results and save the output of the histogram to create a chart in excel. If all is well, the maximum latency should be lower than in the cyclic test run under the Raspbian kernel.

You should also test the behavior of cyclictest while loading the RPi cores during the test. Note that sort_v1.c makes a pretty good cpu loading test. Compare a histogram of a loaded and unloaded RPi, with the RT Kernel.

For example, run 20 – 30 copies of sort_v1 in the background and repeat the command:

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

Use htop to make sure the cores remain busy for the duration of the cyclictest run. You might need to restart sort_v1 programs during the run. You might also consider modifying sort_v1 to run a bit longer.

Perf tests

Re-run the perf tests on the two sort_v1 C codes (unsorted and sorted versions) and note the results. Compare these tests against the sort_v1 tests run under the Raspbian kernel. Are there any changes of note?

Square Wave Tests

The C code square wave generator will be quite a bit different. As noted in class, to get the most out of the PreemptRT kernel, we have to place some controls on memory and stack space. Sample code on the ECE5725 server shows a skeleton of a routine to be used for the PreemptRT C code square wave generator (test_rt_skel_v16.c) Some notes about this code:

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

We pick a priority for the process that is not zero. As we saw in class, this will allow this application to preempt most other processes (including Linux processes).

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

This definition sets the size of the stack that the application is allowed to use.

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

These statements set the scheduling policy for the test to 'Sched_FIFO' with a priority of 49.

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

These statements lock 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 delay.

The for loop used for delay in the code uses will keep the code 'resident'; it will not be de-scheduled as it would if waiting for a timer. After the delay, the code will 'do something'.

Using this skeleton, modify the code so that:

- pigpio is used to access the GPIO pins.
- add code to toggle a GPIO pin (use the same pin as in part 1 of this lab).
- Check code operation with the LED.
- Remember to compile the code with the correct flags for pigpio (se week 1).

Once this is working, name the code test_rt_v16.c Generate a stable square wave, checking it with PiScope. When running this code:

- This code uses a for loop as a delay mechanism rather than nanosleep (have a look at test_rt_skel_v1.c to see the nanosleep version). The for loop keeps the code 'resident' when running using sched_FIFO in the RT kernel.
- You can select the interval with a command line argument therefore, change the interval of the for loop without a recompile.
- Try reducing the interval to smaller values; what frequencies do you see on PiScope?
- Recall that for blink.py (python code) and blink_v7.c (C code) you discovered a performance 'knee' at the upper frequency bound. Do you observe a similar performance knee using test_rt_skel_v7?
- To test a stable, higher frequency, run 'test_rt_v16 730'
    - This will run the test with an interval of 730 for the for loop
    - It should give you a stable frequency around 100 kHz
- Load the cores (with sort_v1, for example). Measure the performance difference of the square wave with a fully loaded system.

Final square wave tests:

For these tests, you will be exploring the features of the Preempt-RT kernel. For each of the steps, record the scheduling algorithm and priority for each of the tests. Summarizing these in a table, with associated notes about each test, would be a good way to record results.

1. Run test_rt_v16 without any system loading (as above) to generate a stable square wave.
    a. Run 'ps –alef | grep test_rt_v16' and note the pid of the running test_rt_v16
    b. Run 'chrt –p pid' where pid = pid of test_rt_v16 from the command above. Record results.
2. Run test_rt_v16 730 with system loading (I used sort_v1 for system loading, running it 20 times in the background). Comment on any impacts to the square wave generated from test_rt_v16.
    a. Run 'ps –alef | grep sort_v1' and note the pid of the running loading test (sort_v1 for my experiment)
    b. Run 'chrt –p pid' where pid = pid of the loading test from the command above. Record results.
3. Run the C code 'blink_v7' from the first week of lab to generate a stable square wave
    a. Run 'ps –alef | grep blink' and note the pid of the running blink test
    b. Run 'chrt –p pid' where pid = pid of the blink test from the command above. Record results.
4. Run 'blink_v7' to generate a stable square wave and add system loading. Comment on any impacts on the output of blink_v7 under loading.

Week 2 Checklist

Show the TAs a copy of the following

- uname –a results for the PREEMPT-RT kernel
- saved data from cyclictest on the PREEMPT-RT kernel
- perf results for the two sort programs
- test_rt_v16, blink_v7 code running with and without system loading
- results of chrt commands from Final square wave tests'
- Backups: Lab4 PreemptRT compiled, Lab4 PreemptRT Installed

Appendix A: raspi-config instructions from Lab 1

After having a brief look at the system, continue with the following install steps:

- issue the command 'sudo raspi-config'
- In the 'boot options' select 'Desktop/CLI' then 'B1 Console'

This change will boot directly to the console window on the next RPi reboot. Since you are already in raspi-config, continue with the notes below for more configuration.

Working your way through the config screens, please make sure to apply the following settings:

- Change the pi user password to something you will remember!
- In 'Network Options', select 'Hostname': set a unique hostname to 'netid_netid' where 'netid_netid' are the Cornell netids of the two team members in your group.
- For Localization options:
    - Change locale:
        - Use the Spacebar and tab to navigate
        - De-select 'en_GB.UTF-8 UTF-8'. Set the Locale 'en-us.UTF8 UTF8'. When prompted, select 'None' for the Default Locale
    - Change Timezone
        - Set the local time to America
        - Set the location to 'New York'
    - Change Keyboard Layout
        - 'Generic 101 key PC' keyboard
        - select 'Other', then 'English US'
        - 'configuring keyboard-configuration'
            - select 'English (US)'
            - next, select 'The default for the keyboard layout'
            - next, select 'no compose key'
            - finally, select "Ctrl-Alt-backspace to exit the X server"

Why these settings? These are the best settings for the time, keyboard and mouse you will be using

- Interfacing Options: enable ssh
- In the ‘Advanced’ settings
    - A1, expand the file system
    - A4 Audio: Force 3.5 mm jack
- Update: update this tool to the latest version

Once you hit ‘Finish’, raspi-config will save your changes and ask if you want to reboot now. Answer ‘yes’.

Appendix B: Perf Installation from Lab 2

Step1: Install Perf with the following commands:

```
sudo apt-get install linux-tools
```

Try the command:

```
perf –help
```

And it should fail. The message will tell you that your system is missing the correct version of perf. Example:

```
pi@RPi-jfs9:~ $ perf --help
/usr/bin/perf: line 13: exec: perf_4.14: not found
E: linux-perf-4.14 is not installed.
```

You can remedy this by running:

```
sudo apt-get install linux-perf-4.9
```

Step2:

Test that perf is operating as designed by running:

```
perf_4.9 --help
perf_4.9 list
perf_4.9 --version
```

Notes:

1. Perf is able to track a number of different hardware and software events. Take a look at the output of 'list', above, to get an idea of these events perf is able to track.
2. Keep in mind that not all of these measurements are available on all platforms.
3. With perf, there is a bash shell installed in /usr/bin/perf. Run this by issuing the command 'perf'. What error do you see? Take a look at the bash script and see if you can form a theory for why this fails and how you might overcome the problem.

Appendix C: Using piscope for python and c-codes

Python code:

```
sudo pigpiod    # start the pigpio daemon
cd PISCOPE
./piscope  # start piscope
```

C-Code:

If you use the pigpio C library within a C program, simply running the program will launch the pigpiod function. So, there is no need to start pigpio independently.

However, the C-code will NOT run if there is an instance of pigpiod currently running. If pigpiod is running the following sequence can be used:

```
sudo killall pigpiod   # to exit the pigpiod daemon

sudo ./blink_v7    # to run your c-code using the pigpio library
                   # NOTE that a C-code with pigpio requires sudo to run

/home/pi/PISCOPE/piscope   # to start piscope
```

Note that piscope will run correctly for this one instance of the C-code. Since the C-code executes the pigpiod function, once the code finishes, the pigpiod function ends and piscope will no longer run correctly. For each run of your C-code, follow the sequence:

```
sudo ./blink_v7     # to run your c-code using the pigpio library
/home/pi/PISCOPE/piscope   # to start piscope
```

to see correct results from piscope.