CSE 237A Winter 2018: Individual Project

Part 1 of 2: Environment setup and workload characterization
Detailed Instructions

During the first part of the project, you will setup the development environment for RPi 3, and then measure its performance. The purpose of this document is to help you to prepare your system for the project and measure performance counters by:

- Installing the required tools
- Downloading and compiling the kernel source code
- Cross-compiling a sample C program and a kernel module
- Enabling PMU and ensuring you can read/write values from registers
- Analyzing the performance of the provided workload

If any step quits prematurely and shows an error message, log the error and read it. Some common errors can be figured out with a quick web search or by posting to class Piazza page. To help us debug the problem, include the error message and output of the following commands

- `uname -a`
- `lsb_release -a`

Common issues:

- Be aware of dashes/spaces/newlines when copying from PDF. When in doubt, try typing out the whole command manually.
- Run "make mrproper" and "make clean" before reconfiguring and recompiling a kernel.

Follow-up for common error messages (in particular "file not found" / "command not found")

- **Google your error first**
- Are you on the right machine?
- Are you in the right directory?
- Is your PATH variable set correctly? (In ~/.bashrc?)
- Did you need sudo?
- If you're trying to run a binary - is it executable?
- If you're trying to access an external device - is your external drive mounted correctly?
- Do you have the right packages/dependencies? (Especially if it's a *.h file that's missing)

Asking for help on Piazza:

- The best posts are public, include your last commands, the full error message, and a description of your hardware/OS environment. "It does not work" is not a sufficient post for us to help you. You must state clearly what you tried, and what indicates a failure (e.g. exact error message, LED lights, blank screen, etc).
- If someone makes a suggestion that works for you, *please report back!*
- If you figure it out on your own, *please report back!*
- "I solved it" or "I made a mistake" is usually not a helpful end to the discussion. Don't be embarrassed by your simple mistakes - we all make them. Please describe the solution and help the next person out!

1. Hardware and software requirements

- Raspberry Pi 3
- MicroSD card (8-32GB)
- MicroSD card reader (external USB reader highly recommended over a built-in one)
- Intel ISA, 64-bit machine (x86_64)
- Ubuntu 64bit 14.04.3 LTS desktop edition
    - You may run it as a virtual machine in VirtualBox.
    - For package installation commands, root access through sudo command is required.
    - If you are familiar with Linux and already have other Linux version (e.g., Ubuntu 16.04 LTS), you might get it to work by using additional repos/packages.
- ARM cross-compiler

2. Install Ubuntu

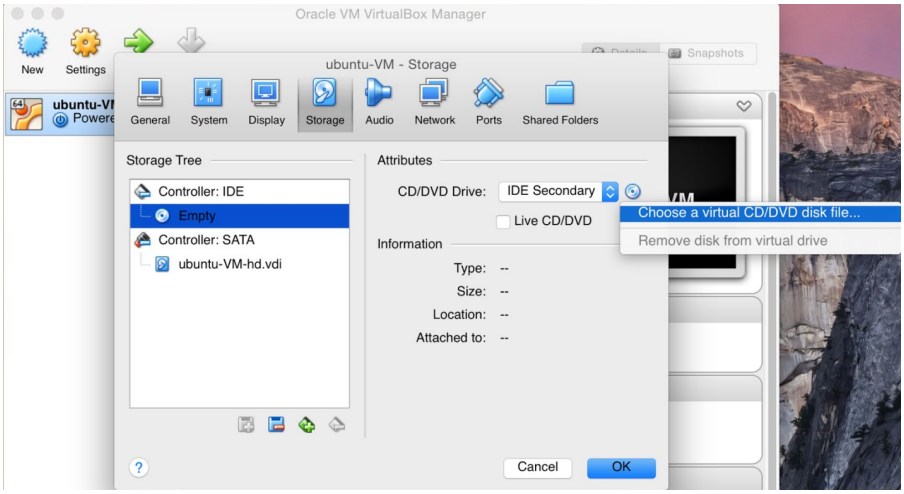
You may install Ubuntu 14.04 directly on your machine ("bare metal") or as a virtual machine (VM). If you are NOT familiar with Linux environment, we highly recommend using a VM. If you are already familiar with Linux and have your own Linux machine, you may use it instead.

Option 1: Bare metal instructions (for Windows):

Create a bootable USB stick: http://www.ubuntu.com/download/help/createausbstickonwindows
Follow official installation instructions: http://www.ubuntu.com/download/desktop/installubuntudesktop
Dualbooting Linux and OS X is not supported as a part of this class.

Option 2: VM instructions (Mac or Windows):

1. Download and install a VirtualBox platform package for your machine: https://www.virtualbox.org/wiki/Downloads. If you are using a USB 3.0 reader, download and install the extension pack as well
2. Download Ubuntu 14.04: From http://releases.ubuntu.com/14.04/ download "64bit PC (AMD64) desktop image" (a large ISO image almost 1GB)
3. Open VirtualBox
4. Click Machine > New…
5. Enter the following:
    - Name: ubuntuVM
    - Type: Linux
    - Version: Ubuntu (64bit)
6. Set a proper RAM size. For example, if your host machine has 4GB+ RAM, you may allocate 1GB to the virtual machine
7. Create a new virtual hard drive with “VDI format” and “Dynamically allocated” options. Allocate *at least 16 GB* for the hard drive size.
8. Leave other settings to their default values unless you know what you're doing.
9. Once the VM has been created, right click > "Settings" > "Storage"



10. Add the downloaded ISO image as a virtual disk file and click "OK"
11. Start the VM
12. Select "Install Ubuntu"
13. Take default options (including "Erase disk and install Ubuntu" this refers to the virtual disk)
14. After installation completes, repeat Step 8 and remove the ISO file (if any) so that the IDE controller list reads "empty" again.
15. Restart as prompted by the installer
16. *Install Guest Additions (optional):*

Guest Additions support many nice features for Ubuntu in VM, including a solution for screen resolution issue. To install the guest additions, click **"Devices" > "Insert Guest Additions CD image"** in the menu of the virtual machine window, and follow its install instruction. You need to reboot the VM after installing it to apply the changes.

If the default CD image for Guest Additions is not completely installed, you may try to install it from Debian packages instead. Open a terminal using CTRL+ALT+T in your VM, and then type:

```
$ sudo apt-get install virtualbox-guest-dkms virtualbox-guest-utils virtualbox-guest-x11
```

If it gives you an error message: "unmet dependency", you may try the following:

```
$ sudo apt-get remove libcheese-gtk23
$ sudo apt-get install xserver-xorg-core
$ sudo apt-get install virtualbox-guest-dkms virtualbox-guest-utils virtualbox-guest-x11
```

3. Build and install kernel for for RPi

Parts of this instruction manual are written based on the official guidelines:
https://www.raspberrypi.org/documentation/linux/kernel/building.md

3.1. Install git, curl, and libncurses

The download of the tool chain and source codes requires `git`. You can also download files (e.g., the sample C code provided on the class website) from the Internet using `curl`. You may also need to install libncurses which will be used to customize the kernel build. With root privilege, you can install them as follows:

```
$ sudo apt-get install git curl libncurses5-dev
```

You can download any required libraries and linux programs in a similar way. For example:

```
$ sudo apt-get install <ANY_PROGRAM_OR_LIBRARY>
```

3.2. Install toolchain

To compile the Linux kernel for RPi 3 on your machine (i.e., bare metal or VM), you require a suitable Linux cross-compilation toolchain - i.e. a compilation suite to make executable binaries for an ISA different from your host machine. Get tool chains by using the following command:

```
~$ mkdir ~/RPdev
~$ cd ~/RPdev
~/RPdev$ git clone https://github.com/raspberrypi/tools
```

Add the toolchain path to your PATH. You may put the line in your `~/.bashrc` to make it permanent.

```
~/RPdev$ export PATH=$PATH:~/RPdev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin
```

- If you use a 32-bit machine, you need to use `gcc-linaro-arm-linux-gnueabihf-raspbian`

3.3. Download kernel source

The kernel source for this project is customized for the low-level performance counter access we need. It is *different* from the standard kernel released with Raspbian. You can clone the repository using git as follows:

```
~/RPdev$ git clone http://seelab.ucsd.edu/~shepherd/cse237a_wi18_kernel.git linux
```

Now, the source code is downloaded in the "`~/RPdev/linux`" directory.

3.4. Build kernel source

To build the sources for cross-compilation, there may be extra dependencies (depending on your Ubuntu version) beyond those you've installed by default with Ubuntu. Enter the following commands to build the sources and Device Tree files for RPi 3:

```
~/RPdev$ cd linux
~/RPdev/linux$ KERNEL=kernel7
~/RPdev/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

The next step builds the kernel - it will take a while. To speed up compilation on multiprocessor systems, and get some improvement on single processor ones, use `-jn` where `n` is number of processors * 1.5 (try `-j4` or `-j8`):

```
~/RPdev/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

To check the compiled kernel images, use the following commands and compare results. (You may see a different sha1 hash value - it depends on the environment that the kernel is compiled in, not the result of the build.)

```
~/RPdev/linux$ file arch/arm/boot/zImage
arch/arm/boot/zImage: Linux kernel ARM boot executable zImage (little-endian)
~/RPdev/linux$ file vmlinux
vmlinux: ELF 32-bit LSB   executable, ARM, EABI5 version 1 (SYSV), statically linked,
BuildID[sha1]=eefb2cfc4f7f12eee56b8df4d5c815d18dbc2a29, not stripped
```

If you get the following error, some files may have been corrupted somewhere along the way:

```
net/netfilter/.nf_conntrack_acct.o.cmd:5: *** unterminated call to function `wildcard':
missing `)'.  Stop.
```

In that case, clean up the files before rebuilding:

```
~/RPdev/linux$ make mrproper
~/RPdev/linux$ make clean
```

3.5. Install kernel

Your RPi will the run the custom compiled kernel from a MicroSD card. In this section, we will flash a bootable operating system image onto the card, then replace the kernel with our custom kernel.

Step 1: Prepare SD card contents

You can start with either a premade NOOBS card that shipped with your Raspberry Pi, or any other SD card of 8-32GB. If you have a NOOBS card, skip to the next section. Otherwise:

1. Format your SD card: https://www.sdcard.org/downloads/formatter_4/ Choose "Overwrite format" option. (For Ubuntu users, you can use gparted)
2. Download RASPBIAN STRETCH WITH DESKTOP image (Release Date: 2017-11-29) from https://www.raspberrypi.org/downloads/raspbian/
3. Burn the image onto the SD card using Etcher https://etcher.io/

- If you need more information, visit the official RP site: https://www.raspberrypi.org/documentation/installation/noobs.md

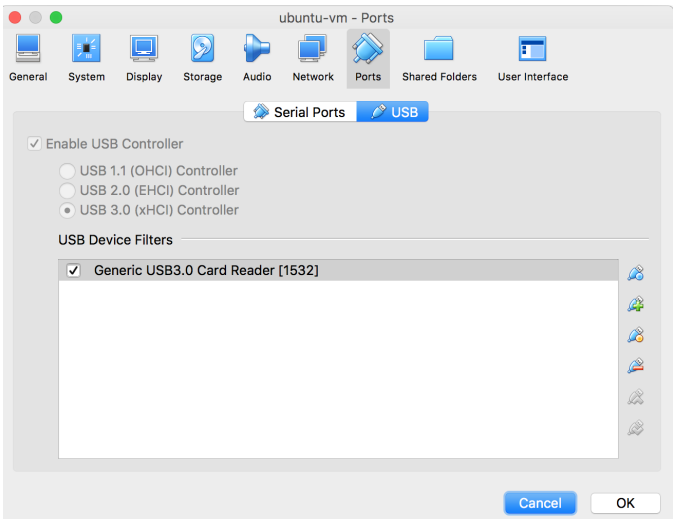
Step 2: Forward the SD card to your VM

If you are using a VM, you need to forward the SD card reader to VM. The device names shown in the images below may vary from machine to machine.

- If you are using the built-in SD card slot of a recent MacBook Pro, you need to forward a raw disk device since it is not treated as a USB device. http://superuser.com/questions/373463/how-to-access-an-sd-card-from-a-virtual-machine

In most cases for VirtualBox with a USB SD card reader (Windows or Mac), the instructions are as follows:

1. Plug in your USB reader but remove the SD card
2. Shutdown your VM (i.e., the installed Ubuntu)
3. Open the VirtualBox application (not the VM). Go to Settings → Ports → USB
4. Add a USB filter for the USB SD card reader (the "+" button on the right side of the dialog)
5. Close the dialog, and start the VM again
6. Once booting is completed, plug the SD card in the reader
7. Type the following command in the terminal to make sure all the OS is aware of new partitions loaded.
   `$ sudo partprobe`



Step 3: Install the kernel

Once the SD card is properly detected by Ubuntu, check the partitions:

```
~/RPdev$ lsblk
```

```
sdb      8:32   1   7.5G  0 disk
├──sdb1   8:33   1   1.2G  0 part
├──sdb2   8:34   1     1K  0 part
├──sdb5   8:37   1    32M  0 part
├──sdb6   8:38   1    66M  0 part
└──sdb7   8:39   1   6.2G  0 part
```

These partition names might be different from what you get depending on your VM or machine (e.g. you may see **sdc** instead of **sdb**). In the above case, it shows the partitions of a purchased NOOBS card. **sdb6** and **sdb7** are the target partitions that the kernel will be installed. If you are using your own SD card, they will probably show up as **sdb1** and **sdb2**:

```
~/RPdev$ lsblk
sdb      8:16   0  29.7G  0 disk
├──sdb1   8:17   0    63M  0 part
└──sdb2   8:18   0   1.2G  0 part
```

Mount the PROPER partitions. You may have to type **sdb1** and **sdb2** in place of **sdb6, sdb7**, respectively.

```
~/RPdev$ mkdir mnt
~/RPdev$ mkdir mnt/fat32
~/RPdev$ mkdir mnt/ext4
~/RPdev$ sudo mount /dev/sdb6 mnt/fat32     # sudo mount /dev/sdb1 mnt/fat32
~/RPdev$ sudo mount /dev/sdb7 mnt/ext4      # sudo mount /dev/sdb2 mnt/ext4
```

Next, install the compiled modules and the new kernel and make it bootable. Enter the second command "`sudo...`" as one line and make sure there is a space before "`INSTALL_MOD_PATH`".

```
~/RPdev$ cd linux
~/RPdev/linux$ sudo env "PATH=$PATH" make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_PATH=../mnt/ext4 modules_install
~/RPdev/linux$ sudo cp arch/arm/boot/zImage ../mnt/fat32/kernel-cse237a.img
~/RPdev/linux$ sudo cp arch/arm/boot/dts/*.dtb ../mnt/fat32/
~/RPdev/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* ../mnt/fat32/overlays/
~/RPdev/linux$ sudo cp arch/arm/boot/dts/overlays/README ../mnt/fat32/overlays/
```

Then, change the configuration so that the system uses the installed "kernel-cse237a.img". Edit "../mnt/fat32/config.txt" using a text editor, e.g., vi or nano or gedit. You need root privileges. For example, if you're using vi:

```
~/RPdev/linux$ sudo vi ../mnt/fat32/config.txt
```

In the opened file, add the line **"kernel=kernel-cse237a.img"**. Replace the line "`kernel=kernel.img`" or "`kernel=kernel7.img`" if it exists.

Finally, unmount all the partitions.

```
~/RPdev/linux$ sudo umount ../mnt/fat32
~/RPdev/linux$ sudo umount ../mnt/ext4
```

Now, you can plug the microSD card (as well as your keyboard, mouse, and monitor) into RPi and turn on its power! To check if the OS boot was successful with your new kernel, open the terminal (click the icon on the menu bar), and type the following in the RPi to check the build date.

Make sure that it shows the date and time when you compiled the kernel.

```
$ uname -a
Linux raspberrypi 4.9.67-v7+ #2 SMP <BUILD_DATE_TIME> armv71 GNU/Linux
```

Before moving to the next section, we recommend updating firmware and drivers. With a valid internet connection, run this command (may take a while):

```
$ sudo rpi-update
```

Lastly, the RPi will automatically boot into a graphical user interface (GUI) with XServer.
If you'd like to boot into the command line next time (so you don't need a mouse), change your options with `raspi-config`:

```
$ sudo raspi-config
→ 3 Boot Options
→ B1 Desktop / CLI
→ B1 Console
```

Troubleshooting and hints

- Most common problems: USB devices, unstable power supply. More info: http://elinux.org/R-Pi_Troubleshooting#Power_.2F_Start-up
- You may need to change your keyboard settings to "English (US)", using `raspi-config`
- We recommend to use either a wired ethernet or a stable wireless connection. If you want to use "UCSD-PROTECTED", the default GUI-based interface for networking may not be able to connect it, since it does not support WPA enterprise protocol. You can find other ways from Google search, but here is one workaround (although it might be sometimes unstable):
    - Modify /etc/network/interfaces

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Please note that this file is written to be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf and 'man dhcpcd.conf'
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d
auto lo
iface lo inet loopback
iface eth0 inet dhcp
allow-hotplug wlan0
iface wlan0 inet dhcp
      pre-up wpa_supplicant -B -Dwext -i wlan0 -c/etc/wpa_supplicant/wpa_supplicant.conf
      post-down killall -q wpa_supplicant
allow-hotplug wlan1
```

    - Modify /etc/wpa_supplicant/wpa_supplicant.conf (You need to put your username and password)

```
ctrl_interface=/var/run/wpa_supplicant
network={
      ssid="UCSD-PROTECTED"
      scan_ssid=1
      key_mgmt=WPA-EAP
      pairwise=CCMP TKIP
      group=CCMP TKIP
      eap=PEAP
      identity="UCSD_AD_USERNAME"
      password="UCSD_AD_PASSWORD"
      phase1="peapver=0"
```

```
    phase2="MSCHAPV2"
}
```

- Reboot the system and open the browser to check the connection (the network icon may not work appropriately with this patch)

- `cp: error writing '../mnt/fat32/bcm2709-rpi-2-b.dtb': Input/output error`
  - Did you mount the partitions twice? (It might have automatically mounted your SD card as something other than mnt/*) Are you out of space? Try some combination of the following: unmount partitions, remove the card, restart the VM and try again.
- `cp: target '../mnt/fat32/overlays/' is not a directory`
  - Did you mount the partitions correctly?

4. Cross-compile user applications

4.1. Download the sample C application

Sample C code is provided on the class website. Download it into the "~/RPdev/test" directory.

```
~/RPdev$ mkdir test
~/RPdev$ cd test
~/RPdev/test$ curl -O http://cseweb.ucsd.edu/classes/wi18/cse237A-a/project/part1/test.c
```

```
#include <stdio.h>

int main() {
       printf("Hello RP World!\n");
       return 0;
}
```

4.2. Compile the application

Cross compile the given source code for the RPi target:

```
~/RPdev/test$ arm-linux-gnueabihf-gcc -o test.out test.c -static
```

The result should be the executable: "test.out".

4.3. Verify the executable

Verify the file type of the cross-compiled output "test.out"

```
~/RPdev/test$ file test.out
test.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, for GNU/Linux 2.6.26, BuildID[sha1]=16de6d4ff05818aa6eb01d84f042113c298f1e74, not stripped
```

The description of "ELF 32-bit LSB executable" and "ARM" indicates that your binary is ready to be executed on the ARM processors of RPi. Your hash value does not have to match the example.

4.4. Check if it really works on RPi

First, copy the file to RPi using either a USB pen drive or Internet (e.g., dropbox, mail, ssh, etc). If you're using a USB pen drive, you have to mount the USB drive in RPi after connecting it:

```
$ sudo mkdir /media/usbdrive
$ sudo mount /dev/sda1 /media/usbdrive
$ cp /media/usbdrive/test.out ~/
```

Then, execute the file:

```
$ cd ~/
$ chmod a+x test.out
$ ./test.out
Hello RP World!
```

5. Build custom kernel modules

A kernel module is an object file that contains procedures to extend the running kernel. You need to develop a kernel module to read performance counters. If you are NOT familiar with kernel modules, we highly recommend visiting the following link: Linux Kernel Module Programming Guide: http://www.tldp.org/LDP/lkmpg/2.4/html/c147.htm

Let's start with a simple kernel module. Use your Ubuntu host to run all kernel module builds. Source files can be found at http://cseweb.ucsd.edu/classes/wi18/cse237A-a/project/part1/sample_module/

5.1. Compile *hello world* kernel module for host machine

Create a kernel module file: hello.c

```
/*
 *  hello.c - The simplest kernel module.
 */
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "CSE237A: Hello world.\n");
    return 0;
}

void cleanup_module(void)
{
    printkh(KERN_INFO "CSE237A: Goodbye world.\n");
}
```

Generate a file named `'Makefile'`

```
obj-m += hello.o
PWD := $(shell pwd)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Build the kernel module using the Makefile.

```
$ make
```

Make sure the whitespace in your Makefile are tabs, NOT spaces, otherwise you will get this error:
`make: Nothing to be done for `all'.`

Check module information with the following command:
`$ modinfo hello.ko`

This kernel module can run only on your host architecture, but not on the Raspberry Pi's architecture (ARMv8). The vermagic prefix (3.13.0.107 in the example below) is the kernel version of your host machine. This will vary depending on your machine configuration.

```
$ modinfo hello.ko
filename:     hello.ko
srcversion:     B32527F71C81E36B88FEED8
depends:
vermagic:       3.13.0-107-generic SMP mod_unload modversions
```

Now load this kernel module. To load your kernel module, use the command "insmod" with root privilege (sudo). To unload your kernel module, use "rmmod".

```
$ sudo insmod hello.ko  # load hello.ko
$ sudo rmmod hello       # unload hello.ko
```

You can check the result of the kernel module with "dmesg" command. The function "`printk`" generates a log for the kernel module. Thus, you can see "CSE237A: Hello world." and "CSE237A: goodbye world." messages with dmesg command. To see the last log lines, use "tail" command as follows:

```
$ sudo insmod hello.ko
$ sudo rmmod hello
$ dmesg | tail -2
[178474.908145] Hello world.
[178479.783346] Goodbye world.
```

5.2. Cross-compile *hello world* kernel module for Raspberry Pi

gcc is the compiler on the host machine specific to its architecture, an Intel-based machine. RPi is an ARM-based machine. You need to specify the architecture type and the ARM-based compiler from the previously-downloaded toolchain to use the Makefile. Create a new directory and copy the hello.c file there. Then, write the following Makefile in that same directory. (You have to give a proper path for KDIR.)

```
obj-m := hello.o
KDIR := /home/YOUR_ACCOUNT/RPdev/linux   # Your kernel source directory
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KDIR) M=$(PWD) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) ARCH=$(ARCH) clean
```

Then, run make to generate a kernel module for the appropriate target architecture and compiler.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

This will generate hello.ko kernel module. Check the module information. This module is compatible with ARMv7 (which your RPi's ARMv8 is backwards compatible with) and is compiled using the provided Linux kernel source for the Raspberry Pi 3.

```
$ modinfo hello.ko
```

```
$ modinfo hello.ko
filename:       /home/pi/hello.ko
srcversion:     B32527F71C81E36B88FEED8
depends:
vermagic:       4.9.67-v7 SMP mod_unload modversions ARMv7
```

Check the kernel number on RPi.

```
$ uname -a
```

If you correctly finished the custom kernel installation in the previous section, you should see:

```
Linux raspberrypi 4.9.67-v7+ #1 SMP <BUILD_DATE_TIME> armv71 GNU/Linux
```

Compare the kernel version with that of hello.ko - these should match. You downloaded/built the kernel source and installed it onto your RPi. This means that both the running kernel and the kernel source have the same configuration.

Copy the kernel module ("hello.ko") to RPi. Use ssh or or copy to a drive. hello.ko should run without any errors. Test your install on RPi by installing and uninstalling the kernel module to get the messages shown in the box below.

```
$ sudo insmod hello.ko
$ sudo rmmod hello
$ dmesg | tail -2
```

```
[  249.740357] CSE237A: Hello world.
[  273.468906] CSE237A: Goodbye world.
```

If you see errors during the install/uninstall process, it may come from a kernel compatibility issue. Even though the running kernel and the kernel source have the same numeric value, the kernel module may not be loaded if the two are using different configuration options. This may result in invalid module format error.

You can check the kernel source version on your VM that you compiled using the kernel image file (vmlinux). Move to the kernel source directory and type

```
$ make kernelversion
4.4.41
```

The version must be same as that of the running kernel of RPi ("`uname -a`" command). If those numbers are different, it means that you didn't successfully install the kernel image. Carefully follow the installation instruction again, and reinstall kernel image. You may refer to the official website:
https://www.raspberrypi.org/documentation/linux/kernel/building.md.

6. Checkpoint Demo Deliverables

Make sure that your RPi 3 ~~is using a single core and~~ can run your sample kernel module. Download the following program binary and run it on your RPi 3 before coming to the demo.

Check if the program runs correctly:

```
$ wget http://cseweb.ucsd.edu/classes/wi18/cse237A-a/project/part1/check_kernel
$ chmod a+x ./check_kernel
$ sudo ./check_kernel
SUCCESS 0x0EBD237A
```

If it prints “FAIL”, your kernel has not been updated correctly. Please make sure that you update your RPi using the provided kernel source. If it prints nothing on the terminal, reboot your system.

Bring your RPi with the compiled and tested sample kernel module to CSE 3219 during Checkpoint 1 for us to test.

7. Project Part 1 - Performance Monitoring Unit (PMU)

The processor (ARM Cortex-A53) includes logic to gather various statistics on the operation of the processor and memory system during runtime, based on the PMU architecture. These events provide useful information about the behavior of the processor that you can use when debugging or profiling code. The processor PMU provides six event counters. Each event counter can measure any of the events available in the processor. Thus, you can access the following counter registers:

- One cycle counter that increments based on the processor clock.
- Six 32-bit counters that increment when the corresponding event occurs.

To access the register, specify CRn, CRm, and Op2 of each event. We use the PMU functionalities with CRn = c9 (the "c9 register"). The following table shows the 32-bit wide system control registers.

Table 4.140. c9 register summary

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c12	0	PMCR	0x41033000	*Performance Monitors Control Register*
			1	PMNCNTENSET	UNK	Performance Monitors Count Enable Set Register
			2	PMNCNTENCLR	UNK	Performance Monitors Count Enable Clear Register
			3	PMOVSR	UNK	Performance Monitor Overflow Flag Status Clear Register
			4	PMSWINC	UNK	Performance Monitors Software Increment Register
			5	PMSELR	UNK	Performance Monitors Event Counter Selection Register
			6	PMCEID0	0x67FFBFFF[a]	*Performance Monitors Common Event Identification Register 0*
			7	PMCEID1	0x00000000	*Performance Monitors Common Event Identification Register 1*
		c13	0	PMCCNTR	UNK	Performance Monitors Cycle Counter
			1	PMXEVTYPER	UNK	Performance Monitors Selected Event Type and Filter Register
			2	PMXEVCNTR	UNK	Performance Monitors Selected Event Counter Register
		c14	0	PMUSERENR	0x00000000	Performance Monitors User Enable Register
			1	PMINTENSET	UNK	Performance Monitors Interrupt Enable Set Register
			2	PMINTENCLR	UNK	Performance Monitors Interrupt Enable Clear Register
			3	PMOVSSET	UNK	Performance Monitor Overflow Flag Status Set Register
	1	c0	2	L2CTLR	-[b]	*L2 Control Register*
			3	L2ECTLR	0x00000000	*L2 Extended Control Register*

Source: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500g/BABFIBHD.html

The detailed field descriptions of these registers (e.g. PMCR) are publicly available in the online ARM technical manual. Others (e.g. PMSELR, PMOVSR) are unavailable online - the relevant pages from the ARM architecture manual have been uploaded to the project website:
http://cseweb.ucsd.edu/classes/wi18/cse237A-a/project/part1/arm/

The skeleton kernel module code, “pmuon.c” is provided on the class website in “pmuon” folder, and also copied below. It uses c9 registers to initialize the PMU.

[pmuon.c]

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("PMUON");

void set_pmu(void* dummy) {
    unsigned int v;
    printk("Turn PMU on Core %d\n", smp_processor_id());

    // 1. Enable "User Enable Register"
    asm volatile("mcr p15, 0, %0, c9, c14, 0\n\t" :: "r" (0x00000001));

    // 2. Reset i) Performance Monitor Control Register(PMCR),
    // ii) Count Enable Set Register, and
    // iii) Overflow Flag Status Register
    asm volatile ("mcr p15, 0, %0, c9, c12, 0\n\t" :: "r"(0x00000017));
    asm volatile ("mcr p15, 0, %0, c9, c12, 1\n\t" :: "r"(0x8000003f));
    asm volatile ("mcr p15, 0, %0, c9, c12, 3\n\t" :: "r"(0x8000003f));

    // 3. Disable Interrupt Enable Clear Register
    asm volatile("mcr p15, 0, %0, c9, c14, 2\n\t" :: "r" (~0));

    // 4. Read how many event counters exist
    asm volatile("mrc p15, 0, %0, c9, c12, 0\n\t" : "=r" (v)); // Read PMCR
    printk("We have %d configurable event counters on Core %d\n",
            (v >> 11) & 0x1f, smp_processor_id());

    // 5. Set six event counter registers (Project Assignment you need to IMPLEMENT)

}


int init_module(void) {
    // Set Performance Monitoring Unit (aka Performance Counter) across all cores
    on_each_cpu(set_pmu, NULL, 1);
    printk("Ready to use PMU\n");
    return 0;
}

void cleanup_module(void) {
    printk("PMU Kernel Module Off\n");
}
```

In the module initialization, we call the `set_pmu()` function on each core using `on_each_cpu()`. Thus, in the RPi3, the function is called four times to set the same PMU registers across all cores.
The c9 registers are accessed using inline assembly code (`asm volatile`). Two instruction types are used: “`mcr`” to write a register value, and “`mrc`” to read a register value. In each instruction, operands specify the target register. See the following link if you need additional information:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489g/Cihfifej.html

The skeleton code accomplishes the following:

1. Linux kernel does not allow a user-space program to access PMU without authorization. In order to access the PMU from the user-space program, set the "user enable bit" to '1' in the **User Enable Register** (PMUSERENR).
   Here is the "`mcr`" assembly line which writes a register value:

```
asm volatile("mcr p15, 0, %0, c9, c14, 0\n\t" :: "r" (0x00000001));
```

   From Table 4.10.c9, you can find that the {CRn, CRm, Op2} of PUSERNENR are {c9, c14, 0}.
   These three variables are the last three parameters of the "`mcr`" assembly instruction.
   "`0x00000001`" corresponds to the placeholder parameter "%0" in the asm volatile function call.
2. Configure the **Performance Monitor Control Register** (PMCR) to initialize all counters. Initialize the **Count Enable Set Register** (PMNCNTENSET) and **Overflow flag status register** (PMOVSR) in a similar way using their {CRn, CRm, Op2} values.
3. Disable the **Interrupt Enable Clear Register** (PMINTENCLR). The purpose of this register is to determine if any of the Performance Monitor Count Registers and the Cycle Count Register generate an interrupt on overflow. Any interrupt overflow enable bit written with a value of 1 clears the interrupt overflow enable bit to 0.
4. Read PMCR to know how many configurable event counters exist. The "`mrc`" assembly instruction is used for reading the register value into the "v" variable.

Cross-compile "pmuon.c" to generate the kernel module. Then, copy it to RPi 3. Then, test by loading the kernel module.

```
$ su insmod pmuon.ko
$ dmesg | tail -15
```

```
Turn PMU on Core 2
Turn PMU on Core 1
Turn PMU on Core 3
We have 6 configurable event counters on Core 1
We have 6 configurable event counters on Core 3
We have 6 configurable event counters on Core 2
Turn PMU on Core 0
We have 6 configurable event counters on Core 0
Ready to use PMU
```

In the dmesg log you should see the messages printed from each core. The CPU has 6 configurable event counters.

7.1. Assign events to performance counter registers in the kernel module

Table 12.28 in the ARM Cortex-A53 manual shows all available events, and their event type IDs:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500g/BIIDBAFB.html

Modify the skeleton code ("pmuon.c") to assign the following events to the six event counters:

- First event counter: # of instructions architecturally executed
- Second event counter: # of L1 data cache access
- Third event counter: # of L1 data cache miss (refill)
- Fourth event counter: # of L2 data cache access
- Fifth event counter: # of L2 data cache miss (refill)
- Sixth event counter: # of L1 Instruction TLB miss (refill)

(Note: The L2 cache is also LLC (last level cache) in the RPi3 system.)

We set the six event counters using the "Event Counter Selection Register" (PMSELR) and the "Event Type Selection Register" (PMXEVTYPER) in the kernel module.

1. **Event Counter Selection Register** (PMSELR)
    a. This register selects the event counter from which we will read.
    b. It should be set before setting an event type (PMXEVTYPER)
    c. You can select the event counter by setting one of the following bits:
    0x0 : First event counter
    0x1: Second event counter
    0x2: Third event counter
    0x3: Fourth event counter
    0x4: Fifth event counter
    0x5: Sixth event counter
2. **Event Type Selection Register** (PMXEVTYPER)
    a. After setting the PMSELR, specify the event type to count and read with event count registers. After selecting it, we can access its actual counter value in the user-space program using PMXEVCNTR (You need to do this in the next section, 7.2.)

Submit your modified "pmuon.c" source code to select the six events for the event counters.

7.2. Read performance counter events in the user-space program

You will read the values of the six event counters during some workloads given in an user-space program. Download the user-space program in the archive "workload_analysis.tar.gz", and test if it is successfully compiled using the provided Makefile. For example, you can compile the provided program as follows in your linux system:

```
$ tar zxvf workload_analysis.tar.gz
$ cd workload_analysis
$ make CROSS_COMPILE=arm-linux-gnueabihf-
```

This creates an user-space program file, `main_part1.` Copy the compiled kernel module and the user-space program binary into the RPi 3. Then run the following:

```
$ sudo insmod pmuon.ko
$ sudo chmod a+x main_part1
$ sudo ./main_part1
Initialization.
Workload 1 is ready.
Characterization starts.
Total Execution time (us): 947514 at 1200000
[Core 0] Execution Time (us): 947140
[Core 0] Cycle Count: 1136569046
[Core 0] Instructions: 0
[Core 0] L1 Cache Accesses: 0
[Core 0] L1 Cache Misses: 0
[Core 0] LLC Accesses: 0
[Core 0] LLC Misses: 0
[Core 0] iTLB Misses: 0
$ sudo rmmod pmuon
```

Once you install the part 1 kernel module, the performance counter registers can be accessed in any userspace program executed with root privilege - don't forget to type **"sudo"** when running `main_part1`. But, as seen above, the provided program does not read the performance counters, reporting all zeros, other than the cycle counter. So, you need to implement code which measures the six performance counter events.

Before diving into that, here is the description of what the provided user-space program actually performs.
The provided "main()" function in "main.c" performs the following tasks:

1. It first sets workloads to be executed:
    a. There are three workloads that need to be characterized. You can register a workload on a specific core using "`register_workload()`" function. The original code includes an example for this function. For example, if you want to execute two workloads on core 0 and 1 (among 4 availables), you can do this:

```
register_workload(0, workload1_init, workload1_body, workload1_exit);
register_workload(1, workload2_init, workload2_body, workload2_exit);
```

    b. This *register* function calls `workload*_init()` function to prepare relevant variables, e.g., array. The other functions, e.g., `workload1_body() and workload1_exit()`, will be executed later.
    c. You can find the source code of the first and second workloads in the "workload_known.c" file. You can execute the third workload (workload 3) provided in "workload_unknown.o", but the source code of the third workload (workload 3) is *unknown*. That is, you need to characterize the workload without the source code, and discuss the workload behavior in terms of PMU measurement in the report.
2. It sets the operation frequency of the CPU:
    a. "`set_userspace_governor()`" changes the CPU governor to "userspace". When the system uses the userspace governor, a user program can change the CPU frequency. In RPi3, all four cores share the same frequency.
    b. "`set_by_max_freq()`" changes the CPU frequency to the maximum value. You may alternatively use "`set_by_min_freq()`". RPi 3 supports two available CPU frequency settings. The two functions switch between the 1.2GHz and 600MHz frequency.
3. Then, the `run_workload()` function will execute all registered workloads on the designated cores, while measuring the PMU events for each core. For each core, the following jobs are executed in order:
   (see `run_and_profile()` function in "workload_util.c"):
    a. Internally, this first calls "`reset_counter()`" to reset the cycle counter and the six event counters selected in the kernel module to zero by using PMCR.
    b. Then, this performs the registered `workload*_body()` function.
    c. At the end of the body function, this calls your **get_single_event**() function to collect the PMU measurements.
    d. You can find the details of these workload execution code in "workload_util.h/.c".
4. The code measures the execution time of the entire workload, and prints measured PMU counters for each core that runs the given workload.
5. After measuring the execution time, "`unregister_workload_all()`" frees all the used data structures by calling `workload*_exit()`, and it changes the governor to the default "ondemand" policy.

Now, you **need to implement** the code that measures the six configurable event counters by modifying "get_single_event()" function in the provided "pmu_reader.c". The skeleton of the function is given as follows:

```
// Fill your code in this function to read an event counter.
// One of the following numbers will be given to "cnt_index"
// to specify the counter index:
// (these are defined in pmu_reader.h)
// #define CNT_INST_EXE 0x00
// #define CNT_L1D_ACSS 0x01
// #define CNT_L1D_MISS 0x02
// #define CNT_LLC_ACSS 0x03
// #define CNT_LLC_MISS 0x04
// #define CNT_TLB_MISS 0x05
unsigned int get_single_event(unsigned int cnt_index) {
    unsigned int value = 0;
    // Implement your code here
    return value;
}
```

- Read the event counter of "`cnt_index`" by reading "PMXEVCNTR". Before accessing "PMXEVCNTR", you have to use "PMSELR" to specify the event counter, `cnt_index`, required to read.

Submit your modified "pmu_reader.c" source code that measures the six performance counter registers for the given workloads.

8. Workload analysis based on performance counter measurement

Now write a report analyzing **three workloads** under **different conditions**, using performance counters. Analyze the performance counters over the following 9 workload combinations of "`workload*_body()`":

- Single-core: Workload 1 on Core 0
- Single-core: Workload 2 on Core 0
- Single-core: Workload 3 on Core 0
- Two-core: Workload 1 on Core 0 + Workload 2 on Core 1
- Two-core: Workload 1 on Core 0 + Workload 3 on Core 1
- Two-core: Workload 2 on Core 0 + Workload 3 on Core 1
- Two-core: Workload 1 on Core 0 + Workload 1 on Core 1
- Two-core: Workload 2 on Core 0 + Workload 2 on Core 1
- Two-core: Workload 3 on Core 0 + Workload 3 on Core 1

You have to measure the 9 combinations over two frequency levels.

Plot the following results (at least) in your report:

- Execution time, L1 cache miss ratio, and LLC miss ratio for each of the three single-core workloads over min/max frequency
- Energy consumption for three single-core workload configurations over min/max frequency
- Changes in execution time, energy, and L1/LLC cache miss ratios when two workloads run on the different cores of the maximum frequency (six cases) at the same time

Then, carefully explain your discussion for the analysis results in the report. It should include answers to the following questions:

- What are the differences among the three workload characteristics?
- How does the frequency affect the execution time and energy?
- Is there an observable change in PMU characteristics if running two workloads together? Why or why not?
- What is the relationship between the miss ratios and execution time? Why or why not?
- What is the relationship between the execution time and the cycle count?

Use the table below to calculate the energy values:

CPU Frequency	Power
1.2 GHz	580 mW
600 MHz	500 mW

9. Individual Project Submission (by 23:59:59 PST)

Submit the following via TED:

- Source code of your modified pmuon.c
    - It must select the six event event counters in the order:
      Instructions, L1 cache access, L1 cache miss, L2 cache access, L2 cache miss, instruction TLB miss
- Source code of your modified pmu_reader.c
    - It must include the code to read the configurable performance counters with the given counter indexes
- Report
    - Maximum 2pgs, 12pt Times New Roman font, excluding figures and table. The figures needed are specified above.
    - Carefully explain what you learned by analyzing the workload combinations over frequency settings. Refer to the prompts in the report above as a guide.
    - Do not include your source code in the report.