CSE306 Spring 2023: Assignment 2

Workload characterization & System Program Implementation
Detailed Instructions

In this assignment, you will implement a simple file archiver program and analyze it to perform optimization by measuring its characteristics.

This assignment consists of two parts:

1) Measurement tool setup (with some quiz)
2) C program implementation – the file archiver

Due:

Upload your submission by June 16 (Friday) 23:59:59

What to submit

Submit a zip file, which includes the following contents in three directories, in LMS.

1. part1/pmuon.c (with the correct answer for Part 1 Quiz)
2. part2/
   A. Include all source code
   B. Include Makefile
      i. Your compile should be compiled by typing "make" without any option
3. report/report.pdf: Write your report (up to two pages) that describes:
   A. Brief explanation about your implementation strategies for Part 2.
   B. PMU measurement results including
      Case 1: When packing 100 files, where each file is 1MB.
      Case 2: When unpacking the archive file having 100 files
      i. Provide a simple discussion that compares each case with the *matmul* program provided.
         1. Recommend considering # of executed instructions, cycles, cache miss ratio, etc.
      ii. You can also discuss whatever you want by testing more cases.
         1. e.g., you may also want to compare the PMU characteristics among packing, unpacking, adding, and deleting, discuss how it performs when packing different numbers of files, etc.

The zip file should be formatted as **HW2_YOUR_STUDENT_NUMBER.zip.**

- For example, **HW2_202301234.zip**

Part 1. Performance measurement setup + some quiz

The purpose of this document is to help you to prepare your system to measure some system characteristics of a program. We are going to use performance counters. What you need to do is:

- Compiling and installing the custom kernel source code (You've already done in the assignment 1.)
- 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. 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.

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?

1. 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, don't worry, you just need to (i) solve the quiz that we're giving, (ii) compile it using the provided Makefile, and (iii) test it on your machine with this guideline.
If you need to learn more details about the kernel module, we 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.

1.1. Compile *hello world* kernel module for your machine (optional)

If you want to learn about how to build a kernel module on **your build machine** – **not RPi** (e.g., your virtual machine), you can follow the guideline in this section.

First, let's 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 */
#include <linux/module.h>

MODULE_LICENSE(“GPL”);


int init_module(void)
{
    printk(KERN_INFO "CSE306: Hello world.\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "CSE306A: 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 (5.15 in the example below) is the kernel version of your build machine. This will vary depending on your machine configuration.

```
$ modinfo hello.ko
filename:      hello.ko
srcversion:     B32527F71C81E36B88FEED8
depends:
vermagic:       5.15-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 in its kernel log history. (Note that you can't use printf and stdout in the kernel module, since it is for user-level applications.). You can see "CSE237A: Hello world." and "CSE237A: goodbye world." messages in the kernel log history using '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.
```

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

gcc is the compiler on the host machine specific to its architecture, e.g., an Intel-based machine. But, RPi is an ARM-based machine. So, 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/rpi-linux   # The custom Rpi kernel source directory in HW1
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 CROSS_COMPILE=aarch64-linux-gnu- ARCH=arm64
```

If an error occurs during the make process, configure the kernel as you did in HW1, build it for your environment, and try again.

This will generate hello.ko kernel module.

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] CSE306: Hello world.
[  273.468906] CSE306: 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.

2. Performance Monitoring Unit (PMU)

Modern processors include logic to gather various statistics on the operation of the processor and memory system during runtime, based on the PMU architecture. For example, you can measure the number of instructions, cycles, cache behaviors, etc., using the PMU. These events provide useful information about the behavior of the processor that you can use when debugging or profiling code. For AArch64 use in RPi4, 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.

You can find the detailed information on the ARM architecture manual (we attached the link below); we provide a kernel module that measures the performance counters for selected events.
ARM architecture manual: https://developer.arm.com/documentation/ddi0500/e/system-control/aarch64-register-summary/aarch64-performance-monitor-registers?lang=en

The provided source code, "pmuon.c", set some event numbers for PMU registers. You are not expected to understand every detail of the implementation, but your job is to determine the proper PMU event numbers of interests.

Let's the take a look at the following part in "pmuon.c"

```
// PMU event number to collect -- YOUR QUIZ!
#define EVT_INSTRUCTIONS    0x08 // # of instructions
#define EVT_CACHE_MISSES    0x03 // # of L1 data cache refills (misses)
#define EVT_CACHE_REFS      0x00 // Replace it with the event number for # of L1 data cache accesses
#define EVT_CPU_CYCLES      0x01 // Replace it with the event number for # of CPU cycles
```

The defined constants specify the PMU events to be set and read here:

```
// 5. Set event counter registers
    asm volatile("msr pmevtyper0_el0, %0" :: "r"(EVT_INSTRUCTIONS));
    asm volatile("msr pmevtyper1_el0, %0" :: "r"(EVT_CACHE_MISSES));
    asm volatile("msr pmevtyper2_el0, %0" :: "r"(EVT_CACHE_REFS));
    asm volatile("msr pmevtyper3_el0, %0" :: "r"(EVT_CPU_CYCLES));

                                                (……...)

// Read all counters
    asm volatile("mrs %0, pmevcntr0_el0" : "=r"(counters[0]));
    asm volatile("mrs %0, pmevcntr1_el0" : "=r"(counters[1]));
    asm volatile("mrs %0, pmevcntr2_el0" : "=r"(counters[2]));
    asm volatile("mrs %0, pmevcntr3_el0" : "=r"(counters[3]));
    printk("[Core %d]\tInsts: %llu, Misses: %llu, Refs: %llu, Cycles: %llu\n", (……)
```

The PMU values will be printed when the kernel module is unloaded.

[QUIZ!] The first two defined constants are correct event numbers, you should find and fill the two other events: EVT_CACHE_REFS and EVT_CPU_CYCLES. Please take a look at the following page of the architecture manual. https://arm-software.github.io/CMSIS_5/Core/html/group__pmu8__events__armv81.html

2.1. Read PMU events for a user-space program

Let's cross-compile the kernel module using the Makefile that we introduced in the last section. Then copy the compiled kernel module, "pmuon.ko" into the RPi. The kernel module starts the PMU measurement when loaded and finishes it when unloaded. So, you should execute your program in-between. For example, if you profile the simple matmul code that we provide together, you can test it like this:

```
$ sudo insmod pmuon.ko
$ ./matmul
$ sudo rmmod pmuon
$ dmesg
```

Don't forget to type **"sudo"** when loading/unloading pmuon.ko. Since the matmul can run any single core on your RPi, you may want to select a core. The "tastset" tool provides the functionality. So, using the script below, provided with "bench.sh", you can automate the benchmarking workflow – (i) load module, (ii) drop page caches (in kernel) (iii) execute the target program on a selected core, (iv) remove the module, and (v) show the results with dmesg.

```
#!/bin/bash
sudo insmod ./pmuon.ko
sudo sync && sudo sysctl -w vm.drop_caches=3
taskset -c 0 $@
sudo rmmod pmuon
dmesg | tail -15
```

Note that "$@" takes the command line arguments, so you can run like this:

```
$ ./bench.sh ./matmul
```

If you correctly selected the correct PMU event numbers in "pmuon.ko" and compiled "./matmul" without any optimization, (e.g., $ gcc -o matmul matmul.c) then your results will look like the red line below for Core 0:

```
[   51.697753] Turn PMU on Core 2
[   51.697753] Turn PMU on Core 1
[   51.697752] Turn PMU on Core 0
[   51.697753] Turn PMU on Core 3
[   51.697765] We have 6 configurable event counters on Core 0
[   51.697765] We have 6 configurable event counters on Core 3
[   51.697765] We have 6 configurable event counters on Core 1
[   51.697768] We have 6 configurable event counters on Core 2
[   51.697777] Ready to use PMU
[   51.924441] sysctl (1117): drop_caches: 3
[   55.181210] PMU Kernel Module Off
[   55.181233] [Core 1] Insts: 40425380, Misses: 376119, Refs: 15853218, Cycles: 54272490
[   55.181235] [Core 2] Insts: 28462741, Misses: 160320, Refs: 10821824, Cycles: 35368394
[   55.181235] [Core 3] Insts: 47854639, Misses: 388373, Refs: 18793216, Cycles: 70818505
[    55.181235] [Core 0] Insts: 7710987850, Misses: 2688447, Refs: 3432335927, Cycles:
4615766098
```

Part 2. Implement File Archiver

The main part of this homework is to implement a program that performs file archiving. You may not hear about "file archiver", but I bet you have many experiences that you already used it. The file compress/decompress utility like Zip is a file archiver but with very advanced functions including file compression.

Unfortunately, we will not provide any skeleton code – the good news is that everything is up to you! There're tons of ways to implement the file archiving mechanism. You need to think about what would be a good structure inside the file you should design. Let's call your archive program by "arcx".

Things to do:

The followings are the commands that the tool should support.

1. pack -- Creating a new archive file

$ ./arcx pack *archive-filename src-directory*

- It archives files stored in a source directory (src-directory) into a new single file named with "*archive-filename*".
    - Note: You would also need to write file metadata such as names and sizes to unarchive them later. Design your own strategy (i.e., a data structure stored in the output file) to save the metadata.
- If "archive-filename" already exists, overwrite it.
- If there is no file in the "src-directory", print an error message and exit the program without creating any file.
- For simplicity, you can assume that the source directory does not have any child directory. If exists, just ignore it.

2. unpack -- Creating a new archive file

$ ./arcx unpack *archive-filename dest-directory*

- It unpacks the archived files into the destination directory.
- If the destination directory does not exist, make the directory first.
- If a file already exists in the destination directory with the same name to unpack, overwrite it.
- If "*archive-filename*" does not exist, just print an error message.

3. add – add a single file into an archive file

$ ./arcx del *archive-filename target-filename*

- It adds a file with "*target-filename"* from the archive file.
- If the archive file already has a file having the same *target-file* (base) name, print an error message and do no change the archive file.
    - For example, if the archive file has "abc.txt" and you're trying to "./arcx add ANOTHER_DIRECTORY/abc.txt", then it is an error.

4. del – remove a single file from an archive file

$ ./arcx del *archive-filename target-filename*

- It deletes the file with "*target-filename"* from the archive file.

- If the archive file does not have the *target-file* name, print an error message and do no change the archive file.

5. list – show the file list in the archive file

```
$ ./arcx list archive-filename
```

- It shows the list of files in the archive.
- It should show the total number of files, the pairs of each file name and size.

Sample: (There's no guideline for the message format. Just try to make it informative.)

```
$ ls -l srcdir
-rwxr-xr-x  1 cellpi4 cellpi4   1 May 27 00:21 text1.txt
-rwxr-xr-x  1 cellpi4 cellpi4   1 May 27 00:21 text2.txt
-rwxr-xr-x  1 cellpi4 cellpi4   8.6K May 27 00:21 img1.jpg
-rwxr-xr-x  1 cellpi4 cellpi4   8.6K May 27 00:21 img2.jpg
-rwxr-xr-x  1 cellpi4 cellpi4   8.6K May 27 00:21 img3.jpg
$ ./arcx pack working.arcx srcdir
5 file(s) archived.
$ ./arcx del working.arcx text1.txt
1 file deleted.
$ ./arcx del working.arcx text1.txt
No such file exists.
$ ./arcx del working.arcx img1.jpg
1 file deleted.
$ ./arcx list working.arcx
3 file(s) exist.
text2.txt 1 byte
img2.jpg 8600 bytes
img3.jpg 8600 bytes
$ ./arcx add working.arcx srcdir/text1.txt
1 file added.
$ ./arcx add working.arcx srcdir/text1.txt
Same file name already exists
$ ./arcx list working.arcx
4 file(s) exist.
text2.txt 1 byte
img2.jpg 8600 bytes
img3.jpg 8600 bytes
text1.txt 1 byte
$ ./arcx upack newfile destdir
$ ls -l destdir
-rwxr-xr-x  1 cellpi4 cellpi4   1 May 27 00:21 text1.txt
-rwxr-xr-x  1 cellpi4 cellpi4   1 May 27 00:21 text2.txt
-rwxr-xr-x  1 cellpi4 cellpi4   8.6K May 27 00:21 img2.jpg
-rwxr-xr-x  1 cellpi4 cellpi4   8.6K May 27 00:21 img3.jpg
```

Grading rules:

1. Functionality
   A. Obviously, it should run without any errors.
2. Speed & Archived file size
   A. Note that since we allow “add” and “delete” for the archive file that already exists, it can make many fragmentations if you don’t carefully handle.
   B. For example, let’s assume that it first archives 100 files and delete 50 files in the random position by just making some spaces used. Then, it may add 50 new files at the end of the archive file and delete 50 random files. If the mechanism is like this deadly simple, the archive file will have a lot of used spaces.
   C. One workaround is to make a new archive whenever performing “add” and “del”. However, this approach will be slow.
   D. So, one of your goal is to utilize the space effectively with good execution time for various cases.