Homework 2

W4118 Fall 2014

UPDATED: Friday 9/26/2014 at 5:50pm EST
DUE: Monday 9/29/2014 at 11:59pm EST

All non-programming, written problems in this assignment are to be done by yourself. Group collaboration is permitted only on the kernel programming problems. All homework submissions (both individual and group) are to be made via **Git**.

You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Be aware that commits pushed after the deadline will not be considered. Refer to the homework policy section on the **class web site** for further details.

Individual Written Problems:

The Git repository you will use for the individual, written portion of this assignment can cloned using:

```
git clone https://os1.cs.columbia.edu/UNI/hmwk2-writ.git
```

(replace `UNI` with your own UNI). This repository will be accessible only by you, and you will use the same SSH public/private key-pair used for homework 1.

Exercise numbers refer to the course textbook, *Operating Systems Principles and Practice*. Each problem is worth 5 points.

1. Exercise 2.1 (Chapter 2, Exercise 1)
2. Exercise 2.2
3. Exercise 2.3
4. Exercise 2.9
5. Exercise 2.12
6. Exercise 2.15

Group Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository your entire group will use to submit the group programming problems can be cloned using:

```
git clone https://os1.cs.columbia.edu/TEAM/hmwk2-prog.git
```

(Replace `TEAM` with the name of your team, e.g. `team1`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the checkpatch.pl script on the default path in the provided VM. Errors from the script in your submission will cause a deduction of points.

The kernel programming for this assignment will be done using a custom **Android** device emulator. As a part of this assignment, you will be experimenting with the Android platform and gaining familiarity with the development environment. Subsequent assignments will involve using the Android SDK and devtools to modify a Google **Nexus 7**. The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Android emulator is built on a system called **QEMU** which emulates an ARMv5 processor (specifically the ARM926E).

Because the target CPU is (most likely) different from the CPU running in your personal computer, you will have to *cross-compile* any software, including the linux kernel, to run on the different platform. The relevant Android/ARM SDK files have been pre-installed in a new virtual machine available for download **here**. The username and password information is the same as the previous VM you used for homework 1.

In order to work with the emulator, you will need an X11 display. This can be accomplished in several ways, one of which is SSH with X-forwarding:

a. If you have a Linux host, skip to step c. If you have a Mac host, skip to step b. If you have a Windows host, install an SSH client, like **PuTTY**.

b. Install an X server, like **XQuartz** (Mac) or **Xming** (Windows).

c. In your VM run the following to get your VM's IP address:

```
getMyIP.sh
```

d. Access your VM via SSH.

- In terminal (on Linux or Mac), the command is:

```
ssh -X w4118@[VM IP Address]
```

- In PuTTY (on Windows), SSH connections are made via a GUI. Make sure your X server is running and to enable X-forwarding in the PuTTY settings under Connection/SSH/X11.

---

1. **(5 pts.) Create an Android Virtual Device (AVD), and install a custom 3.4 kernel in it.**
   The appropriate *tools* directory has already been added to your path, so you should be able to run the Android SDK manager (assuming you have setup X11 display forwarding properly) using the `android` command.

   Once your AVD is setup, you can start it from the GUI to make sure it boots.
   NOTE: initial device rotation seems somewhat quirky in this version of the emulator. Use the Ctrl-F11 key sequence to "rotate" the emulator a couple of time to get the GUI state to sync.

   Once the emulator is running, it's time to build a custom kernel. The kernel source is located in your team's git repository in the *kernel* directory, and an appropriate **ARM cross-compiler** has been installed in the VM, and resides in */home/w4118/utils/arm-2013.11/bin*. A default kernel configuration has been provided in the *arch/arm/configs* directory, so building the kernel should be as easy as:

   ```
   make ARCH=arm goldfish_armv7_defconfig
   make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
   ```

   (the trailing dash is necessary).

What's going on here?
You are *cross-compiling* the linux kernel to run on a machine whose CPU architecture is different from the machine on which you are compiling. The `ARCH=arm` variable passed to make tells the build system to use ARM specific platform/CPU code. The `CROSS_COMPILE` make variable tells the build system to use compiler / linker tools whose names are prefixed by the value of the variable (check - you should have a program named *arm-none-linux-gnueabi-gcc* in your path). You may want to shorten the amount of typing you do to start a compile / config by adding an alias to your *~/.bashrc* file:

```
alias armmake="make -j2 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-"
```

This allows you to simply type *armmake* in the root of the kernel directory instead of constantly providing all the make variables.

The final kernel image (after compilation) is output to *arch/arm/boot/zImage*. This file is the one you will be using to boot the emulator.

Boot the Android emulator with your custom kernel using the `emulator` command:

```
emulator -avd NameOfYourAVD -kernel arch/arm/boot/zImage -show-kernel
```

You might want to make a shell alias for that command as well.

2. **(45 pts.) Write a new system call in Linux**
The system call you write should take two arguments and return the process tree information in a depth-first-search (DFS) order. Note that you will be modifying the Android kernel source which you previously built, and cross-compiling it to run on the Android emulator using the techniques described in problem 1.

   The prototype for your system call will be:

   ```
   int ptree(struct prinfo *buf, int *nr);
   ```

   You should define *struct prinfo* as:

   ```
   struct prinfo {
           pid_t parent_pid;               /* process id of parent */
           pid_t pid;                      /* process id */
           pid_t first_child_pid;          /* pid of youngest child */
           pid_t next_sibling_pid;         /* pid of older sibling */
           long state;                     /* current state of process */
           long uid;                       /* user id of process owner */
           char comm[64];                  /* name of program executed */
   };
   ```

   in `include/linux/prinfo.h` as part of your solution.

   The argument `buf` points to a buffer for the process data, and `nr` points to the size of this buffer (number of entries). The system call copies at most that many entries of the process tree data to the buffer and stores the number of entries actually copied in `nr`.

   If a value to be set in `prinfo` is accessible through a pointer which is null, set the value in `prinfo`

to 0. For example, the `first_child_pid` should be set to 0 if the process does not have a child.

Your system call should return the total number of entries on success (this may be bigger than the actual number of entries copied). Your code should handle errors that can occur but not handle any errors that cannot occur. At a minimum, your system call should detect and report the following error conditions:

- `-EINVAL`: if `buf` or `nr` are null, or if the number of entries is less than 1
- `-EFAULT`: if `buf` or `nr` are outside the accessible address space.

The referenced error codes are defined in **include/asm-generic/errno-base.h**.

Each system call must be assigned a number. Your system call should be assigned number **223**.

NOTE: Linux maintains a list of all processes in a doubly linked list. Each entry in this list is a `task_struct` structure, which is defined in **include/linux/sched.h**. When traversing the process tree data structures, it is necessary to prevent the data structures from changing in order to ensure consistency. For this purpose the kernel relies on a special lock, the `tasklist_lock`. You should grab this lock before you begin the traversal, and only release the lock when the traversal is completed. While holding the lock, your code may not perform any operations that may result in a sleep, such as memory allocation, copying of data into and out from the kernel etc. Use the following code to grab and then release the lock:

```
read_lock(&tasklist_lock);
...
...
read_unlock(&tasklist_lock);
```

HINT: In order to learn about system calls, you may find it helpful to search the linux kernel for other system calls and see how they are defined. You can use the **Linux Cross-Reference** (LXR) to investigate different system calls already defined. The files **kernel/sched/core.c** and **kernel/timer.c** should provide good reference points for defining your system call.

HINT: You should not try to create your own linked list method for the data structures inside the kernel, but use the existing infrastructure. See `include/linux/list.h` and look for other places in the kernel where lists are used for examples on how to use them (there are many such places). Also, the course materials contain information about linked lists in the kernel.

HINT: When getting the `uid` of a process, you can simply follow the `cred` field.

3. **(10 pts.) Test your new system call**
   Write a simple C program which calls `ptree`. Your program should print the entire process tree (in DFS order) using tabs to indent children with respect to their parents. For each process, it should use the following format for program output:

```
printf(/* correct number of \t */);
printf("%s,%d,%ld,%d,%d,%d,%d\n", p.comm, p.pid, p.state,
        p.parent_pid, p.first_child_pid, p.next_sibling_pid, p.uid);
```

Example program output:

```
...
init,1,1,0,60,0,0
        ...
        servicemanager,44,1,1,0,43,1000
        vold,45,1,1,0,44,0
        netd,47,1,1,0,45,0
        debuggerd,48,1,1,0,47,0
        rild,49,1,1,0,48,1001
        surfaceflinger,50,1,1,0,49,1000
        zygote,51,1,1,745,50,0
                system_server,371,1,51,0,0,1000
                ...
                android.launcher,530,1,51,0,517,10008
                ...
...
kthreadd,2,1,0,42,1,0
        ...
        ksoftirqd/0,3,1,2,0,0,0
        kworker/0:0,4,1,2,0,3,0
        ...
        khelper,6,1,2,0,5,0
        ...
```

The `ps` command in the emulator will help in verifying the accuracy of information printed by your program.

NOTE: Although system calls are generally accessed through a library (libc), your test program should access your system call directly. This is accomplished by utilizing the generall purpose `syscall(2)` system call (consult its man page for more details).

Compiling for the Android Emulator:
You will have to cross-compile your test program to run under the emulator. Fortunately this is straight-forward so long as you use static linking. Compile your code with the ARM toolchain installed on your emulator. For single-file projects this is:

```
arm-none-linux-gnueabi-gcc -static -o prog_name src_file.c
```

.
For more complex project (or even the single file) you can use the following minimal Makefile:

```
# Set this to the name of your program
TARGET = output_name

# Edit this variable to point to all
# of your sources files (make sure
# to put a complete relative path)
SOURCES = mysrc1.c \
          mysrc2.c

# ------------------------------------------------
#
# Don't touch things below here unless
# you know what you're doing :-)
#
OBJECTS = $(SOURCES:%.c=%.c.o)
INCLUDE = -I.
CFLAGS = -g -O2 -Wall $(INCLUDE) -static
LDFLAGS = -static
CC = arm-none-linux-gnueabi-gcc
LD = arm-none-linux-gnueabi-gcc
```

```
default: $(SOURCES) $(TARGET)

$(TARGET): $(OBJECTS)
        @echo [Arm-LD] $@
        @$(LD) $(LDFLAGS) $(OBJECTS) -o $@

%.c.o: %.c
        @echo [Arm-CC] $<...
        @$(CC) -c $(CFLAGS) $< -o $@

clean: .PHONY
        @echo [CLEAN]
        @rm -f $(OBJECTS) $(TARGET)

.PHONY:
```

A good (although minimal) reference on compiling / debugging command line programs for Android can be found **here**.

Installing / Running on the Android Emulator:
In order to run the program you just compiled, you will have to move it to the Android emulator's filesystem. This is done using the multi-purpose Android Debug Bridge (**adb**) utility. After starting up your Android virtual device with the custom kernel, you can check your connection by issuing:

```
adb devices
```

If you see the emulator listed there, then you can proceed to install your custom executable, and run it.

To move a file to the emulator:

```
adb push /path/to/local/file /data/misc
```

The /data/misc directory is read/write on the emulator, and you should be able to execute your program from there. To execute, you can either enter a shell on the emulator, or call the program directly from adb:

```
adb shell
# /data/misc/exe_name
```

or

```
adb shell /data/misc/exe_name
```

To pull a file out of the emulator:

```
adb pull /path/in/emulator /local/path
```

To debug on the emulator:

```
arm-none-linux-gnueabi-gdb exe_name
(gdb) target remote localhost:1234>
```

4. **(10 pts.) Investigate the Android process tree**

    a. Run your test program several times. Which fields in the `prinfo` structure change? Which ones do not? Discuss why different fields might change with different frequency.

    b. Start the mobile web browser in the emulator, and re-run your test program. How many processes are started? What is/are the parent process(es) of the new process(es)? Close the browser (press the "Home" button). How many processes were destroyed? Discuss your findings.

    c. Notice that on the Android platform there is a process named *zygote*. Investigate this process and any children processes:

        i. What is the purpose of this process?

        ii. Where is the *zygote* binary? If you can't find it, how might you explain its presence in your list of processes?

        iii. Discuss some reasons why an embedded system might choose to use a process like the *zygote*.
        HINT: A key feature of embedded systems is their resource limitations.