Homework 3

W4118 Fall 2014

UPDATED: Wednesday 10/1/2014 at 5:45pm EST
DUE: Tuesday 10/14/2014 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 be cloned using:

```
git clone https://os1.cs.columbia.edu/UNI/hmwk3-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 4.2 (Chapter 4, Exercise 2)
2. Exercise 5.7
3. Exercise 5.9
4. Exercise 6.1
5. Exercise 6.3
6. Exercise 6.7

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/hmwk3-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`. You can see the documentation for these by writing `$ man git-pull` etc. You may need to install the `git-doc` package first (e.g. `$ apt-get install git-`

doc.

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 scripts/checkpatch.pl script included in the Linux kernel. Errors from the script in your submission will cause a deduction of points.

All kernel programming assignments in this year's class are done on the Android operating system and targeting the ARM architecture. For more information on how to use adb and aliasing refer to Homework 2. Use "adb -e" for emulator commands and "adb -d" to direct the commands to the device. If you run in to a read-only disk error then type "adb remount" add -e or -d accordingly.

You can use Homework 2's VM for this assignment, which can be downloaded from **here**.

The kernel programming for this assignment will be done using a Google **Nexus 7 (2013)** The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Google Nexus 7 uses an ARMv7 CPU. The Android emulator can also be used and is built on a system called QEMU which emulates an ARMv7 processor. However, to use the emulator, you will have to use a new emulator that we have provided to simulate accelerator devices that you will use for this assignment. In your team's git repository in the *utils* directory, there is a custom emulator that you should install in your VM to run with your kernel:

```
sudo apt-get install libsdl1.2debian:i386
```

```
cp emulator-hmwk3 ~/utils/android-sdk-linux/tools/ # copy binary once
```

```
emulator-hmwk3 -avd NameOfYourAVD -kernel ../flo-kernel/arch/arm/boot/zImage -show-kernel
```

1. **(5 pts.) Install a custom kernel on your Nexus 7**

   Make sure you still have the Android SDK unpacked in your home directory on the VM as specified by Homework 2.

   The kernel source is located in your team's git repository in the *flo-kernel* directory, and an appropriate cross-compiler has been installed in the VM, and resides in the */home/w4118/utils* directory. In this homework, there are two configurations that can be used to complete the assignment - one configuration for the emulator, and one configuration for the device (these two devices have different [virtual] hardware and thus require different drivers enabled/disabled). To switch between the two is straight-forward using the kernel's default configuration mechanism. To compile for the Nexus 7 device, use the custom flo_android default configuration:

   ```
   make ARCH=arm flo_defconfig
   ```

   To compile for the emulator, use the *goldfish* default configuration:

   ```
   make ARCH=arm goldfish_armv7_defconfig
   ```

   These commands configure the Makefiles in the kernel to include all the proper drivers / support for the chosen platform. After you have configured the kernel (which is only necessary when switching between emulator and device), you can compile the kernel for Nexus 7 using:

   ```
   make ARCH=arm CROSS_COMPILE=arm-eabi-
   ```

To compile the kernel for the goldfish emulator, you can use:

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

Modifications to the configuration can be done using:

```
make ARCH=arm menuconfig
```

however, these changes do not get saved into the *default* configuration, so if you modify the kernel config and then use one of the default configuration commands, your custom changes will be erased!

The final kernel image (after compilation) is output to: *arch/arm/boot/zImage*. This file is the one you will be using to boot either the Nexus 7 or the emulator.

Booting the kernel on the Nexus 7 requires the `fastboot` utility, which has been provided in your class VM. This utility can boot a custom kernel on the device, and can also flash the custom kernel into the device's boot partition. Here are the steps you must follow in order to set up your environment and successfully boot a new kernel on your Nexus 7:

1. Copy the file from your team's git repository in *utils/51-android.rules* to the path */etc/udev/rules.d/* of your VM, change its permission to be both readable and accessible, then type the following command to acquire the permission to access your device.

   ```
   sudo service udev restart
   ```

2. Put the device into "fastboot" mode by holding down the *lower* key while powering-up the device. Alternatively, you can type "adb reboot bootloader" in your VM if you have enabled USB Debugging. You will know that the device has successfully been placed in "fastboot" mode when the screen shows a green android.

   NOTE: You have to enable USB debugging on your device before you connect it to the VM. Go to "Settings->About tablet" on your Nexus 7 device. Tap on the icon "Build number" at least seven times to enable "Developers Options". When it's enabled, go back to "Settings->Developer Options". Make sure you check the "USB debugging" option. Next, connect your Nexus 7 USB device to the VM (go to settings on your VMware product and connect to your USB device). Then type "adb devices" to see if the device is connected.

3. Once the device is in "fastboot" mode, check the connection to the VM with:

   ```
   fastboot devices
   ```

   If you see a line which looks like:

   ```
   HT851N002807 fastboot
   ```

   then you are ready for the subsequent steps.

4. Unlock your device and flash the Android 4.4.4 factory image on it. **Unlocking is a one-time process.** Type the following command in fastboot mode to unlock your bootloader.

   ```
   fastboot oem unlock
   ```

Then download an Android 4.4.4 factory image and unpack the file:

```
wget https://dl.google.com/dl/android/aosp/razor-ktu84p-factory-b1b2c0da.tgz
```

```
tar -zxvf razor-ktu84p-factory-b1b2c0da.tgz
```

Go to the newly created directory, and execute the script *flash-all.sh*. After a while, you will see your device boots up with Android Kitkat 4.4.4. Go to "Settings->About tablet" on your Nexus 7 to verify the version number.

5. To boot your custom kernel (and leave the existing kernel in-place for subsequent reboots of the device), set the device to **fastboot** mode and then go to the directory *utils* in your team's git repository to run the *update-kernel-ram.sh* script:

```
./update-kernel-ram.sh ../flo-kernel/arch/arm/boot/zImage
```

The device should eventually boot up with your new kernel.

6. To write the new kernel image into the boot partition of the device's flash memory, use the following fastboot command:

```
./update-kernel-flash.sh ../flo-kernel/arch/arm/boot/zImage
```

This will allow your system to boot the new kernel after a power-cycle.
BE CAREFUL THOUGH - YOU WILL WANT TO ENSURE SYSTEM STABILITY BEFORE COMMITTING YOUR WORK TO FLASH!

2. **(10 pts.) Write a user-space daemon to pass device acceleration into the kernel**

On the Android platform, the accelerometer provides a three dimensional vector indicating the direction and magnitude of acceleration. When the device is sitting on a table, the accelerometer reads a magnitude of g = 9.81 m/s^2. Similarly, when the device is in free fall and therefore rapidly accelerating toward the ground at 9.81 m/s^2, the accelerometer reads a magnitude of g = 0 m/s^2. However, floating point operation is not well supported in kernel, so the acceleration values can instead be scaled by a factor of 100 with any remaining fraction ignored. For example, a value of -9.54 can be converted to -954.

You are to write a daemon process, called `accelerate_d`, which polls the accelerometer and writes it to the kernel. You will write a system call defined below that writes the accelerometer data to the kernel. You do not need to keep track of any history, so subsequent calls to the system call will overwrite the accelerometer data in the kernel. You should not make your CPU busy reading the device, so you are required to make the daemon intermittent. For example, you can add a timer or make the daemon pause using `usleep`. Define the time interval in your daemon program file.

Use the system call interface below:

```
/*
 *Define time interval (ms)
 */

#define TIME_INTERVAL  200

/*
 * Set current device acceleration in the kernel.
 * The parameter acceleration is the pointer to the address
 * where the sensor data is stored in user space.  Follow system call
 * convention to return 0 on success and the appropriate error value
```

```
 * on failure.
 * syscall number 378
 */

int set_acceleration(struct dev_acceleration __user * acceleration);

/*
 * The guide on how to get an instance of the default acceleration
 * sensor is available here.
 */

/*
 * The data structure to be used for passing accelerometer data to the
 * kernel and storing the data in the kernel.
 */

struct dev_acceleration {
        int x; /* acceleration along X-axis */
        int y; /* acceleration along Y-axis */
        int z; /* acceleration along Z-axis */
};
```

Only the administrator (`root` user) should be able to update the device accelerometer. All related functions should be placed in `kernel/acceleration.c`, and `include/linux/acceleration.h`

A basic command-line daemon project has been provided in your homework repository under the `acceleration_d` directory, which consists of:

- `accelerationd.c`: the main function of the deamon process resides and the implementation of polling sensor data.
- `acceleration.h`: define dev_acceleration structure
- `inc/`: sensor data output and polling interface is defined in hardware/sensor.h
- `akm8975.h`: device driver headfile

While this project provides a basic template for you as a starting point, you will probably need to modify `accelerationd.c` and `acceleration.h`. Please see the comments in `accelerationd.c` to see where you may want to make modifications. To add files to the project, edit the Makefile and add lines to the `SOURCES` variable. Please do not remove any of the existing source files.

To compile the program simply type `make`. To install the program on the device (or emulator) type `make install_dev` (or `make install_emu`). To run on the device (or emulator) type `make run_dev` (or `make run_emu`).

NOTE:
This must be a true daemon i.e. it should place itself in the background using a call to `fork()` as opposed to forcing the program into the background using shell techniques such as the *&* symbol or the `Ctrl-Z, bg` command sequence.

3. **(45 pts.) Acceleration based Synchronization Primitive**

Design and implement a new kernel synchronization primitive based on acceleration. Shaking is a well known gesture based on the accelerometer. It can be detected via an acceleration difference within some time interval along a coordinate axis. For example, whether the phone is being shaken (not stirred!) in a horizontal or vertical fashion could be read from acceleration changes along the X-axis and Y-axis. A shake may also cause acceleration changes along multiple axes.

To detect shakes properly, it is important to avoid tiny accidental movements that should not be considered shakes, and avoid having accelerations in different directions canceling each other out. In this homework, monitoring a device shake requires the following steps. First, you will need to buffer sensor data in the kernel to calculate the difference between every two successive accelerations for some number of accelerations. The difference between successive accelerations is called a movement or sample. Second, you will need to analyze a WINDOW of samples to determine how many of them have a strength that exceed a NOISE value. The strength of a movement is computed by summing the acceleration changes aggregated along all three x, y, and z dimensions. A certain number of movements within a WINDOW, each of which has strength greater than NOISE, are considered a Motion. Since acceleration data is generated at each TIME_INTERVAL as described in Problem 2 of this assignment, WINDOW is defined in terms of the number of TIME_INTERVALS. Note that although TIME_INTERVAL is defined as 200 ms in Problem 2, it is possible that you may find it useful to change the TIME_INTERVAL to detect shakes better based on your actual experience with your device.

Your kernel synchronization primitive should respond to shake motion events that exceed some baseline. A shake motion event is defined based on having a number of movements, acceleration differences, that are not considered noise and occur with enough frequency within some window of time. Your primitive will allow processes to wait on any shake motion events occurring that exceed some baseline, then wake up when such events occur. For example, a process might be waiting for a shake motion event where the movement along the x-axis is greater than 5 and the number of qualified movements exceeds 3.

The API for this synchronization mechanism is the following set of new system calls which you will implement:

```
/*Define the noise*/
#define NOISE 10

/*Define the window*/
#define WINDOW 20

/*
* Define the motion.
* The motion give the baseline for an EVENT.
*/
struct acc_motion {

     unsigned int dlt_x; /* +/- around X-axis */
     unsigned int dlt_y; /* +/- around Y-axis */
     unsigned int dlt_z; /* +/- around Z-axis */

     unsigned int frq;   /* Number of samples that satisfies:
                            sum_each_sample(dlt_x + dlt_y + dlt_z) > NOISE */
};


/* Create an event based on motion.
 * If frq exceeds WINDOW, cap frq at WINDOW.
 * Return an event_id on success and the appropriate error on failure.
 * system call number 379
 */

 int accevt_create(struct acc_motion __user *acceleration);

/* Block a process on an event.
 * It takes the event_id as parameter. The event_id requires verification.
```

```
 * Return 0 on success and the appropriate error on failure.
 * system call number 380
 */

 int accevt_wait(int event_id);


/* The acc_signal system call
 * takes sensor data from user, stores the data in the kernel,
 * generates a motion calculation, and notify all open events whose
 * baseline is surpassed.  All processes waiting on a given event
 * are unblocked.
 * Return 0 success and the appropriate error on failure.
 * system call number 381
 */

 int accevt_signal(struct dev_acceleration __user * acceleration);

/* Destroy an acceleration event using the event_id,
 * Return 0 on success and the appropriate error on failure.
 * system call number 382
 */

 int accevt_destroy(int event_id);
```

You should begin by thinking carefully about the data structures that you will need to solve this problem. Your system will need to support having multiple processes blocking on different shake motions at the same time, so you will probably need a set of range descriptor data structures, each of which identifies an event. Those data structures will need to be put in a list from which your code can find the appropriate descriptor corresponding to the accelerating changing range that you need. Space for the descriptors should be dynamically allocated, most likely using the kernel functions kmalloc() and kfree().

You should not make any assumptions about whether the system is a uniprocessor or multiprocessor system. Be sure to properly synchronize access to your data structures. Moreover, be sure to select the appropriate synchronization primitives such that they are both correct and efficient, in this order. For instance, you should prefer a spinlock over a semaphore *if-and-only-if* you don't plan to sleep while holding it. You should understand the producer-consumer problem and what you need to do to synchronize access to a buffer in the kernel. You may find it helpful to use the kernal structure *kfifo*.

You can choose to work at the level of wait queues using either the associated low-level routines such as `add_wait_queue()`, `remove_wait_queue()`, or the higher-level routines such as `prepare_to_wait()`, `finish_wait()`. You can find code examples both in the book (pages 58 - 61 of *Linux Kernel Development*) and in the kernel. If you prefer to use functions such as `interruptible_sleep_on()` and `sleep_on()`, then plan carefully because they can be racy in certain circumstances.

HINT: a useful method to guarantee the validity of a data structure in the presence of concurrent create, access and delete operations, is to maintain a reference count for the object. Then, the object should only be freed by the last user. The file system, for example, keeps a reference count for inodes: when a process opens a file the reference count of the inode is incremented. Thus if another process deletes the file, the inode remains valid in the system (albeit invisible) because its count is positive. The count is decremented when the process closes the corresponding file descriptor, and if it reaches zero then the inode is freed. A reference count must be protected against concurrent access, either using explicit synchronization or using atomic types (see `atomic_t` in Chapter 10 of the *Linux Kernel Development* book).

You should properly handle any errors that may occur and report meaningful error codes e.g. -ENOMEM in the event a memory allocation fails. As always, you are encouraged to look for existing kernel code that performs similar tasks and follow the coventions and practices it provides.

4. **(10 pts.) Write a C program to test the system**

Shake or not: Different applications have different sensitivity to the shaken motion. For example, some applications are sensitive to verticle shaking, some are sensitive to horizon shaking, and others are sensitive to both of them. Define these three categories. Forks N children and assign sensitivity to them. Each of them are reponsible to detect a shaking motion with its sensitivity.

Print "%d detected a vertical shake"

Print "%d detected a horizonal shake"

Print "%d detected a shake"

%d should be replace by the process identifier of the respective child. When an event is notified, all processes waiting on the event should be unblocked. If there are no processes waiting on the event, then nothing happens.

The original test process should wait for some period of time (>= 60 seconds), then close all child processes by closing the opened event (*not* by sending a signal or other such method). Make obvious the shaking needed to trigger the respective event!

Testing on the Emulator:

- Install the `sensorsim` binary, found in the `acceleration_d` directory in your team's git repository, into the emulator using:

```
adb -e remount
adb -e push sensorsim /system/bin
```

- After starting the emulator, you can simulate hardware sensor events using the `sensorsim` binary. The program will prompt you for a *Sensor ID*, enter 0.