CIS 520 - Fall 2013 - Project #3 – Android OS

DUE: Upload via K-State OnLine no later than 11:59 pm on Wednesday, Dec. 4, 2013

TO DO: Upload a zipped file called Proj3.zip that includes:

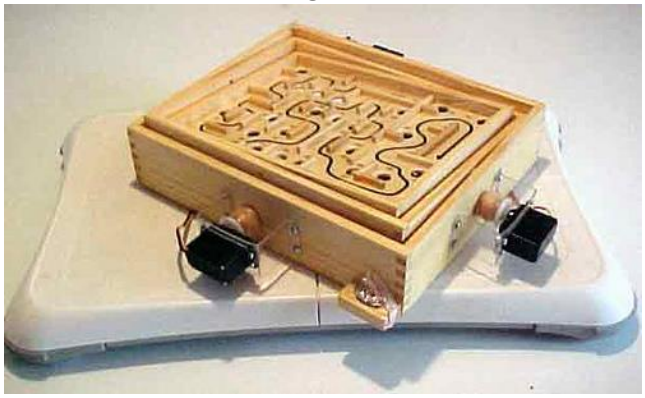
- A design document describing your applications and how to use them, in the top-level directory. This document should also answer the questions presented below.
- An Eclipse project folder for your applications.
- Updated kernel source in Kernel.tgz (remember to make clean before creating the archive).

The Android Operating System is available in a variety of flavors. The base version installed on our Samsung Galaxy Tabs is Honeycomb (Android Version 3.1). It can be upgraded to 3.2 or Ice Cream Sandwich (Android Version 4.0) via Kies, or a later version such as Jelly Bean (Android Version 4.1) via a custom rom installation. The official Samsung versions of the source code are available on-line at http://opensource.samsung.com, just click on Mobile Phone and search for GT-P7510 wifi; that is, search for Galaxy Tab (GT) and the version for our tablets, P7510 wifi only; e.g., (GT-P7510_OpenSource.zip) is version 3.1, it is also available in the public directory for CIS 520. If you unzip that file, you will have two gzipped tar files; the kernel source is stored in Kernel.tar.gz. Those files are extracted to a sub-directory called kernel. For this project, we'll modify the kernel and develop two new applications using the Eclipse IDE and the Android ADT. On the lab machines (and most likely on your own machine) you'll need to download and install the Android Development Tools (ADT) Plugin for Eclipse – see http://developer.android.com/sdk/eclipse-adt.html. For Android development, there is a wealth of information and sample projects that will help you to get started. There is a modified AccelerometerPlay example that shows how the accelerometer sensor input can be used to control a graphical display. You can download all of the other examples for Android 3.1 using the Android SDK Manager. The purpose of this assignment is to help you explore Android's capabilities for using phone/tablet sensors to gather information about the phone/tablet state and the user's behavior and context. A second goal is to understand the workings of the Android OS, and how the applications interact with the operating system. A nice introduction to Android is available online at: http://www.youtube.com/watch?v=h3gPo7qFOFw.

Part 1: Marble Game

Create a new project called MarbleGame with an Activity called AccelerometerPlay.

- The Activity will need some means to start (or restart) the game. For example, you could use a button.
- The game should involve moving a marble or marbles through a random maze as quickly as possible. Optionally, you can add trap holes as shown below in the figure so that the user has to avoid traps.
- You can add sound effects, etc., but don't spend too much time on this (unless you plan to make money on Android Marketplace ;-).
- You can start with the AccelerometerPlay example, available on the Android Developer website, and you can use any reasonable maze generator; e.g., your maze cannot have just one square – for example, you could use a randomized Prim's algorithm or a recursive backtracker to generate the maze. The maze can be visible or hidden, and game modifications are welcome.



Part 2: TimedAccelerometerPlay - Adding Intents and Services

Extend the project by adding a Timing Service. When the game is started, a Timing Service should be notified to start the stopwatch, and when the user completes, the Timing Service should be contacted to obtain the time that was required to complete the game. If the user is taking too long, the Timing Service should be allowed to issue a warning and later terminate the Activity.

- The Activity will need some means to start and stop the background process running the timer logic. For example, you can use a Button to toggle the state of the alarm; e.g., whether it is started or not.
- Create a Service called TimingService. It should run in the background and host the timer logic. Your Service should post an ongoing notification (e.g., time expired) which cannot be cleared by the user. This notification should only disappear when the Service is shut down by the user by completing the game. Hint: Notification.FLAG_ONGOING_EVENT and Notification.FLAG_NO_CLEAR may be worth a look.
- Design and implement the sensor logic needed to trigger the warning. This should all be done inside the Service. Which sensor you use for this is up to you, but we suggest you use the timer and the accelerometer or the orientation sensor. Your logic should recognize no movement (which we could arbitrarily define as little change in sensor readings for a period $\Delta m > 10sec$). Movements within some time ($\Delta m < 10sec$) should not cause an alarm.
- After the warning has been activated, the user should have a certain period of time (Δt) during which he or she can make progress to disable the warning.
- When Δt has elapsed, the tablet should ring an alarm (i.e., play a sound file) and kill the Activity.
- Be aware that when the phone is in a low power state, it behaves differently than when it is not; e.g., consider wakelocks.

Extra Credit: (+5 pts) Add Traps to New Instances of the Activity

Add traps that allow a user to be presented with a new maze to be solved by instantiating a new instance of the TimedAccelerometerPlay Activity. If the user is able to solve the puzzle, then they should be returned to the previous instance with an opportunity to continue where they left off (just beside the trap). Alternatively, allow following through a trap to create a new instance on another tablet – add some network communication. Just don't spend too much time on this deviation from real work.

Part 3: Android OS

For this part of the project, we will modify the existing kernel and add some new system calls. Then, we will write some test applications to test the new system calls. To debug on an actual device or virtual device, it is useful to use the Android Debug Bridge: http://developer.android.com/tools/help/adb.html . On the tablet side, you will need to enable debugging; e.g., on Honeycomb, select Applications + Development + click on USB Debugging. Then, from a Windows command prompt, you should be able to use commands like: adb devices – to see the list of devices attached, adb shell – to start up a shell on the device being debugged, etc.

The kernel source is located in the public directory in /pub/CIS520/GT-P7510_OpenSource.zip. As noted above, extract the files to create your own copy of the "kernel" source. In the kernel sub-directory, you will need to configure and build the source. The configuration files are found in the arch/arm/configs subdirectory. In the kernel sub-directory, just type: **`make samsung_p4wifi_defconfig`** to create the .config file for a kernel to run on the Galaxy Tabs. However, QEMU won't quite emulate our kernel, but you can test by emulating the Versatile Express with ARM CPU using **`vexpress_defconfig`** as describe in: **http://balau82.wordpress.com/2012/03/31/compile-linux-kernel-3-2-for-arm-and-emulate-with-qemu/.**

In either case, edit the Makefile to specify the location of the cross-compiler; e.g., set:

CROSS_COMPILE := /pub/CIS520/usr/arm/bin/arm-angstrom-linux-gnueabi-

Note that this is a prefix for the binaries in the bin folder, so the last dash on the previous line is correct.

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-angstrom-linux-gnueabi-gcc* in your PATH or you can use the absolute path shown above). Finally, compile the kernel using: **`make -j4 ARCH=arm`**

Note that the **`-j4`** tells the compiler to use 4 processors to speed up the compilation. 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 –j4 ARCH=arm " This allows you to simply type *armmake* in the root of the kernel directory instead of constantly providing the other make variables. The final kernel image (after compilation) is output to ***`arch/arm/boot/zImage`***. This is the file used to boot the emulator. Boot the Android emulator with your custom kernel using the emulator command: **emulator -avd <NameOfYourAVD> -kernel arch/arm/boot/zImage -ramdisk ramdisk.img -show-kernel**

If there is only one device, you don't have to identify the target with the **-avd** option. Recall that the command **avd devices** will give you a list of all available devices. You might want to make a shell alias for this command as well.

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 above.

The prototype for your system call will be:

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

You should define *struct prinfo* as:

```
struct prinfo {
        long state;                 /* current state of process  */
        pid_t pid;                  /* process id                */
        pid_t parent_pid;           /* process id of parent      */
        pid_t first_child_pid;      /* pid of youngest child     */
        pid_t next_sibling_pid;     /* pid of older sibling      */
        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 **245**.

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.c and kernel/timer.c should provide good reference points for defining your system call.

Test your new system call

Write a simple C program which calls prinfo. 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 print the process name (comm field) followed by its pid in square brackets, its current state (state field), and then the remaining fields of struct prinfo (excluding pid and state). The state should be mapped to a single character output similar to the ps unix utility's process state output:

- S = The task is sleeping
- R = The task is running
- Z = The task is a zombie!

NOTE: The three characters do not completely map to all of the different states a process can move through. Document why you chose to map particular kernel process states to a particular character.

Sample program output:

```
init [1] S, 0,33,0,0
       /system/bin/sh [27] S, 1,0,28,0
       /system/bin/vold [28] S, 1,0,29,0
       ...
       zygote [33] S, 1,61,0,100
              system_server [61] S, 33,0,129,501
              com.android.phone [129] S, 33,0,130,1000
              com.android.settings [130] R, 33,0,131,1000
              ...
```

```
kthreadd [2] S, 0,3,0,0
       ksoftirqd/0 [3] S, 2,0,4,0
       events/0 [4] S, 2,0,5,0
       khelper [5] S, 2,0,6,0
       ...
```

The output of the program should be easy to read, and the program should check for errors such as invalid input or too many / too few arguments. The **ps** command in the emulator or via avd shell 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-angstrom-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 include all of your sources files
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-angstrom-linux-gnueabi-gcc
LD = arm-angstrom-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 using the Android Emulator or Galaxy Tab

In order to run the program you just compiled, you will have to move it to the Android emulator's (or tabs) filesystem. This is done using the multi-purpose Android Debug Bridge (adb) utility. After starting up your Android virtual device (or actual device) with the custom kernel, you can check your connection by issuing: **`adb devices`**. If you see the emulator (or tab) 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` followed by **# /data/misc/exe_name**
or both together using:
`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-angstrom-linux-gnueabi-gdb exe_name`**
(gdb) target remote localhost:1234
>

Investigate the Android process tree

1. 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.
2. 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.
3. Notice that on the Android platform there is a process named ***zygote***. Investigate this process and any children processes:
    a. What is the purpose of this process?
    b. Where is the *zygote* binary? If you can't find it, how might you explain its presence in your list of processes?
    c. 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.