Lab 2: ARM Programming

18–342 Fundamentals of Embedded Systems

Released: September 29, 2014, 11:59pm EDT
Part 1 Due: October 6, 2014, 11:59pm EDT
Part 2 Due: October 20, 2014, 11:59pm EDT

Contents

1 Introduction

1.1 Overview

Lab two consists of two parts which covers programming simple C applications, and the full software stack underneath them, down to the interrupt level. In particular, the goals for the first part of the lab includes:

- Writing pure assembly Linux applications with no C code.
- Implementing a partial C standard library (libc).
- Writing Linux applications in C that link against your library, to create binaries that consist entirely of your own code.

The goals for the second part of the lab includes:

- Writing software interrupts to implement a number of Linux syscalls.
- Implementing a single-task mini-kernel capable of running userspace applications.
- Using your mini-kernel to execute your Linux application binaries unmodified.[1]

While the applications, libraries, and kernel developed in this lab conform to the ARM Linux Application Binary Interface (ABI), we are implementing only a very tiny subset of the overall functionality. While this limits the capability of our applications, we intend for it to be an exposure to "what's going on under the hood." As always, we hope it will also be fun.

1.2 Expectations

You and your partners need to start *both* parts early to ensure that enough time is available to complete the lab. *Do not wait until the last day to start the lab*, you will run out of time.

We again emphasize that all submitted code must compile and execute properly. Code submitted that fails to compile will receive a failing grade. Therefore it is *essential* that all code is tested on the gumstix hardware prior to submission.

Finally, some portion—at least 5%—of the lab's grade will be devoted to code style points, as well as adherence to following the proper submission procedure. In particular, we require all source code to be commented, and we require all files be submitted to the handin directory with the proper directory hierarchy and proper file names, including proper capitalization. If for some reason the mechanism your group uses to transfer files either garbles file names or changes file name case, it is your responsibility to login to a Linux machine (use `unix.andrew.cmu.edu` if you cannot find anything else) and rename the files appropriately.

In short, we expect to be able to upload the code to the gumstix hardware, type "`make`", and be able to execute your applications without any modification or renaming of files. Any modifications we do have to make will result in grade deductions.

In part two you are expected to create additional source files in the `lab2/part2/kernel` subdirectories for your implementation of a SWI handler, syscalls, and kernel code, and add the proper dependencies to the kernel Makefile. You are mostly free to choose your own file and function names, although these names must be appropriate in the context of this lab.

At the end of this lab, you must turn in all source code, including the support code and any new files you create, following the handin procedure specified in Section 7.1.

[1] Well OK, we have to"preload" the ELF executables by stripping the ELF header and add some padding between segments—but all the machine code is still the same.

1.3 Lab Support Code

To ease the process of writing assembly applications and implementing your libc, we have provided support code for this lab which may be downloaded from course web site[2].

Throughout this lab we will ask you to modify the support code in order to implement your lab solution. At the end of the lab, you must turn in these files according to the handin procedure specified in Section 7.1.

Part 1: ARM Linux Programming

2 ARM Linux Assembly

Each Linux process has its own register set and address space in which it may perform calculations. However, processes typically avoid accessing I/O and other peripheral devices directly to avoid contention from competing processes. Instead, all I/O operation including file, disk, and network access is performed by the Linux kernel on behalf of user processes. Kernel services are provided to user processes most often in the form of system calls.

2.1 Linux Syscalls & OABI

A system call is similar to a C function call. On most architectures, syscalls are implemented as traps or software interrupts. A software interrupt is an assembly instructions that forces the processor to enter supervisor mode and jump to a specific instruction in memory (typically a syscall handler). The kernel decodes the syscall, branches to the appropriate kernel function, performs the service on behalf of the user process, and finally switches back to user mode.

User processes are required to make syscalls by following an operating system specific syscall convention, which is one component of a larger Application Binary Interface (ABI). ARM Linux supports two different ABI's, the historical (and currently more popular) Obsolete ABI (OABI), and the emerging Embedded ABI (EABI). While most ARM Linux systems are beginning to transition to the EABI, the gumstix platform still uses the OABI, which is the ABI we will use in this lab[3].

The ARM Linux OABI syscall convention is similar to the ARM Procedure Call Standard (APCS) for C functions with a few differences in argument passing. Syscalls are identified by the syscall number, stored as an immediate value in the lower 24 bits of the `SWI` instruction. A list of syscall numbers and the functions they map to is available in `/usr/include/asm-arm/unistd.h` on the gumstix. For example, the syscall `exit` corresponds to number 0x900001, while the syscall `read` corresponds to 0x900003.

The following table is a comparison of OABI syscall argument passing and the APCS:

Argument #	OABI Register	APCS Register
1	r0	r0
2	r1	r1
3	r2	r2
4	r3	r3
5	r4	[sp]
6	r5	[sp+4]
7	r6	[sp+8]

As in the APCS, OABI syscall return values are returned in r0. Unlike the APCS (which only preserves registers r4–r13 & pc) all register values are preserved across a syscall except r0.

Linux syscalls are documented in section 2 of the Linux manpages.[4] Be aware that Linux returns error values from syscalls different from the method describe in the manpages, this is explained in detail in Section 3.1.1.

[2] Labs/Lab2/lab2/support.tar.gz

[3] verdex-pro boards support (or at least claim to support) EABI but we will not use EABI to be consistent with all the sections

[4] For example, the read(2) syscall would be accessible as "man 2 read".

2.2 Example ARM Linux Assembly Program

The support code for this lab contains an example ARM Linux assembly program, `lab2/part1/exit/exit.S`:

```
#include <bits/swi.h>

        .file   "exit.S"
        .text

        .global _start
_start:
        mov     r0, #42
        swi     EXIT_SWI
```

The `exit.S` contains the minimum code necessary for a valid ARM Linux program—all it does is terminate with exit status 42.

Notice that the source code file for this program has a `.S` suffix unlike previous labs, where assembly code typically had a `.s` suffix. When `gcc` is called with a `.S` file, the source code is first preprocessed with `cpp` (the C preprocessor) before being fed the assembler. This allows for the use of "`#include`" and "`#define`" preprocessor directives in assembly source code.[5] In particular, this code includes the `bits/swi.h` header file which contains defined constants for each syscall you will use in this lab. Please use these constants (e.g,. "`EXIT_SWI`") instead of specifying the syscall number directly.

2.3 Hello World in Assembly

For the first lab exercise, write a pure assembly version of "Hello world!" as `lab2/part1/hello/hello.S` using the `write` and `exit` syscalls. Use the `exit.S` source code as a starting point, and consult the *GNU Assembler Programming Tips* document on the course web site for tips of defining strings in assembly. You may also want to take a look at `lab2/part1/libc/include/bits/*.h` for other defined constants that may be useful in your program.

As in lab1, the "Hello world!" program should:

- Write the string "`Hello world!`" followed by a new line to *stdout*.
- Terminate with exit status 0.

Although it is a good practice to verify the return value of all syscalls, you may ignore the return value for `write` syscalls in this program. Doing so assumes that there will not be a short write, but also dramatically simplifies the assembly code.

Please remember to compile your "Hello world!" program with the included Makefile, and verify that the output is proper when executed.

3 Writing a C Library from Scratch

A statically-linked C library has four main components that form the basis of an operating system's C support code:

- A set of syscall wrappers that provides functions to interface C language code with each system call. The actual work is performed by the syscall (or the handler for the syscall), the wrapper is used to interface with user C code.
- A set of utility functions that use operating system specific syscall wrappers to implement the ANSI C standard library API (e.g., printf, strcmp, etc.).

[5] Source code generated by GCC is already preprocessed, so GCC outputs a `.s` file that gets passed to the assembler directly when `gcc` is later invoked on it.

- A set of header files that declare type definitions and function prototypes for the standard library.
- An execution startup routine (`crt0.o`) that prepares arguments for `main`, calls `main`, and calls `exit` once main is finished.

The support code for this lab already contains a set of header files (`lab2/part1/libc/include/*`) containing the prototypes for each of the functions you are required to implement in your libc. For the sake of simplicity, we will not require that you implement any of the ANSI C standard library functions. Thus, the only two major components that remain unimplemented are the syscall wrappers and `crt0.o`.

3.1 Syscall Wrappers

Syscall wrappers provide C-callable functions that invoke a particular system call. Nearly every syscall supported by an operating system has a corresponding syscall wrapper. These wrappers are necessarily implemented in assembly, and perform the these tasks:

- Convert function arguments from APCS to OABI conventions.
- Invokes a syscall via a `SWI` instruction.
- Checks the syscall return value for the presence of an error, and sets the global `errno` variable to the error value if an error is present.

3.1.1 Linux Syscall Errors

Most Linux syscalls return a positive value or zero on success, and any syscall that returns a negative value indicates an error.[6] However, it is standard convention that libc syscall wrappers return –1 on error, and set the global `errno` to the negative[7] of the syscall return value. For example, if the `read` syscall is called with an invalid file descriptor, Linux will return –9, and the `read` syscall wrapper will return –1 while setting `errno` to 9 (which corresponds to `EBADF`).

3.1.2 Writing Syscall Wrappers

Following the tasks listed above, implement three system call wrappers for your libc:

- `exit` in `lab2/part1/libc/exit.S`.
- `read` in `lab2/part1/libc/read.S`.
- `write` in `lab2/part1/libc/write.S`.

Make sure your syscall wrappers sets the appropriate `errno` value on error. Note that the `exit` syscall never returns, and thus, can't return an error.

3.2 Execution Startup Routine: `crt0.o`

When Linux spawns[8] a new process it creates a new address space, creates new kernel data structures, loads the application code into memory, and jumps to the entry point of the program. At the moment when the new user program begins execution, all registers except sp & pc have undefined values.[9]

The kernel does, however, place three arguments on the userspace stack: `argc` and the `argv` & `envp` arrays. Ignoring `envp`, here is a diagram of the userspace stack when a new process starts:

[6] A few syscalls (e.g., `lseek`) actually can return a negative value on success; however, it is guaranteed any syscall that returns a negative value in the range of –1..–4095 indicates an error. Since none of the system calls used in this lab return negative values on success, it's sufficient to consider any negative value an error.

[7] absolute value

[8] the cumulative effect of a `fork` + `exec`

[9] At least, for the purposes of this lab they are undefined.

Address	Variable
[sp+4(argc)+4]	NULL
[sp+4(argc)]	argv[argc–1]
...	...
[sp+8]	argv[1]
[sp+4]	argv[0]
[sp]	argc

The execution startup routine (contained in `crt0.o`) is a piece of assembly code that configures the process so that it can execute C code in the `main` function. In particular, `crt.o` needs to:

- Generate `main`'s `argc` & `argv` parameters from values placed on the stack by the kernel.
- Call the `main` function.
- If `main` returns, call `exit` with the return value from main as the exit status.

As an example, a partial implementation for `crt0.o` in C would be:

```
void _start(void) {
        int argc;
        char **argv;

        /* Set argc & argv from values on the stack. */

        exit(main(argc, argv));
}
```

3.2.1 Writing `crt0.o`

Following the steps listed above, implement the `crt0.o` routine for your libc by writing assembly code in:

- `lab2/part1/libc/crt0.S`

4 A ROT13 Test Application for the C Library

In this lab, we require you to write one application to test your libc. However, we welcome and encourage you to write additional test applications to test other various aspects of the library.

4.1 The ROT13 Cipher Algorithm

ROT13 (rotate 13 places) is a simple Caesar cipher—a very weak encryption algorithm often used to hide spoilers on the Internet. The algorithm is quite simple: rotate all alphabetic characters of message by thirteen places.

More technically, this involves mapping ASCII values 65–77 (A–M) to 78–90 (N–Z) and vice versa, while mapping ASCII values 97–109 (a–m) to 110–122 (n–z) and vice versa. In other words, the following translation of top row characters to bottom row characters occurs:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm
```

Since ROT13 serves as its own inverse, decrypting ROT13 ciphertext simply requires a second application of the algorithm.

4.2 Writing ROT13

Modify the `lab2/part1/rot13/rot13.c` source file to implement a program that performs ROT13 using the `read` & `write` syscalls with the following behavior:

- Reads a "block" of input on *stdin*.
- Terminates with exit status 0 if zero bytes are read.
- Otherwise, performs ROT13 on the input data.
- Writes the entire ciphertext to *stdout*.
- Indefinitely loops to read another block of input.
- Immediately terminate with exit status 1 upon any syscall error encountered.

Note that while the `read` syscall fills a buffer of fixed size, it can return with a short count, meaning your input "block" is of less size than requested. This "short count" behavior is often desired. For example, when a process's *stdin* is connected to a terminal device (such as the serial console), the `read` syscall transfers a single line of text at a time, returning a short count that corresponds to the size of the line of text. This allows for efficient processing of lines of text of varying size.

Also note that the `write` syscall can also return a short count, although this typically doesn't happen when *stdout* is connected to a terminal device. Even so, you *must* ensure that all of the ciphertext for a particular encrypted block is written before accepting another block of input.

In addition to checking for short counts, you must also test the return value of all syscalls in your program for errors. If an error should occur, your program should immediately exit with exit status 1.

Again, please remember to compile your ROT13 program with the included Makefile, and verify that the output is proper when executed.

4.3 Compiling ROT13 with uClibc

If you suspect your libc implementation has bugs and would like to compile your ROT13 program with uClibc to ensure that there's no bugs in the program itself, you may coerce make into compiling against uClibc with the following command:

```
# make clobber && make "CFLAGS=-O2 -Wall -Werror" LDFLAGS= CRT0= LDLIBS= rot13
```

However, be sure to run "`make clobber`" again before attempting to compile against your libc.

Part 2: ARM "Bare-Metal" Programming

5 Das U-Boot: The Universal Boot Loader

5.1 Introduction

U-Boot is a popular bootloader for Linux on embedded platforms and is the bootloader used on the gumstix. As a bootloader, U-Boot is responsible for the initial hardware setup of the gumstix board before passing control to the Linux kernel. In addition, U-Boot also provides a management console and primitive debug monitor to facilitate the development and debugging of kernels. As part of this monitor framework, U-Boot also provides the ability to execute standalone applications. These standalone applications are able to make use of a limited subset of the U-Boot monitor API, but otherwise run directly on the gumstix "bare-metal" hardware.

5.2 The Bootstrapping Process

U-Boot code and data is stored in the first two sectors (256 kB) of the gumstix's onboard flash. When the gumstix is first powered on, the CPU begins execution at address 0x0, which corresponds to the first sector of flash memory and the start of the U-Boot code.[10] As the bootloader, U-Boot is responsible for preparing the gumstix for booting Linux with these steps:

1. Set the CPU to supervisor mode, disable interrupts, set the CPU clock speed, and reset other clocks/timers.
2. Initialize the SDRAM controller and bring RAM online.[11]
3. Copy U-Boot code from flash to RAM and begin execution from RAM.
4. Setup the rest of the U-Boot environment (exception handlers, stacks, heap, serial console, etc.)
5. Load the Linux kernel from onboard flash or an MMC.
6. Start execution of the kernel.

In order to facilitate the above steps, U-Boot, like most bootloaders, contains a highly gumstix specific sequence of code for performing hardware initialization, as well as rudimentary driver support for the serial and MMC devices. The U-Boot driver code is complete enough so that it can read and load the kernel from a JFFS2 or FAT filesystem, but it cannot write to these filesystems or the MMC device.

5.3 U-Boot Console

The U-Boot console is accessed by interrupting the normal boot procedure by pressing a key at the "`Hit any key to stop autoboot:`" prompt[12]:

```
U-Boot 1.1.4 (Nov  6 2006 - 11:20:03) - 400 MHz - 1161

*** Welcome to Gumstix ***

U-Boot code: A3F00000 -> A3F25DE4  BSS: -> A3F5AF00
RAM Configuration:
Bank #0: a0000000 64 MB
Flash: 16 MB
Using default environment

SMC91C1111-0
Net:   SMC91C1111-0
Hit any key to stop autoboot:  0
GUM>
```

Once at the "`GUM>`" prompt, entering "`help`" will display a list of available commands. Of these commands, the few that are needed to execute standalone programs are:

`mmcinit` Initializes the MMC controller.

`fatls` List the contents of a FAT filesystem directory.

`fatload` Loads a binary file from a FAT filesystem.

`go` Start a standalone application.

`reset` Hard reboot the gumstix.

In addition to these commands, there are a few others that might prove useful when debugging but otherwise aren't required for this lab. If you wish to experiment with U-Boot debugging, take a look at the commands: `cmp`, `md`, `mm`, and `mw`.

[10] Address 0x0 intentionally also corresponds to Reset vector in the exception vector table.
[11] Prior to this, the gumstix can only read from flash ROM and use registers for storage.
[12] Some of the gumstix boards use U-Boot 1.2, therefore, the messages you may acutally see might be slightly different

5.4 U-Boot Standalone Applications

U-Boot standalone applications are ARM assembly or C programs that are capable of making use of a limited subset of U-Boot monitor functions, but otherwise run directly on the gumstix hardware. U-Boot's supporting role in standalone applications has basically three purposes:

- To load the standalone application binary and start execution.
- To provide a minimal set of I/O functions for communicating with the gumstix serial console (FFUART).
- To provide limited post-mortem application debugging.

U-Boot does place a few requirements on standalone applications to ensure that the monitor functions execute properly when called from an application, and to ensure that U-Boot may resume execution when the application exits. Other than these few restrictions, a standalone application is free to alter most of execution environment, including installing exception handlers such as IRQs and SWIs.

5.4.1 U-Boot Standalone Application (Exports) API

U-Boot makes a number of internal monitor functions available to standalone applications. These functions prototyped in `lab2/part2/uboot/include/exports.h` of the support code, and documentation is available for them in the *U-Boot Standalone Application (Exports) API* document on course web site (under Labs/Lab2).

U-Boot launches the standalone application as if it were calling a C `main` function with APCS, that is, command line arguments are passed as `argc` and `argv` variables in r0 and r1 respectively and the link register (lr) contains the return address to the U-Boot monitor. To exit, the standalone application sets an exit status in r0 and sets the program counter to the return address provided.

Instead of being linked into the application binaries directly, U-Boot's exported functions are made available to standalone applications by a custom ABI. Prior to executing the application, U-Boot sets r8 to the address of a U-Boot global data structure which contains a jump table consisting of function addresses for each of the exported functions. The standalone applications themselves are compiled with stub code for each of these functions (contained in `lab2/part2/uboot/stubs.c`). Each of the stub functions fetches the real function address from the jump table referenced by r8 and then branches to the real function.

By compiling against `stubs.c` the exports ABI is made transparent to standalone applications—calls to exported functions are made via APCS like any other. However, this requires that the contents of register r8 be preserved throughout standalone application code, otherwise the location of the jump table will be lost. The Makefiles provided in the support code compile standalone application C code with the "`-ffixed-r8`" option, telling the compiler to preserve r8 for us in C code automatically. However, *it is your responsibility to ensure that r8 is preserved in your assembly code.*

5.4.2 U-Boot Standalone Application Example: Hello World

The support code for this lab contains an example U-Boot standalone application, the ubiquitous "Hello world!" in `lab2/part2/hello/hello.c`:

```
#include <exports.h>

int main(void) {
        puts("Hello world!\n");

        return 0;
}
```

To compile and execute this program:

1. Upload and extract the support code on the gumstix.
2. Run `make` in `lab2/part2/uboot` to generate `stubs.o`.

3. Run `make` in `lab2/part2/hello` to generate `hello.bin`.

4. Transfer `hello.bin` to the FAT partition on the MMC with the following commands for basix boards:

```
# mount /mnt/mmc1
# cp hello.bin /mnt/mmc1
# umount /mnt/mmc1
```

   or the following commands for verdex-pro boards:

```
# cp hello.bin /media/card
# umount /media/card
```

5. Reboot the gumstix and interrupt the boot procedure to get the U-Boot prompt.

6. Load and execute the `hello.bin` program with the commands:

```
GUM> mmcinit
...
GUM> fatload mmc 0 a2000000 hello.bin
reading hello.bin

200 bytes read
GUM> go a2000000
## Starting application at 0xA2000000 ...
Hello world!
## Application terminated, rc = 0x0
```

7. When finished, reboot the gumstix back into Linux with the `reset` command.

In general, the `mmcinit` command **must be issued before any U-Boot operation on the MMC may be performed**. If you're unsure of the what files are available for loading, you may also issue:

```
GUM> fatls mmc 0
   876752   uimage
      349   gumstix-factory.script
      200   hello.bin
```

to see a list of available files.

If you wish to pass command line arguments to a standalone program, you may use append them to the `go` command:

```
GUM> go a2000000 here are some arguments
```

5.5 U-Boot Execution Environment

U-Boot executes on the gumstix in supervisor mode with both IRQs and FIQs masked. The MMU is disabled, so all memory accesses refer to physical addresses. Section 2.12 of [4] shows the full processor memory map for PXA255 Processor on basix boards (Section 28.1 of [5] for the PXA270 Processor on verdex-pro boards), with the gumstix relevant portion reproduced below:

Start Address	End Address	Type
a0000000	a3ffffff	SDRAM (64 MB)
48000000	4bffffff	Memory Controller (MMIO)
40000000	43ffffff	Memory Mapped Registers
00000000	00ffffff	StrataFlash ROM (16 MB)

U-Boot does specify an exception handler for each exception type, but each one results in a system panic—each register is printed to the console and the system is rebooted. Thus, the exception handlers may be freely replaced as U-Boot does not depend on their behavior.

U-Boot code sits in the first two sectors of flash ROM (starting at address 0x0), and early in its execution, U-Boot copies itself into high RAM and executes from there. Once U-Boot initialization is complete, the layout of RAM is:

Start Address	End Address	Type
a3f00000	...	U-Boot Code
a3edf000	a3efffff	Heap (malloc)
a3edee00	a3edefff	U-Boot Global Data Struct
a3ededf4	a3ededff	Abort Stack
...	a3ededf3	Supervisor Stack
a0000000	...	Free

The supervisor stack grows downward from 0xa3edef0 and is the stack used by both U-Boot and standalone applications. All of RAM below the stack is available for standalone application use, as long as a reasonable amount of room is left available for the stack to grow.

6 Mini-Kernel Implementation

6.1 Expected Behavior

For the remainder of the lab you will be implementing various components of your mini-kernel. Some of these are described in their own sections, but the following is an overview of the expected behavior. Upon entry into the kernel's `main` function, it should:

1. "Wire in" your SWI handler.
2. Switch to user mode with IRQs & FIQs masked.
3. Setup a full descending user mode stack (with the stack top at 0xa3000000).
4. Push U-Boot's argc & argv on the user stack.
5. Jump to a loaded user program at address 0xa2000000.

Upon receiving a SWI from a user program, your kernel should:

1. Store all non-banked user space registers.
2. Determine the SWI number called.
3. Dispatch to the appropriate syscall.
4. Restore non-banked user space registers (only r0 should be modified).
5. Return to user space.[13]

Additional expectations are:

1. The kernel follows APCS with respect to U-Boot's calling the `main` function (i.e., callee-save registers must be preserved, exit status in r0, etc.).
2. The kernel follows the ARM Linux OABI syscall convention as specified in part one of the lab.
3. The kernel restores U-Boot's original SWI handler upon exit.

[13] Except in the case of `exit`, which never returns.

4. The kernel does not corrupt any portion of U-Boot that prevents it from running properly after the kernel exits.[14]

5. Userspace registers are preserved across SWIs, except where modifications are expected (e.g., return values).

Also note that the kernel is not allowed to place "unusual" restrictions on user space processes. For example, most kernel functions will probably assume that r8 references U-Boot's global data/jump table, allowing exported functions to work in syscalls. The kernel must not restrict user space code (which does not use the exports API) from modifying r8. Instead, the kernel must arrange for r8 to be properly stored and loaded when switching to and from user mode.

6.2 "Wiring in" a SWI Handler

The exception vector table starts at memory address 0x0. Usually, assigning (or reassigning) the address of a handler function to the SWI vector is sufficient for "wiring it in." However, according to the processor memory map (see Section 5.5), the vector table actually sits in ROM. Since it is not practical (nor safe) to reflash the gumstix to modify the SWI vector, nor is it simple to enable the MMU (this is what Linux does), you must "wire in" the SWI handler via a different means.

Fortunately, the load instructions that U-Boot uses for its exception vectors point to handler functions in the RAM copy of U-Boot. By hijacking the first few instructions of U-Boot's SWI handler, you may redirect SWIs to your own handler.

To use this method in a safe and consistent manner across any version of U-Boot, the implementation of the hijacking process restricts make the assumptions you may make about U-Boot's code. In particular, you are allowed to assume that:

- The location of the SWI vector is a fixed, well known constant in the ARM architecture.
- The SWI vector load instruction points to a valid SWI handler function in RAM.
- The U-Boot's SWI handler function is at least eight bytes long.

Since these are the only assumptions you may make, the implementation of your hijacking function must perform these tasks to ensure correct modification of U-Boot's SWI handler:

- Confirm that the SWI vector contains a "`ldr pc, [pc, #imm12]`" instruction, and correctly identify whether the `#imm12` offset is positive or negative. If it does not, the kernel should print a message to the console stating the instruction is unrecognized and must exit with status 0x0badc0de.
- Determine the address of U-Boot's SWI handler by locating the 32 bit value referenced by the `#imm12` offset in the `ldr` instruction.

Once you have located U-Boot's SWI handler, you may modify it as appropriate so as to transfer control to your own handler. However, you also need to store any modified U-Boot handler code so that the original handler is restored when your kernel exits.

6.3 Pushing U-Boot's argc & argv on the User Stack

Command line parameters specified in U-Boot's `go` command are passed to the kernel's `main` function as `argc` & `argv` arguments. Assume these arguments are intended not for your kernel, but for the user program your kernel launches. This requires that your kernel places the `argc` & `argv` values on the user stack in the manner described by Section 3.2 of the part one handout.

Normally, when Linux places `argv` on a user stack, it must copy all strings referenced by `argv` to the user stack since kernel memory is protected and not readable in user mode. Since the MMU is disabled, there is no memory protection in your kernel. It is OK to reuse the same strings for your user `argv` array, even though they reside in "kernel memory."

[14] Since there's no user space memory protection, assume that no user space code intentionally corrupts U-Boot.

6.4 Implementing Syscalls

Your kernel must implement the following syscalls, their details are presented in the following sections:

SWI Number	Syscall Name
0x900001	`exit`
0x900003	`read`
0x900004	`write`

Additionally, any SWI whose number does not match one of the above should be considered an invalid syscall. The kernel should print a message to the console stating the problem and must exit with status 0x0badc0de.

6.4.1 Exit Syscall

`void exit(int status):`

The `exit` syscall takes a single argument, the exit status. The `exit` syscall should exit the kernel, returning to U-Boot while passing the exit status back. The `exit` syscall never returns to user space.

6.4.2 Read Syscall

`ssize_t read(int fd, void *buf, size_t count):`

The `read` syscall takes three arguments, the file descriptor for reading, a buffer for writing, and (at most) the number of bytes to read. Since the U-Boot API is only capable of reading from *stdin*, the `read` syscall should return error (specifically, `-EBADF`) for any file descriptor that doesn't match *stdin*. `read` should also return `-EFAULT` if the memory range specified by the buffer and its maximum size exists outside the range of writable memory (SDRAM).

Otherwise, `read` should read characters into the buffer from *stdin* and echo the character back to *stdout* until the buffer is full. Once the buffer is full, the syscall should return with the number of characters read into the buffer.

While reading characters from *stdin*, the `read` syscall should process certain special values appropriately. Specifically:

- An EOT character[15] neither gets placed in the buffer nor echoed to *stdout*, but instead the `read` syscall should return immediately with the number of characters read into the buffer thus far. For example, if EOT is the first character read, nothing is written in to the buffer and 0 is returned.
- A backspace[16] or delete[17] character neither gets placed in the buffer nor echoed to *stdout*. Instead the previous character should be removed, and the string "`\b \b`" should be printed to *stdout*.[18]
- A newline[19] or carriage return[20] character results in a newline (only) being placed in the buffer and echoed to *stdout*. Additionally, the syscall should return with the number of characters read into the buffer thus far (including the most recent newline).

6.4.3 Write Syscall

`ssize_t write(int fd, const void *buf, size_t count):`

The `write` syscall takes three arguments, the file descriptor for writing, a buffer for reading, and the number of bytes to write. Since the U-Boot exports API is only capable of writing to *stdout*, the `write` syscall

[15] ASCII value 4, produced when "C-d" is typed.
[16] ASCII value 8, "\b" in C.
[17] ASCII value 127.
[18] This ensures that the previous character is erased from the console.
[19] ASCII value 10, "\n" in C.
[20] ASCII value 13, "\r" in C.

should return error (specifically, `-EBADF`) for any file descriptor that doesn't match *stdout*. `write` should also return `-EFAULT` if the memory range specified by the buffer and its maximum size exists outside the range of readable memory (StrataFlash ROM or SDRAM).

Otherwise, `write` should write characters from the buffer to *stdout* until the buffer is empty. Once the buffer is empty, the syscall should return with the number of characters written to *stdout*.

6.5 Testing argc and argc

In order to test whether your stack is being set up correctly, we would like you to modify the ROT13 program to take in an arbitrary number of parameters when it is instantiated and print them out to the screen before doing the ROT13 cipher. When uploading the files you only need to turn in this version of ROT13.

6.6 Testing the Kernel

The kernel Makefile links the kernel at address 0xa3000000. The range of memory from 0xa3000000 to the shared kernel/U-Boot supervisor stack is considered "kernel space".[21] This leaves the entire address range from 0xa0000000–0xa2ffffff available for user applications, and indeed the user space stack is placed at the top of this range.

Testing the kernel involves loading the kernel binary, loading a user application binary, and starting execution at the kernel entry point. All applications developed as part of part one (`exit.bin`, `hello.bin`, and `rot13.bin`) should be compatible with your kernel and function as they do under Linux. We encourage you to develop more test applications by copying the skeleton code for one of the part one programs.

To compile and execute test programs from part one:

1. Upload the completed part one solution code to the gumstix.
2. Run `make` in `lab2/part1/libc` to generate `crt0.o` and `libc.a`.
3. Run `make` in the `lab2/part1` application directories to generate `.bin` files.
4. Run `make` in `lab2/part2/uboot` to generate `stubs.o`.
5. Run `make` in `lab2/part2/kernel` to generate `kernel.bin`.
6. Transfer `kernel.bin` and any application binaries to the FAT partition on the MMC with the following commands[22]:

   ```
   # mount /mnt/mmc1
   # cp kernel.bin ../../lab2/part1/*/*.bin /mnt/mmc1
   # umount /mnt/mmc1
   ```

7. Reboot the gumstix and interrupt the boot procedure to get the U-Boot prompt.
8. Load and execute the `rot13.bin` program with the commands:

   ```
   GUM> mmcinit
   ...
   GUM> fatload mmc 0 a3000000 kernel.bin
   reading kernel.bin

   1028 bytes read
   GUM> fatload mmc 0 a2000000 rot13.bin
   reading rot13.bin
   ```

[21] Although the distinction is not overwhelmingly important since there is no memory protection.

[22] Modify these commands, as shown earlier, for verdex-pro boards

```
308 bytes read
GUM> go a3000000
## Starting application at 0xA3000000 ...
abjurer
nowhere
## Application terminated, rc = 0x0
```

9. You may load another user program without having to reload `kernel.bin`:

```
GUM> fatload mmc 0 a2000000 hello.bin
reading hello.bin

40 bytes read
GUM> go a3000000
## Starting application at 0xA3000000 ...
Hello world!
## Application terminated, rc = 0x0
```

10. When finished, reboot the gumstix back into Linux with the `reset` command.

Also note that early in the development process, you may want to test a kernel that does not yet execute user code. To do so, simply skip loading a user application in the above steps.

6.7 Plan of Attack

You are free to implement components of your kernel in any order you wish. However, if you're not quite sure where to start, here are some suggestions:

First, copy the `lab2/part2/hello` project directory and try writing other standalone applications to familiarize yourself with the more useful functions of the U-Boot exports API. Some kernel components (e.g., read & write syscalls) may be developed and tested in this environment without the need for user mode transitions or a SWI handler.

Next, try developing a SWI handler & dispatcher. "Wire in" the SWI handler using a brute force method and hardcoded values first to make sure it works, before you attempt to do instruction decoding to "wire in" the SWI handler properly.

You may test SWIs from supervisor mode, all that needs to be done is to execute a `swi` instruction from assembly. Beware that, if you execute a SWI from supervisor mode, the link register (lr) will not be banked out, and will likely be modified on return from the SWI—so preserve it somewhere in your test code first.

Once your SWI handlers and syscalls are working, write code to enter user space and setup the user space stack. Try using 0 values for `argc` and `argv` first. Once the user space transition code works, try putting the real `argc` and `argv` values on the stack.

By this time, you should be ready to implement the `exit` syscall. It could be implemented sooner, but depending on the approach you take for implementing it (there are multiple right approaches), it might be easiest to finish at the end.

Once `exit` is finished, clean up your code and do a final scan looking for correctness and compliance issues.

Finally, test your kernel extensively by running the part one applications, and writing additional applications. Not all of the kernel functionality is tested by the applications given or developed in part one.

7 Completing the Lab

7.1 What to Turn In

When finished with the lab, please submit the following source code and project files in an archive lab2-XX-partY.tar.gz, where Y=1 for part 1 and Y=2 for part 2, XX is your Andrew ID (maintain the directory

paths in the archive if you donot want to lose points), on Blackboard. For part 1 submission, you can use the unmodified part 2 directory from support code in the archive you submit. .

- `lab2/part1/hello/Makefile`
- `lab2/part1/hello/hello.S`
- `lab2/part1/libc/Makefile`
- `lab2/part1/libc/crt0.S`
- `lab2/part1/libc/errno.c`
- `lab2/part1/libc/exit.S`
- `lab2/part1/libc/include/bits/errno.h`
- `lab2/part1/libc/include/bits/fileno.h`
- `lab2/part1/libc/include/bits/swi.h`
- `lab2/part1/libc/include/bits/types.h`
- `lab2/part1/libc/include/errno.h`
- `lab2/part1/libc/include/stdlib.h`
- `lab2/part1/libc/include/sys/types.h`
- `lab2/part1/libc/include/unistd.h`
- `lab2/part1/libc/read.S`
- `lab2/part1/libc/write.S`
- `lab2/part1/rot13/Makefile`
- `lab2/part1/rot13/rot13.c`
- `lab2/part2/kernel/Makefile`
- `lab2/part2/kernel/include/bits/errno.h`
- `lab2/part2/kernel/include/bits/fileno.h`
- `lab2/part2/kernel/include/bits/swi.h`
- `lab2/part2/kernel/kernel.c`
- `lab2/part2/kernel/start.S`
- `lab2/part2/uboot/Makefile`
- `lab2/part2/uboot/include/exports.h`
- `lab2/part2/uboot/include/bits/types.h`
- `lab2/part2/uboot/include/exports.h`
- `lab2/part2/uboot/include/stdarg.h`
- `lab2/part2/uboot/stubs.c`

Please also submit any additional source files that you may have created and that are required for successful compilation of your project code.

You do not need to submit the `lab2/part2/hello` subdirectory, but please do submit the entirety of the `lab2/part2/kernel` and `lab2/part2/uboot` subdirectories as well as any additional source files that are required for successful compilation of your project code.

7.2 Where to Get Help

The user manual for the GNU assembler is available on the web [3] and is linked from the GNU Binutils website [2]. The gas manual documents the syntax of all assembler directives and most ARM specific features, although the document is incomplete in a few areas.

The ARM instruction set is fully documented in the *ARM v5TE Architecture Reference Manual* [1].

Tips on defining strings in assembly and other gas features is available from in the *GNU Assembler Programming Tips* document on the course website.

The Intel PXA255, Intel PXA270 and XScale developer manuals [4], [5] [6] contain "more than you wanted to know" about OS level programming for the PXA processor. You probably don't need to consult these references, but it's possible that a diagram or two (such as the full memory map) might come in handy.

The *ARM v5TE Architecture Reference Manual* [1] contains instruction encodings for each instruction. This may be useful when attempting to decode instructions.

The U-Boot Exports API is documented in the *U-Boot Standalone Application (Exports) API* document on the course website.

References

[1] ARM Limited. *ARM Architecture Reference Manual*, June 2000. Available from: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html`.

[2] Free Software Foundation. GNU Binutils [online]. Available from: `http://www.gnu.org/software/binutils/`.

[3] Free Software Foundation. *Using as*, Aug. 2007. Available from: `http://sourceware.org/binutils/docs-2.18/as/`.

[4] Intel Corporation. *Intel PXA255 Processor Developer's Manual*, Jan. 2004. Available from: `http://pubs.gumstix.com/documents/PXADocumentation/PXA255/PXA255ProcessorDevelopersManual[278693-002].pdf`.

[5] Intel Corporation. *Intel PXA27x Processor Developer's Manual*, Oct. 2004. Available from: `http://mmpod.googlecode.com/files/Intel_PXA270_Developers_Manual.pdf`.

[6] Intel Corporation. *Intel XScale Core Developer's Manual*, Jan. 2004. Available from: `http://pubs.gumstix.com/documents/PXADocumentation/XScale/XScaleCoreDevelopersManual.pdf`.