Lab 2: Timers, Interrupts and System Calls

You need to spend time crawling alone through shadows to truly appreciate what it is to stand in the sun.

14-642 Fundamentals of Embedded Systems

Checkpoint Due: 11:59PM EST October 11th, 2016
Final Due: 11:59PM EST October 18th, 2016

Contents

1 Introduction

1.1 Overview

The goal of this lab is to build the first part of your kernel that provides user space isolation, interrupts, and the ability to host a simple audio application. In the last lab, you implemented a cyclic-executive style (single infinite loop) program that ran entirely in supervisory mode. In this lab, you will create a kernel capable of running arbitrary programs in user space. Your new kernel will be able to handle requests from user processes via *system calls*. Finally to test your kernel, you'll write a frequency detection program using a few custom defined system calls that is similar in spirit to a guitar tuner.

1.2 Grading

Your kernel should meet the following requirements:

- Checkpoint 1 **(10 points)**
- Working IRQs with the ARM timer **(10 points)**
- User mode isolation with working SWI handlers **(25 points)**
- All system calls, except for the ADC system calls **(15 points)**
- ADC system calls, correctly implemented with correct IRQ handler integration **(15 points)**
- User space application to detect frequency of an audio signal and print it to the serial console (ftditerm) **(15 points)**
- Style **(10 points)**

1.3 Starter Code

Much of this lab depends on your ADC driver and UART implementation from lab 1. You will need to merge the new started code into your existing gitlab repository.

Updating Your Repository

Refer to the *getting updates* section of `gitlab.pdf`.

Compiling and running

We've added a new flag to your `Makefile` that allows you to specify the loading of a user program:

```
make USER_PROJ=<directory of user program> gdb
```

The above command does the following:

- Compiles your kernel and user program (specified by the `USER_PROJ` flag)
- Start `gdb`, and load both the kernel and user ELF files into the Raspi's memory.

You can then debug and run your program like the previous labs. For example, to load your kernel with the frequency detector program, run:

```
make USER_PROJ=tuner gdb
```

Once in GDB, don't forget to set breakpoints or type `continue` to launch the program.

2 IRQs and ARM timer

The first step is to implement an IRQ handler. This will require updating the ARM vector table, writing an IRQ handler, and configuring a timer that will eventually trigger the IRQ. In our case, the IRQ will be used to sample audio from the microphone.

2.1 Installing the Jump Table

By default, the RPi loads the kernel at `0x8000`. As discussed in lecture, the ARM processor responds to interrupts by setting the program counter to the corresponding entry in an interrupt vector table. This vector table starts at address `0x0`. For example, if a software interrupt occurs, the processor will set `pc` to `0x08`. The instruction at `0x08` will then load the address of the actual interrupt handler into `pc` i.e jumping to the handler. When writing interrupt handlers as an embedded designer, you will usually have to write some assembly since every CPU architecture handles interrupts differently. We have already defined an ARM jump table for you. You need to load it at `0x0`. Look inside of `kernel/src/supervisor.S`. You need to fill out the assembly for the function `install_interrupt_table`. This function should do the following:

- Get the address of the `interrupt_vector_table` assembly label in a register.
- Store all 8 `ldr` instructions starting at `0x0` upward. Don't worry about the order of the handlers in the table. The starter code provided already matches what the ARM CPU expects for each interrupt.
- **Fill in the remaining**, 7 soft vector table labels.

You should compile your kernel and look at `kernel.asm` to get a sense for how the C source maps to assembly. In this file, you should find the `interrupt_vector_table` label, the vector table and the soft vector table. This method is used to jump to a full 32-bit address of an interrupt handler with the limitation of the small address field offset in the `ldr` instruction.

2.2 Writing an IRQ ASM handler

After you installed the vector table correctly we need to fill out the `irq_asm_handler`. This routine will be called when an ARM IRQ interrupt occurs. The handler needs the following:

- Set the stack pointer to the top of the IRQ stack. The provided linker script has defined the label `__irq_stack_top` for you to use in your code (see Section 5: Assembly Tips and Calling Conventions for more info).
- Save the register context and the address of where you need to return to on the stack.
- Branch *and link* to the `irq_c_handler` (defined in `kernel/src/kernel.c`).
- Restore register context off the stack.
- **ATOMICALLY**, set the pc where you need to return to and move the spsr into the cpsr. If you don't do this, then you could be interrupted while trying to return from an interrupt (bad things happen).

2.3 ARM Timer

Now we need to configure the ARM timer to generate an IRQ interrupt to see if the handler is working correctly. To do this, we first need to configure the ARM interrupt controller using its MMIO interface. The description of the interrupt controller starts on page 109 of the BCM2835.pdf datasheet. In a nutshell, the interrupt controller has 3 main registers that you should focus on: IRQ basic pending, Enable Basic IRQs, Disable Basic IRQs. The MMIO map for these is on page 112, The rest of the registers handle GPU interrupts and FIQs which we are not handling in this class. In each of these registers, you should look for which bit handles enabling, disabling, and pending ARM timer IRQs.

Next, we need to look at how to set up the ARM timer to generate IRQs at the rate we want. The description of the ARM timer starts on page 196 of the BCM2835.pdf datasheet. The function definitions in `kernel/include/timer.h` describe the interface you need to implement, which is in `kernel/src/timer.c`.

After you implement this, you can put a simple `printk()` in the `irq_c_handler` to see if you can receive interrupts! You will need to call `enable_interrupts()` defined in `349libk/include/arm.h` to enable IRQ interrupts in the CPSR since they are disabled by default when the kernel boots. Just being able to generate IRQs is enough for now. Later in this lab, we will have our IRQ handler sample the ADC.

2.4 IRQ Tips and Tricks

- For loading the `interrupt_vector_table`, look at the `stm` (store multiple) and `ldm` (load multiple) ARM instructions.
- Always think about calling conventions when writing assembly (see Assembly Tips and Calling Conventions for more info)!
- Look into the `movs` ARM instruction. It will be helpful for interrupt handlers.
- Ignore all bits in the timer control register that do not pertain to running in 32 bit mode, enabling interrupts, setting the prescaler, and enabling the timer.
- Don't forget to check the errata!
- Don't forget to clear the ARM timer interrupt in `irq_c_handler`.
- Don't forget about the ARM pc offset when figuring out where you need to return to in your `irq_asm_handler`.
- Look into the `ldr <register>, =<label>` ARM instruction for how to load an assembly label into a register.

CHECKPOINT *DUE: 11 Oct 11:59 PM*: You must submit a version of your kernel that successfully handles ARM timer IRQs and simply prints `"IRQ!"` every second. See Section 6: Submission for how to tag this version of your kernel on gitlab.

3 Supporting a User Mode

Now that we have IRQs, we can setup the infrastructure to provide a user mode. At that point, we have an actual kernel! You will implement system calls for your kernel which will allow for user mode programs access to kernel data structures. Before we continue, you will need to understand how user programs are loaded and the typical execution flow of a user program. As you learned in lecture, the ARM processor supports different modes of operation. Your kernel is loaded at memory address `0x8000` and runs in *supervisor* mode, while your user program will be loaded at memory address `0x300000` and should only run in *user* mode. This prevents user mode programs from accessing reserved registers, arbitrarily changing processor mode, etc. Whenever a user program needs access to system resources, it must make a request via a *system call* to the kernel.

Ideally, peripherals and resources like UART, I2C, and system timers should be managed by the kernel and only accessible to user programs through these system calls [1]. In ARM, these system calls will be implemented via *software interrupts* (SWI).

3.1 Entering User Mode

Remember, your kernel is loaded at address `0x8000` while user programs are loaded at address `0x300000`. The RPi CPU will first begin executing at `0x8000` i.e your kernel. Your kernel should set up user space and jump to the user mode program at `0x300000`. This process will need to be implemented in the function `enter_user_mode` defined in `code/kernel/src/supervisor.S`. Setting up user mode consists of the following:

[1] We won't be implementing memory protection in our kernel, so technically your user programs will still have access to peripherals through MMIO.

1. Saving the current supervisor context on the kernel's stack.
2. Changing the processor mode to user mode in the CPSR and enabling IRQ interrupts.
3. Setting the user mode stack pointer to the linker defined label `__user_stack_top`.
4. Jumping to where user code is loaded, which is defined by the `__user_program` linker label.

Your kernel should then call this method to begin executing a user program after it has done all other initialization needed (UART, ADC, IRQs, etc).

3.2 Implementing System Calls for `newlib`

Now that we have a way to launch programs in user mode, we need system calls for user mode programs to communicate with the kernel. The SWI numbers and syscalls that you need to support are defined in `code/349libk/include/swi_num.h` and `code/kernel/include/syscalls.h` respectively. Your user programs will be linked with `newlib`, a popular C standard library implementation intended for use on embedded systems. This will allow your user programs to use any libc function, such as `printf`, `malloc`, etc. These library functions will use your system calls via assembly stub functions, defined in `code/newlib/349include/swi_stubs.c`. Each stub just initiates a software interrupt with the appropriate SWI number we have defined for you in `code/349libk/include/swi_num.h` and then returns the result to the caller (see Calling Conventions for more information on return values). You need to fill in all these stubs. This is an extremely common bootstrapping process when bringing a system up on a new platform.

Now you'll need to implement the syscalls defined in `code/kernel/include/syscalls.c`. These are the kernel system call routines. We have given you a few dummy implementations for the syscalls that we do not care about (for example, ones that use a file system). We have also implemented the system call `syscall_sbrk` for you. This is used by `malloc` to increase the size of the user heap. This lab is challenging enough without having to deal with user heap issues in `malloc`. For now, you'll have to implement the following:

```
void syscall_exit(int status);
```

The `exit` syscall is called when the user program has finished executing, or wishes to exit early on purpose. An Example of this is seen in `crt0.S`. Historically, there is an assembly file called `crt0.S` which is used by the kernel to call the user program's `main()`. This file is located at `newlib/349include/crt0.S`. Notice how after the branch to user `main`, the `exit` system call is called if the user program returns with a `swi` instruction. Your system call implementation for `exit` should print out the exit status of the user program and hang with interrupts disabled. Since we are not requiring you to load user programs into memory, we will not require your kernel to handle a user program returning.

```
int syscall_read(int file, char *ptr, int len);
```

Since we do not have a filesystem, your `read` syscall should only support reading from `stdin` and return -1 if this is not the case. For the Pi, `stdin` is treated as the serial console via UART. To mimic traditional behavior of reading from `stdin`, `read` should read bytes one by one and echo each byte back as they are read. It should also process certain special bytes differently:

- An end-of-transmission character (ASCII value 4) notes the closure of the stream our characters are coming from. The syscall should return immediately with the number of characters previously read.
- A backspace (ASCII value 8) or delete (ASCII value 127) character neither gets placed in the buffer nor echoed back. Instead the previous character should be removed from the buffer, and the string "\b \b" should be printed (this erases the previously written character from the console).
- A newline (ASCII value 10) or carriage return (ASCII value 13) character results in a newline being placed in the buffer and echoed back. Additionally, the syscall should return with the number of characters read into the buffer thus far (including the most recent newline).

`int syscall_write(int file, char *ptr, int len);`

Since we do not have a filesystem, your `write` syscall should only support writing to `stdout` and return -1 if this is not the case. For the Pi, `stdout` is treated as the serial console via UART. `write` can simply output each byte in the buffer to the serial console.

SWI Assembly Handler

After implementing these C system calls, we need only one more piece of the puzzle to make system calls work. We need an assembly SWI handler that calls a C SWI handler and figures out which system call the user wants. This process is very similar to what we did for IRQs with some subtle differences. Every system call is generated with the `swi` instruction which generates a software interrupt. The ARM CPU will then execute the assembly handler `swi_asm_handler`. This handler needs to do the following:

- Support nested SWIs. Your handler must be re-entrant.
- Support IRQs during SWIs.
- Use the link register (plus some offset) to get the address of the `swi` assembly instruction in user mode. Then bit mask this instruction to get the SWI number the user requested. Pass this in as argument one to your `swi_c_handler`.
- Pass in the pointer to the user system call arguments as argument two for your `swi_c_handler`.

Once you write this assembly handler, you can write the `swi_c_handler` (in `kernel/src/kernel.c`) as one huge switch statement on the user's SWI number and call your kernel C system call functions. If all goes well, you should have a fully functional kernel, capable of running user programs that use standard libc functions! Try playing around in `code/tuner/src/main.c` and using C library functions like `printf`, `read`, `write`, etc. to verify that your kernel and system calls are working correctly.

User Mode and SWI Tips and Tricks

- You can't directly operate on status registers (CPSR, SPSR) like you can with normal registers. You'll need to use the `msr` and `mrs` instructions instead.
- When debugging system calls, having IRQs running will just make your life harder. It would be best to leave interrupts disabled during this.
- To handle nested SWIs, you will need to save the SPSR on the kernel stack along with the register context.

3.3 349libc ADC Custom System Calls

Now that we have working IRQs and system calls, we need to add system calls to allow a user program to actually use the timer. We've defined a simple interface for users to schedule periodic sampling of the ADC. You may have already noticed these functions while implementing your SWI handlers. The two system calls are as follows (defined in `code/kernel/src/syscalls.h`):

```
/**
 * @brief Starts sampling the ADC periodically, calling the given callback
 *        with the ADC sample value.
 *
 * @param freq      frequency at which to sample
 * @param channel   channel to sample
 * @param callback  function to be called every time a sample is read from the adc.
 *
 * @return 0 on success or -1 on failure
 */
int syscall_sample_adc_start(int freq, uint8_t channel, void (*callback)(uint16_t));
```

```
/**
 * @brief Stops periodic sampling of the ADC.
 *
 * @return 0 on success or -1 on failure
 */
int syscall_sample_adc_stop();
```

`syscall_sample_adc_start` should start timer IRQs and call the given callback at the specified frequency inside the IRQ C handler. Each time the callback is called, an audio sample should be read from the ADC and passed to the callback. `syscall_sample_adc_stop` should stop the timer. You need to fill out the C system call kernel stubs in `kernel/src/syscalls.c` and add the callback to your C IRQ handler along with a call to your ADC driver to read the microphone.

NOTE: The callback routine you pass into the `syscall_sample_adc_start` system call will run in SUPERVISOR MODE. Therefore, you should not be calling any system calls (like `printf()` would) inside this function. You can also not use `printk()` since the callback is compiled against the user library `newlib`. If you absolutely need to debug in this callback, try using a global variable and print this global elsewhere in your user space application or use GDB with breakpoints.

4 User mode program

You will implement a simple peak frequency detector to test the kernel functionality that you just implemented. This user application will use the ADC driver through your kernel system call and IRQ interface to figure out the frequency of the sound the microphone is hearing.

4.1 Frequency Detection using Real-input Discrete Fourier Transforms

We've provided you a simple Fourier transform library called KissFFT in `code/tuner/src/kiss_fft` and `code/tuner/include/kiss_fft`. Here's an example of how to use the library to perform a real-input discrete Fourier transform:

```
kiss_fft_scalar input[NUM_SAMPLES];
kiss_fft_cpx output[NUM_SAMPLES/2+1];
kiss_fftr_cfg cfg = kiss_fftr_alloc(NUM_SAMPLES, 0, 0, 0);

// ...fill input with sample data

kiss_fftr(cfg, input, output);
for (int i = 1; i < NUM_SAMPLES/2+1; i++) {
  // do something with fft_output[i].r and fft_output[i].i
  // which are the real and imaginary parts of the output
}
```

We provide a more comprehensive example project that performance peak detection below.

4.2 Guitar Tuner

To demonstrate all of the components of the lab, you will implement a basic guitar tuner in `code/tuner/src/main.c`. The program should periodically sample the microphone using the syscalls you implemented earlier (in `code/newlib/349include/349libc.h`). The sampling should be periodic and interrupt driven. The main function in the tuner application should loop infinitely and perform FFTs on any incoming audio data. We require that

your application process audio continuously in a streaming fashion. Double buffering is a common technique that will allow your program to perform processing while microphone data is being collected in the background. In a double buffered system, an interrupt should fill one buffer while your application processes a second buffer. Once the interrupt is finished loading data in to the first buffer and the new buffer is ready (has been processed), the buffers should be seamlessly swapped. This requires that the application process the previous buffer faster then the sampling function loads the new buffer (the consumer must be faster then the producer). For the tuner app, we recommend using a sampling frequency of 1598Hz with 1600 samples per FFT.
You can find a detailed example of how to perform an FFT and then search for a peak frequency value in `code/fft_demo/src/main.c`. Run the example program against your kernel with the following command:

```
make USER_PROJ=fft_demo gdb
```

This program creates a buffer filled with a synthetic 82kHz sine wave (E note). It then performs an FFT followed by a peak frequency detection step. The value displayed over ftdi term is the peak frequency found in that audio sample. Your tuner application should build on this by continuously printing out the frequency of live captured audio after each FFT finishes as shown below. This is an example output if 329Hz E note is given as an input:

```
freq=97.181250
freq=97.181250
freq=97.181250
freq=329.700000
freq=329.700000
```

In the standard tuning, the six strings on a guitar should have a peak frequency at 82.4,110.00,146.83,196.00,246.94 and 329.63 Hz representing the notes E,A,D,G,B,E. For testing, we recommend downloading a tone generator app on your computer or smartphone. Note, phone and computer speakers will not accurately generate low frequencies, so try testing the higher notes. A real instrument should generate the full range of frequencies, but we will only test on frequencies above 200Hz. It is also worth noting that the example program implements a low-pass filter at 330Hz to avoid aliasing noise. For best performance, you should keep this in the tuning application.

5 Assembly Tips and Calling Conventions

1. To load 32-bit constants (such as memory addresses) into a register, you can use the `ldr` pseudo instruction. See the ARM reference for details on how to use this: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/Babbfdih.html`

2. For help understanding what `.global`, `.section`, etc mean in ARM assembly files, see the `gnu_arm_ref.pdf` in the `docs` directory.

3. For help with ARM assembly, see the `arm_isa_ref.pdf` in the `docs` directory.

4. ARM calling conventions are defined by ACPS (ARM Procedure Call Standard). This standard is defined here: `https://en.wikipedia.org/wiki/Calling_convention#ARM_.28A32.29`. If you have any questions, take a look at the ARM assembly generated for your kernel in `kernel.asm`. This will help you see how arguments/return values are passed around by the compiler and which registers are callee or caller saved. If you are still confused, ask a TA.

5. Variables defined in the linker script (`kernel/349util/kernel.ld`) such as `__irq_stack_top`, are defined at compile time by the linker. You can think of these as C global variables from the compiler telling you where some sections of a given code binary begin or end. So we can define a variable in C like the following: `extern uint32_t __irq_stack_top;` and have it be automatically filled with the address from the linker script by the compiler as it makes our ELF file. You don't need to understand how the linker script does this. However, you will need to understand how to use these labels to complete this lab.

6 Submission

1. To submit the checkpoint, use the tag `lab2-checkpoint1`.
2. To submit the final, use the tag `lab2-submit`.