Interrupts and Monitor

ECE598: Advanced Operating Systems – Homework 4

Spring 2018

Due: Thursday, 22 February 2018, 2:00pm

This homework involves getting a periodic interrupt running and writing a small command-line interpreter.

You may work in a group of two for this assignment, although it should be possible to do it on your own.

1. **Download the homework code template**
   - Download the code from:
     `http://web.eece.maine.edu/~vweaver/classes/ece598/ece598_hw4_code.tar.gz`
   - Uncompress the code. On Linux or Mac you can just
     `tar -xzvf ece598_hw4_code.tar.gz`
   - The code I provide is a starting point that contains solutions to the previous homework. If you prefer to use your own code from HW#3 as a basis, that is fine.
   - The following new code has been added (compared to HW#3):
     - `boot.s` – modified to set up IRQ vector
     - `gpio.c` and `gpio.h` – GPIO convenience functions
     - `hardware.h` – defines for hardware/cpu detection
     - `led.c` and `led.h` – code for driving the LED
     - `serial.c` – modified so you don't need '\r'
     - `shell.c` and `shell.h` – interpreter shell
     - `string.c` and `string.h` – string manipulation
     - `timer.c` and `timer.h` – timer code

2. **Set up an interrupt handler and the timer interrupt (4pt)**
   - Don't forget to comment your code!
   - First set up a periodic timer (Chapter 14 of the peripherals document has the full details).
     In `timer.c` we set up the timer. We enable a 32-bit timer that interrupts when the value we load in `TIMER_LOAD` counts down to zero (it auto-reloads after each interrupt). Pick a value to write to `TIMER_LOAD` that will give a 1Hz interrupt frequency. The system base clock is 250MHz, we divide that by 250, then again by 256. Choose an appropriate `TIMER_LOAD` value that will give a count close to 1Hz.
   - When the timer interrupt triggers, it will call the interrupt vector we setup in `boot.s`.
     This is the `interrupt_handler()` function in `interrupts.c`.
     First check that it was the timer interrupt that triggers (you can printk a warning if it was not).
     Next acknowledge (clear) the interrupt flag.
     Finally, modify this routine to alternately turn on and turn off the GPIO18 LED each time this interrupt vector is called. You can use the provided `led_on()` and `led_off()` functions.
   - The next step is to enable the ARM interrupt circuitry (as described in Section 7). To do this, in `timer.c` uncomment the `IRQ_BASIC_ENABLE` line.

- The final step is to enable global interrupts. Uncomment the `enable_interrupts();` line in `kernel_main.c`. You might want to look at the relevant code in `interrupts.h` just as a reminder of what that code is doing.
- Compile, write this code to your memory card, and boot your kernel. If all went well the LED should be blinking!

3. **Set up a simple command line interpreter (2pt)**

   - Make a simple operating system "monitor" or "shell" thats reads keypresses into a buffer and then executes the commands when enter is pressed.
   - Put the code into `shell()` in the `shell.c` file.
   - Have an infinite loop as before, doing a `ch=uart_getc()`
   - Have a character buffer (such as `char buffer[4096];`) where each character is put. Have an index variable keeping track of where to store each additional character you read. After you read a character, still do a `uart_putc()` to echo it to the screen.
   - Once Enter (`'\r'`) is pressed then put a NUL terminating char at the current offset, then call your parsing routine on the buffer.
   - Writing a full command line parser is tricky, especially without any string library available. For this assignment, check to see if the command `print` is typed and if so do a `printk()` of `"Hello World"` to the screen. If anything else is typed, printk() `"Unknown Command"`
   - You can cheat a bit with your parser and do something as simple as: `if ((buffer[0]=='p') && (buffer[1]=='r')) {` to detect the command. The provided string library (`string.c`) also supports `strncmp()`
   - On return be sure to reset your offset pointer back to 0.

4. **Something Cool (1pt)**

   - Add another command of your choice that is handled by your parser. It can do anything; some suggestions are to print your name, print your OS version number, clear the screen, etc. Be sure to document the command and what it does in the answers document.

5. **Answer the following questions (3pt)**

   Put your answers to these questions in the README file.

   (a) What is the difference between an ARM IRQ interrupt and a FIQ interrupt?
   When might this difference be useful?

   (b) You receive an interrupt and check the `BASIC_PENDING` register to see what it was. Bit 19 has been set to one. What was the cause of the interrupt? (Hint: Read Chapter 7 of the BCM2835 Peripherals document)

   (c) The ARM processor boots up in SVC mode. How can you manually switch to IRQ mode?

   (d) If you look at the `kernel7.dis` disassembly of your operating system you see that at the end of the interrupt handler it uses the instruction: `subs pc, lr, #4`
   to return from the interrupt. Why does it subtract 4 from the link register before returning?

6. **Submit your work**

   - Run `make submit` in your code directory and it should make a file called `hw4_submit.tar.gz`. E-mail that file to me.