ECE531: Advanced Operating Systems – Homework 4
Interrupts and Monitor

Due: Friday, 29 September 2023, 5:00pm

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

1. **Download the homework code template**
   - Download the code from:
     `https://web.eece.maine.edu/~vweaver/classes/ece531/ece531_hw4_code.tar.gz`
   - Uncompress the code. On Linux or Mac you can just
     `tar -xzvf ece531_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 table
     - `console_write.c` – wraps the uart write routines for printk() use
     - `device_tree.c` – used to detect pi model
     - `gic-400.c` – extra interrupt controller needed by the pi4
     - `gpio.c` – GPIO convenience functions
     - `hardware_detect.c` – used to detect Pi model
     - `interrupt.c` – minimal interrupt handler
     - `led.c` – code for driving the LED
     - `printk.c` added support for printing strings with %s
     - `serial.c` – modified so you don't need '\r'
     - `shell.c` – interpreter shell
     - `string.c` and `memcpy.c` – string and memory routines, used by device_tree code
     - `timer.c` – 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 (See Chapter 14 of the BCM2835 peripherals document or Chapter 12 in the BCM2711 document)
     - 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.
     - NOTE: it's unclear if the system clock is actually 250MHz on a Pi4. If your timer doesn't quite tick at 1Hz even though you think you calculated right, don't worry about it.

- Be sure to use `bcm2835_read()` and `bcm2835_write()` to access the MMIO registers. These internally adjust for the io_base register differences between Pi3 and Pi4. There are some non-adjusting mmio functions but those are only used for accessing the gic-400 interrupt controller (GICD/GICC registers) as this is not located in the standard MMIO region.

- 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`, so edit that file.
  - Ideally we'd first check that the interrupt that happened actually was a timer interrupt, and print an error message otherwise. For this homework we're going to not do that and just assume it was a timer interrupt.
  - Next acknowledge (clear) the TIMER 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 SoC interrupt circuitry. It turns out this can be complex, especially on a Pi4, so the code was provided. You will need to activate it though by removing the `#if 0` block around the code at the end of `timer.c`.
- 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. (Be sure to overwrite kernel7.img on a Pi2/3, kernel7l.img on a Pi4) 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()`
- When you return from handling the input line, be sure to reset your offset pointer in the buffer to 0 and then keep looping forever.

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 are running on a Pi3 and 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) (Also, the manual is a bit misleading. The interrupts specified match those in the ARM table)

    (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.