ECE531: Advanced Operating Systems – Homework 4

Timer Interrupts and System Monitor

Due: Monday, 6 October 2025, 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 that `printk()` uses
     - `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 a 1Hz timer interrupt (4pt)**

   - Don't forget to comment your code!
   - First set up a periodic timer (See Chapter 14 of the BCM2835 peripherals 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.
     - Be sure to use `bcm2835_read()` and `bcm2835_write()` to access the MMIO registers. These internally adjust for the io_base register differences between Pi models.
   - When the timer interrupt triggers, it will call the `interrupt_handler()` function in `interrupts.c`, so edit that file.

  - There's code that checks to make sure it was a timer interrupt by checking the `IRQ_BASIC_PENDING` register. Add your code inside that block of code.
  - Add code to Acknowledge (clear) the TIMER interrupt flag as described in class.
  - Add code to alternately turn on and turn off the ACT LED each time this interrupt vector is called. You can use the provided `act_led_on()` and `act_led_off()` functions.
- The next step is to enable the ARM SoC interrupt circuitry. The code is provided at the end of the init function in `timer.c`, just uncomment it.
- 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, and if all goes well you will have a `kernel.img` that you can write to the SD card. Power it on and if all went well the LED should be blinking at 1Hz!

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.
- The provided code has two nested infinite loops. The outer one handles full lines (so only runs after ENTER is pressed)
- The inner one reads a character from the serial port using `ch=uart_getc()` and then prints it with `uart_putc()`
- Add code that takes these keypresses and puts them in the `buffer[]` string. You will need to add a variable to store the current offset into the string and then gets incremented with each character. After you read a character, still do a `uart_putc()` to echo it to the screen.
- Once Enter (`'\n'`) is pressed then put a NUL (0) terminating byte at the current offset, then call your parsing routine on the buffer.
- Writing a full command line parser is tricky. A `strncmp()` routine is provided in `string.c` that can you use.
- 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"`
- 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 by updating the "help" command in the parser.

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) Your kernel receives an interrupt and your code checks 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 `kernel.dis` disassembly of your operating system, specifically the code for the interrupt handler you will see that gcc adjusts `LR` (link-register / return address) by subtracting 4 off at the beginning before saving it on the stack. Why does it do that?

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.