Interrupts and Monitor

ECE598: Advanced Operating Systems – Homework 3

Spring 2015

Due: Wednesday, 25 February 2015, 5pm

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

1. **Download the homework code template**

   - Download the code from:
     `http://web.eece.maine.edu/~vweaver/classes/ece598_2015s/ece598_hw3_code.tar.gz`
   - Uncompress the code. On Linux or Mac you can just
     `tar -xzvf ece598_hw3_code.tar.gz`
   - If you wish to use your HW2 code as a basis, you can. In that case you will need to cut and paste your code over into the new tree. If I had had more time I would have organized the code better to make this easier to accomplish, I apologize for the inconvenience.

2. **Set up an interrupt handler and the timer interrupt (2pt)**

   - 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 2Hz interrupt frequency. The system base clock is 1MHz, and we are dividing that by 256 with the prescalar. Modify the mmio_write() to have the proper value.
   - When the timer interrupt triggers, it will call the interrupt vector we setup in `boot.s`.
     This is the `interrupt_vector()` function in `interrupts.c`.
     Modify this routine to alternately turn on and turn off the LED each time this interrupt vector is called. You can use the provided `led_on()` and `led_off()` functions.
   - 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" that reads keypresses in a buffer and then executes the commands when enter is pressed.
   - 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.
- On return be sure to reset your offset pointer back to 0.

4. **Add your own system call (2pt)**

   - Add a system call called `SYSCALL_BLINK` that takes one argument. An argument of 0 disables the blinking of the LED, an argument of 1 re-enables it.
   - You will want to add a case for `SYSCALL_BLINK` in the `syscalls.c` file. See the `SYSCALL_WRITE` code as an example.
   - The first argument from the syscall will be in r0. Use this to appropriately set the global variable `blinking_enabled` (defined in `interrupts.c`) to 0 or 1.
   - Update your interrupt vector code in `interrupts.c` to disable/enable the blinking based on the status of the `blinking_enabled` variable.
   - Add code to your command line interpreter so that if you type `blink on` it calls your new syscall to enable blinking, and if you type `blink off` it disables it.
     You can use the `syscall1()` helper routine (that's syscall followed by the number 1) to generate the proper system call instruction; this is inline assembly defined in `syscalls.h`. Just call `syscall1()` with the first argument being 0 or 1 and the second argument being the proper syscall number `i.e. SYSCALL_BLINK`.

5. **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.

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

   Put your answers to these questions in some sort of document (.txt, .pdf, .doc).

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

   (b) How does one change from kernel mode to user mode on an ARM processor?
   Once in user mode, how can execution re-enter kernel mode?

   (c) In the following C code, what section of memory would the variable be allocated (data, bss, stack, heap, etc.). Assume the code specified is inside the `main()` function.

   i. `int a=0;`
   ii. `static int b=1;`
   iii. `static int c=0;`

iv. `int *d=malloc(sizeof(int));`

(d) What is the difference between first-fit and best-fit memory allocation? Why might you use one over the other?

7. **Submit your work**

- Run `make submit` in your code directory and it should make a file called `hw3_submit.tar.gz`. E-mail that file to me as well as the document with the answers to the questions.