Project 3: Simple Shell & System Calls (4%)

ENEE 447: Operating Systems — Spring 2019

Assigned: Tuesday, Feb 26; **Due:** Sunday, Mar 10

Purpose

This project has you implement vectored interrupts on the Raspberry Pi. Vectored interrupts are one of the most important facilities that hardware offers, because they allow a wide range of asynchronous operation to occur — whenever an interrupt occurs, the PC is redirected to a completely different location, which allows the operating system to split its attention between multiple things.

Vectored Interrupts in ARM

The ARM implementation puts the vector table at the very start of memory, and it must contain a set of jump *instructions* as opposed to jump *addresses*. A typical layout might look like the following:

```
.globl _start
_start:
        // jump table:
        b res_handler                 // RESET handler              - runs in SVC mode
        b und_handler                 // UNDEFINED INSTR handler    - runs in UND mode
        b swi_handler                 // SWI (TRAP) handler         - runs in SVC mode
        b pre_handler                 // PREFETCH ABORT handler     - runs in ABT mode
        b dat_handler                 // DATA ABORT handler         - runs in ABT mode
        b hyp_handler                 // HYP MODE handler           - runs in HYP mode
        b irq_handler                 // IRQ INTERRUPT handler      - runs in IRQ mode
        1st instr of FIQ handler      // FIQ INTERRUPT handler      - runs in FIQ mode
        … (FIQ handler can simple be written in-line)
```

So, whenever the system takes a RESET interrupt, the number 0x00000000 is loaded into the program counter, which causes the processor to jump to address zero. At address zero is an instruction

```
b res_handler
```

that tells the hardware to branch to the location of `res_handler`.

Similarly, whenever the system takes a SWI interrupt (which is caused by the `svc` assembly-code instruction), the number 0x00000008 is loaded into the program counter, which causes the processor to jump to address 0x08 (the third word in memory). At address 0x08 is an instruction

```
b swi_handler
```

that tells the hardware to branch to the location of `swi_handler`.

And so forth. Note that the 'b' instruction cannot jump arbitrarily far, so your code can't be spread far and wide … but for the purposes of our projects, this should not be an issue.

Ridiculously Simple Shell

Right now, you have worked with several functions that the operating system can provide to users, including getting the time from a running clock, pointing debug/logging info to the console, blinking the LED, etc. The following shows an extremely simple user interface that can access some of these features:

```
[c0|00:02.021] ...
[c0|00:02.023] System is booting, cpuid = 00000000
[c0|00:02.027] Kernel version: [p3-solution, Mon Mar 4 16:40:18 EST 2019]
[c0|00:02.034] Available devices:
[c0|00:02.037] Null
[c0|00:02.039] Device number:  00000000
[c0|00:02.043] Device type:    00000000
[c0|00:02.046] Init function:  000000DC
[c0|00:02.050] Read function:  000000DC
[c0|00:02.053] Write function: 000000DC
```

```
[c0|00:02.057] LED
[c0|00:02.059] Device number:  00000001
[c0|00:02.062] Device type:    00000001
[c0|00:02.066] Init function:  00000670
[c0|00:02.069] Read function:  00000474
[c0|00:02.073] Write function: 000004A4
[c0|00:02.077] Console
[c0|00:02.079] Device number:  00000002
[c0|00:02.082] Device type:    00000001
[c0|00:02.086] Init function:  00001694
[c0|00:02.090] Read function:  000004D0
[c0|00:02.093] Write function: 000004E0
[c0|00:02.097] Clock
[c0|00:02.099] Device number:  00000003
[c0|00:02.102] Device type:    00000002
[c0|00:02.106] Init function:  0000123C
[c0|00:02.109] Read function:  000004F4
[c0|00:02.113] Write function: 00000474
[c0|00:02.117] KernLog
[c0|00:02.119] Device number:  00000004
[c0|00:02.122] Device type:    00000002
[c0|00:02.126] Init function:  00000900
[c0|00:02.130] Read function:  00000474
[c0|00:02.133] Write function: 00000458
[c0|00:02.137] ...
[c0|00:02.139] Please hit any key to continue.
```

<the ENTER key is pressed>

```
Running shell.
Available commands:
 LED  = 0044454C
 LOG  = 00474F4C
 TIME = 454D4954
 EXIT = 54495845

Please enter a command.
> LED
CMD_LED - on

Please enter a command.
> LED
CMD_LED - off

Please enter a command.
> LOG "THIS IS A TEST MESSAGE"
[c0|00:30.303] THIS IS A TEST MESSAGE

Please enter a command.
> TIME
CMD_TIME = [00000000 02441F42]

Please enter a command.
> EXIT
CMD_EXIT exiting ...
```

The system boots up in kernel mode and runs the shell in kernel mode, That means that all facilities are available to the shell, for now. We have given you the base code that parses the input, and it is up to you to arrange it into a system-call-like framework so that the correct functions are called.

The Simplest Possible Operating System

At its simplest, an OS is nothing more than a collection of vectors: it does nothing unless it is responding to interrupts. This is what we will be building in Projects 3 and 4. Project 3 implements a system-call facility and provides an extremely rudimentary shell that runs in user mode — but still within the kernel proper, for now. As shown above, the shell understands the following commands:

- **led** — toggles the LED on and off.
- **time** — gets the 64-bit time value from the kernel into a 64-bit "long long" integer (a uint64_t).
- **log** "string" — sends the string (which may contain spaces) to the kernel, which prints it out as a console log message

- **exit** — exits the shell.

The **led**, **time**, and **log** functions all call the kernel via system calls. In addition, the character I/O that the shell uses to receive the keyboard input and print to the screen is all handled via the system-call facility as well. You will find that the shell, because it runs in user mode, does not have direct access to the UART or other I/O facilities that the kernel controls … that is why system calls are used in real systems.

Generalized/Virtualized I/O Devices

You will see that the I/O subsystem makes significant use of indirection. There are two forms of I/O supported in the kernel:

- **word-granularity**. These I/O operations require only a single atomic data structure: a 32-bit integer or smaller. Thus, this can perform character I/O and data transfers of words, but not larger-granularity data items such as strings or blocks of data. The devices that use word-granularity I/O include the following:
  - **LED**. The LED is write-only and takes a zero/nonzero value in its word to indicate that the LED should be turned off/on, respectively. A read of the LED simply returns a '1' indicating non-error.
  - **Console**. The Console is the system terminal (keyboard and monito) and is read/write at a character granularity. This is the UART device.

  The two I/O system calls that handle word-granularity I/O are *read* and *write* and are defined as follows:

```
int syscall_read_word(int device_number);

int syscall_write_word(int device_number, long data_value);
```

  These load the registers as needed and call an SVC assembly-code instruction, which interrupts the operating system.
- **stream-based**. These I/O operations transfer data in larger granularities than what can fit in a single register. Thus, this is how one handles data items such as strings or blocks of data. The devices that use stream-based I/O include the following:
  - **Clock**. The Clock is read-only. The Clock device returns a 64-bit integer value, which causes problems because the 32-bit ARM cores can only handle 32 bits at a time. Even though the compiler can create "long long" data structures, this cannot be transferred through a single register and thus may not work as an atomic return value from a simple in-kernel I/O routine transferring interrupt-return values through the register file. Thus, we will implement the Clock device by having the kernel copy 8 bytes from kernel space to user space.
  - **Kernel Log**. The Kernel Log device is write-only at a string granularity. We will assume that strings are the size of a *character buffer* (*CBUFSIZE* bytes) or smaller (this will come into play in the next project).

  The two I/O system calls that handle stream-based I/O are *read* and *write* and are defined as follows:

```
int syscall_read_stream(int device_number, void *buffer, int bufsize);

int syscall_write_stream(int device_number, void *buffer, int bytes);
```


These load the registers as needed and call an SVC assembly-code instruction, which interrupts the operating system. The *buffer* pointers must point to statically allocated space within the calling entity. The *read* function's *bufsize* argument is the size of the buffer (a maximum number of bytes to return, including a '\0' character at the end of a string); the *write* function's *bytes* argument is the maximum number of bytes to write to the device.

You will see within the io.c module the following data structure:

```
//
// struct dev {
//  char name[8];
//  int type;
//  pfv_t init();
//  pfi_t read();
//  pfi_t write();
// }
//

struct dev devtab[MAX_DEVICES+1] = {
    {
        "Null",
        DEV_INVALID,
        dummy,
        (pfi_t)dummy,
        (pfi_t)dummy,
    },

    {
        "LED",
        DEV_WORD,
        init_led,
        io_read_led,
        io_write_led,
    },

    {
        "Console",
        DEV_WORD,
        init_uart,
        io_uart_recv,
        io_uart_send,
    },

    {
        "Clock",
        DEV_STREAM,
        init_time,
        io_get_time,
        io_error,
    },

    {
        "KernLog",
        DEV_STREAM,
        init_log,
        io_error,
        io_klog,
    },

    {
        "NONE",
        DEV_INVALID,
        (pfv_t)io_error,
        io_error,
        io_error,
    },

};
```

This is the collection of *read/write* and *init* functions that are set up for each specific device. The *init_io()* function calls each of the *init()* functions in sequence, at which point all devices are initialized and ready for I/O. In the *trap_handler*, user requests to read and write these devices are vectored to the appropriate routing by indexing into this table. For instance, to read a character from the keyboard, one calls a *read()* function on the Console device: the system call is a SYSCALL_RD_WORD type, and the Console has

device number 2. The *trap_handler* function thus indexes into this table to the 2nd entry and calls its *read()* function. Because it is a SYSCALL_RD_WORD system call, the trap handler invokes it simply as

```
<device>.read();
```

because, as described above, beyond the device number (which is used by the *trap_handler* to find the device-specific read/write routines and thus is not handed to the device-specific read/write routines), the word-granularity *read()* function takes no argument and simply returns a single word-sized value.

To write a value to the monitor, one calls a *write()* function on the Console device: the system call is a SYSCALL_WR_WORD type, and the Console has device number 2. The *trap_handler* function thus indexes into this table to the 2nd entry and calls its *write()* function. Because it is a SYSCALL_WR_WORD system call, the trap handler invokes it as

```
<device>.write(data_value);
```

because, as described above, beyond the device number (which is not handed to the device-specific read/write routines), the word-granularity *write()* function takes a single argument that is a word-sized value to be written to the device.

To write a string to the kernel's log, one calls a *write()* function on the KernLog device: the system call is a SYSCALL_WR_STREAM type, and the KernLog has device number 4. The *trap_handler* function thus indexes into this table to the 4th entry and calls its *write()* function. Because it is a SYSCALL_WR_STREAM system call, the trap handler invokes it as

```
<device>.write(data_ptr, len);
```

because, as described above, beyond the device number (which is not handed to the device-specific read/write routines), the stream *write()* function takes a data pointer and a length. Because the I/O routine for the KernLog is specific to that device, it knows that it will receive a character pointer to a string, so it can be written to expect a **char** * as an argument.

To get the system time, one calls a *read()* function on the Clock device: the system call is a SYSCALL_RD_STREAM type, and the Clock has device number 3. The *trap_handler* function thus indexes into this table to the 3rd entry and calls its *read()* function. Because it is a SYSCALL_RD_STREAM system call, the trap handler invokes it as

```
<device>.read(data_ptr, len);
```

because, as described above, beyond the device number (which is not handed to the device-specific read/write routines), the stream *read()* function takes a data pointer and a buffer size. Because the I/O routine for the Clock is specific to that device, it knows that it will receive a pointer to a uint64_t (an ***unsigned long long*** data type), so it can be written to expect a **unit64_t** * as an argument.

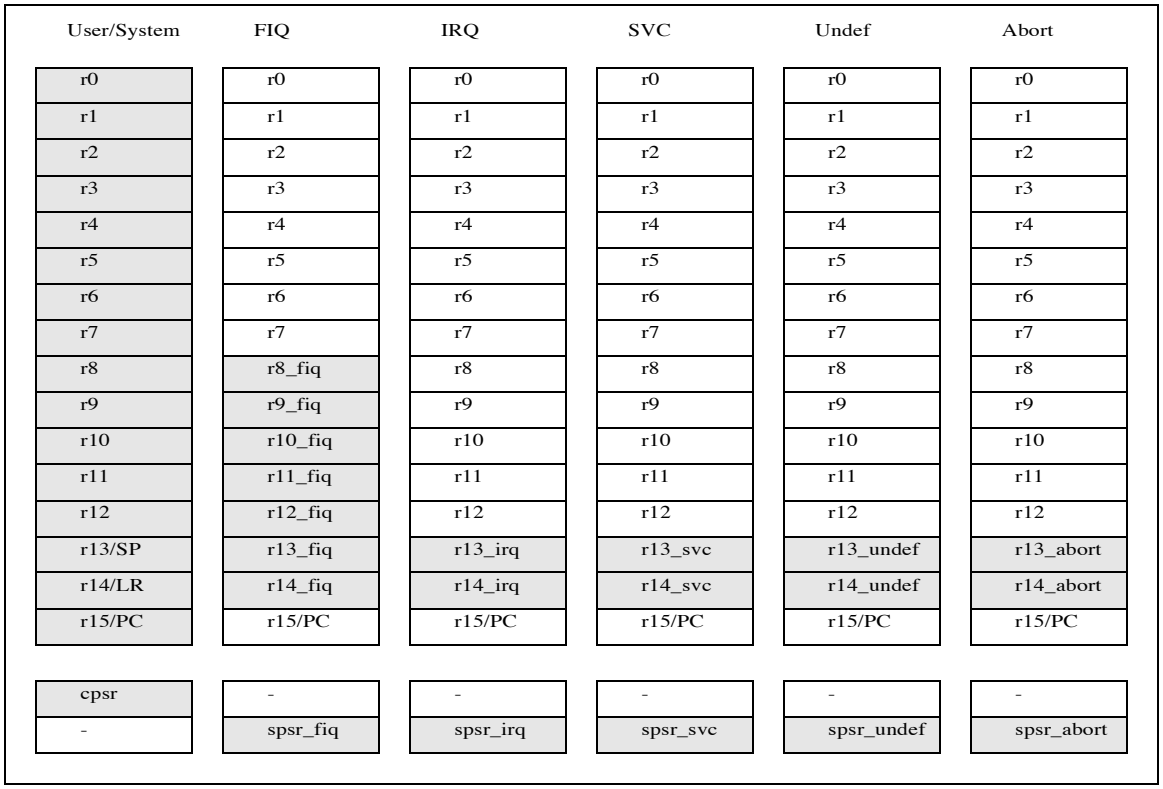
And so forth. The reason operating systems design things this way is to make them easily extendable to handle numerous different device types.

Implement System Calls via Vectored Interrupts

Your task is to implement the ARM vector table (see the example jump table shown above) and the SVC interrupt handler. We will give you code that drives most of this; your job is just to set up the jump table how you want it, finish the handler, write the system calls that actually call the kernel from user mode, and finish the I/O handling from within the kernel at the other end of the trap (what goes on in *trap_handler* and the **io.c** module). The boot code you are given starts up the processor and ultimately runs the shell process (*run_shell*) in user mode.

A Note on Modes and Stacks

Note that each of the vectors runs in a different **mode** (except for two that both run in ABT mode). Recall the register-file arrangement in the ARM architecture:



User/System	FIQ	IRQ	SVC	Undef	Abort
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13/SP	r13_fiq	r13_irq	r13_svc	r13_undef	r13_abort
r14/LR	r14_fiq	r14_irq	r14_svc	r14_undef	r14_abort
r15/PC	r15/PC	r15/PC	r15/PC	r15/PC	r15/PC
cpsr	-	-	-	-	-
-	spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abort

When a mode is invoked, its corresponding register set becomes visible. So, for instance, each mode has its own banked stack pointer and link register — the **sp/lr** registers, r13 and r14. In addition, the FIQ mode also has its own private r8–r12 registers. Why this is interesting is that you can set up a separate stack for each mode that becomes visible when that mode is invoked. The code is as follows:

```
.equ    USR_mode,   0x10
.equ    FIQ_mode,   0x11
.equ    IRQ_mode,   0x12
.equ    SVC_mode,   0x13
.equ    HYP_mode,   0x1A
.equ    SYS_mode,   0x1F
.equ    No_Int,     0xC0

    cps     #IRQ_mode
    mov     sp, # IRQSTACK0

    cps     #FIQ_mode
    mov     sp, # FIQSTACK0

    cps     #SVC_mode
    mov     sp, # SVCSTACK0

    cps     #SYS_mode
    mov     sp, # KSTACK0
```

This has been set up for you in the file **1_boot.s**.

A Note on "USR" Mode vs. "SVC" Mode

This project is structured to emulate the behavior of user code invoking the operating system, but as you will probably realize when you start writing, compiling, and running the code, the "user" code is in the

same binary as the "kernel" code! How could that be considered "user" code? Well, at this point, we have not developed the concept of multiple tasks, and we have not developed the concept of virtual memory, and so having a separate binary is a little beyond what we can do (for the moment).

However, we do not *need* any of those facilities to accomplish the goal of setting up a system-call interface. The multiple privilege levels allow us to emulate the behavior all in one binary. You will notice in the 1_boot.s code the kernel is invoked, to initialize everything, and then the process is set to USR mode and "user" code is called:

```
cps             #SYS_mode
mov             sp, # KSTACK0
bl              init_kernel

// set up user stack and jump to shell
cps             #USR_mode
mov             sp, # USTACK0
bl              run_shell
```

The shell is invoked, and it runs in USR mode, which does *not* have access to the various facilities (including the various devices) that are available in SYS mode or SVC mode. Effectively, in this project, you are running two separate things

1. the kernel code, which runs in privileged mode and has direct access to the various devices
2. the "user" code which runs in user mode (non-privileged mode) and does NOT have access to the various devices

Even though the "user" code is in the same binary as the kernel (for now), you should still think of it as a "user" program. It does not have access to any of the kernel facilities, in particular.

That means that, within the shell, *log()* does not work, *led_on()/led_off()* don't work, and none of the other routines work, either [e.g., *uart_put32x()/uart_puts(),* etc.] While they are not available *directly*, they *are* available *indirectly*. You have to get to those functions through system calls. That is kind of the point of the project. The only things you should be using in user space are functions in the u_* and z_* files and the system calls provided to you.

Build It, Load It, Run It

Once you have it working, show us.

