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:

```assembly
.globl_start
_start:
    // jump table:
b res_handler     // RESET handler      - runs in SVC 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.

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. 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.
The *exit* function 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 monitor) 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:

```c
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:

```c
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:

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

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

```c
<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

```c
<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

```c
<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

```c
<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 `uint64_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:

<table>
<thead>
<tr>
<th>User/System</th>
<th>FIQ</th>
<th>IRQ</th>
<th>SVC</th>
<th>Undef</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>r8</td>
<td>r8</td>
<td>r8</td>
<td>r8</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>r9</td>
<td>r9</td>
<td>r9</td>
<td>r9</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>r10</td>
<td>r10</td>
<td>r10</td>
<td>r10</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>r11</td>
<td>r11</td>
<td>r11</td>
<td>r11</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>r12</td>
<td>r12</td>
<td>r12</td>
<td>r12</td>
<td>r12</td>
<td>r12</td>
</tr>
<tr>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
</tr>
<tr>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
</tr>
<tr>
<td>r15_fiq</td>
<td>r15_fiq</td>
<td>r15_fiq</td>
<td>r15_fiq</td>
<td>r15_fiq</td>
<td>r15_fiq</td>
</tr>
<tr>
<td>spsr_fiq</td>
<td>spsr_fiq</td>
<td>spsr_fiq</td>
<td>spsr_svc</td>
<td>spsr_svc</td>
<td>spsr_svc</td>
</tr>
</tbody>
</table>

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 HYF_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`.

Build It, Load It, Run It

Once you have it working, show us.