Project 3: Interrupts, Syscalls, Devices (4%)

ENEE 447: Operating Systems — Spring 2021
Assigned: Tuesday, Feb 23; **Due:** Sunday, Mar 7

Purpose

This project has you implement system calls through 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.

System calls are essentially *functions* — but they are functions that an application cannot perform on its own, which is why the application needs to ask the operating system to do them. In practice, system calls are the software's invocation of the *interrupt* facility, and this arrangement is how the user code gets the attention of the operating system, typically to perform I/O operations on its behalf, because a user application typically cannot perform I/O (talk to disks, keyboards, monitors, networks, etc.) on its own.

Lastly, an important function that the operating system performs is that of *abstraction:* the operating system presents an abstract version of devices to the user application, so that the user application can perform a set of common routines on devices, instead to having to learn the intricate device-operation details and implement them via an interactive routine.

You have been given a kernel and an "application" (called "app" and located in its own directory). After the kernel initializes, it then starts up the app, which actually runs within the kernel, so it is not really a "user application" … we will get to that in the next few projects. The app's only purpose is to continually invoke different system calls, each with a different system call number and, for each system call number, a different device number. The kernel handles the system call, invoked through a "svc" assembly-code instruction, with certain information left for the kernel in pre-defined registers. Your job is to extend this facility to create ***open(), read(), write(),*** and ***close()*** system calls, for an array of devices.

Vectored Interrupts in ARM

First off is how system calls are delivered to the operating system: through vectored interrupts. The ARM implementation of vectored interrupts puts the vector table at the very start of memory, and it must contain a set of jump *instructions* as opposed to jump *addresses*. Each of these represents a different ***vector*** into the OS. A typical layout of the ARM interrupt-vector table 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: a *trap/syscall* 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
```

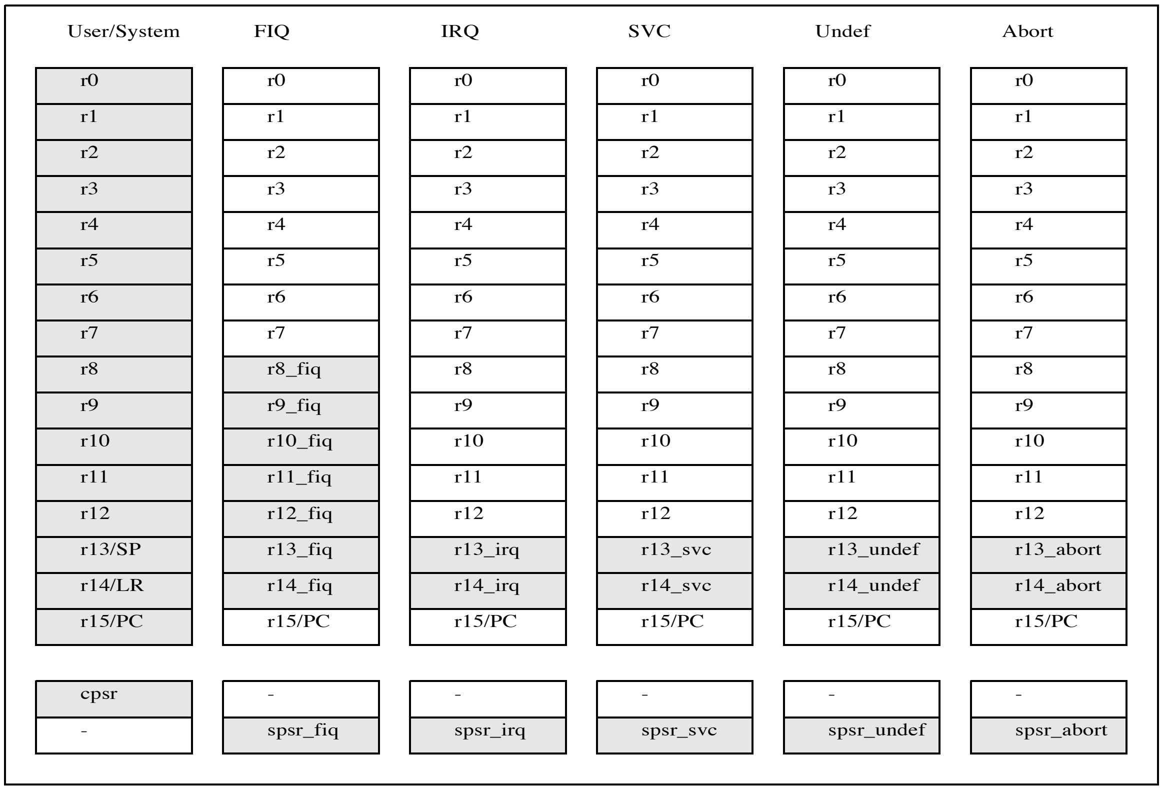
that tells the hardware to branch to the location of `swi_handler`.

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.

Modes and Stacks

System calls are not functions: they do not pass arguments on a stack. They do, however, need to pass information into and out of the kernel, so some form of communication must be defined and agreed upon. In most operating systems, the way that system calls typically transmit information from user space to kernel space is through the **register file**. When a user application invokes a particular system call, it puts that system call’s unique number into a known register, it. puts a device number into another known register, and it puts any other associated arguments into additional registers. Then it executes the **svc** assembly-code instruction, which causes an interrupt. The kernel wakes up inside the **SVC** handler, and the values are in the registers, where the user code just left them. That means that the user code and the kernel code share some registers. How this is arranged is different in each hardware architecture, so we will discuss here how it is implemented in the ARM architecture.

In the ARM architecture, each of the **interrupt vectors** in the table above 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**.

Therefore, when your trap handler runs, it will SVC mode; it will have its own banked registers (the **r13_svc** and **r14_svc** registers), and it will have its own stack as set up by **1_boot.s**, via the assembly-code snippet above.

Traps and System Calls, Generally

System calls are generally implemented as a bunch of library routines: for instance, in the Unix operating system, the most commonly used system calls are ***open(), close(), read(),*** and ***write().*** These are implemented as **library routines** that place the special values into the pre-defined registers and then execute the **svc** assembly-code instruction. They effectively serve as vectors into the operating system and, as defined in Unix, operate on all possible devices (disks, keyboards, monitors, networks, etc.). In other words, you can **read**() the disk, **read**() the keyboard and network, **write**() to the disk, the monitor, and the network, etc. You do not need a separate system call for everything you want to do; you simply need a different combination of a fundamental system call (**open**(), **close**(), **read**(), and **write**()) and device.

The library routines that implement system calls look like the following in their implementation:

```
int
generic_syscall(int device, long data1, long data2, long data3)
{
    register long r7 asm("r7") = SYSCALL_NUMBER;            // #define'd in syscall.h
    register long r0 asm("r0") = device;
    register long r1 asm("r1") = data1;
    register long r2 asm("r2") = data2;
    register long r3 asm("r3") = data3;

    asm volatile (
        "svc #0\n"
        : "=r"(r0)                                          // output: return status
        : "r"(r0), "r"(r1), "r"(r2), "r"(r3), "r"(r7)       // input: syscall & arg/s
        : "memory");

    return r0;
}
```

Each system call is uniquely defined by an integer that is specified in a system library file (e.g., in our operating system the set of numbers is found in **syscall.h**). A system call's unique number is placed in a known register (in our implementation it is **r**7). The primary argument to the system call is a device number, which we will place in **r0**, because that corresponds to the first argument in a function's argument list. Then there are optional additional arguments, which have been shown above as *data1*, *data2*, and *data3*.

The **r0..r3** format is chosen to resemble the calling functions of the ARM function-call interface (the first four arguments to a function are all passed through registers **r0..r3** in the register file). Thus, this arrangement will correspond to the following form for an interrupt handler within the OS kernel:

```
int
handler(unsigned long device, unsigned long arg1, unsigned long arg2, unsigned long arg3)
{
        // handler body
}
```

The registers **r0**, **r1**, **r2**, and **r3** correspond to the argument list within a function, as we have seen in the discussion of stacks on the ARM. That is why the arguments to the system call are communicated through registers **r0 .. r3**. The system call number is transmitted through **r7**, and so the main ***svc_handler*** routine uses this value in **r7** in a jump table or switch statement to invoke the handler that corresponds to the desired system call. This produces a system-call format of up to three arguments beyond the device number (system calls will not in general have the same number of arguments: for. example, a simple ***write()*** system call will have one more argument than a ***read()*** system call, if it transmits the data to be written as part of the argument list).

In your project write-up you are given, in **trap_handlers.c**, the initial trap handler code that responds to the SVC interrupt. What you are given simply prints out the contents of the various registers when the interrupt is received. It then returns to the "user" code that called it. The output looks like the following:

```
00:04.494 <pc=00003C30> - Trap Handler:
00:04.497 <pc=00003C40> -    r7 00000000
00:04.501 <pc=00003C50> -    r0 00000000
00:04.505 <pc=00003C60> -    r1 F00D0000
00:04.508 <pc=00003C70> -    r2 C0FFEE00
00:04.512 <pc=00003C80> -    r3 00020FB4
00:04.516 <pc=00003C90> -    SP 00023FD8
00:04.519 <pc=00003CA0> -   EPC 000000D4
00:04.523 <pc=00003CB0> -  SPSR 200001DF

00:05.527 <pc=00003C30> - Trap Handler:
00:05.530 <pc=00003C40> -    r7 00000000
00:05.533 <pc=00003C50> -    r0 00000001
00:05.537 <pc=00003C60> -    r1 F00D0001
00:05.541 <pc=00003C70> -    r2 C0FFEE01
00:05.544 <pc=00003C80> -    r3 00020FB4
00:05.548 <pc=00003C90> -    SP 00023FD8
00:05.552 <pc=00003CA0> -   EPC 000000D4
00:05.555 <pc=00003CB0> -  SPSR 800001DF

00:06.559 <pc=00003C30> - Trap Handler:
00:06.562 <pc=00003C40> -    r7 00000000
00:06.566 <pc=00003C50> -    r0 00000002
00:06.569 <pc=00003C60> -    r1 F00D0002
00:06.573 <pc=00003C70> -    r2 C0FFEE02
00:06.577 <pc=00003C80> -    r3 00020FB4
00:06.580 <pc=00003C90> -    SP 00023FD8
00:06.584 <pc=00003CA0> -   EPC 000000D4
00:06.588 <pc=00003CB0> -  SPSR 800001DF

...
```

You are to create the jump table/switch statement that looks at the value in **r7** and calls the corresponding internal function that handles the desired system call.

System Devices

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, 4, and 5. Project 3 implements a system-call facility, and you are to implement the read, written, open, and close system calls to operate on the following devices:

- **LED** — writing 1/0 turns the LED on/off.
- **CLOCK** — reading the clock gets the current time of day.

- **LOG** — writing to this device puts a string into the kernel log.
- **DISK** — you can open and read files on the SD card.

You can define how this works however you want — in the next projects, we will provide formal definitions.

Reading from the SD Card

The “app” user application is a separate file that is on its own on the SD card. It is loaded into memory by the kernel after startup and initialization, and then the kernel simply jumps into the code. Thus, it is not really “user” code in that regard … but it is a separate program, and it call the svc assembly-code instruction to invoke a number of system calls, so that you can see the interaction for yourself before modifying it to do what is required for this project. A large part of the latest kernel code is the SD card interface, and this shows a little bit of that.

The following shows the boot sequence for the code as given to you. In the ***kernel.c*** module, there is a function called ***test_read()*** that provides an example on how to interface with the ***SDCard.c*** module. After initializing the SD card and opening up the ***kernel7.img*** file (your bootable file on the card), it reads into a local buffer the first 1024 bytes of the ***kernel7.img*** file, prints out the first 64 words of it, and then goes into a forever loop.

```
[c0|00:02.263] ...
[c0|00:02.265] System is booting, kernel cpuid = 00000000
[c0|00:02.270] Kernel version: [p3, Mon Feb 22 22:06:56 EST 2021]
[c0|00:02.276] Initializing SD Card ...
[c0|00:02.280] ---------------> sdInitCard [init]
[c0|00:02.284] EMMC: reset card.
[c0|00:02.287] EMMC: setting clock speed to  00061A80
[c0|00:02.292] GO_IDLE_STATE 00000000
[c0|00:02.295] SEND_IF_COND 000001AA
[c0|00:02.299] APP_CMD 00000000
[c0|00:02.302] SD_SENDOPCOND 50FF8000
[c0|00:02.706] APP_CMD 00000000
[c0|00:02.708] SD_SENDOPCOND 50FF8000
[c0|00:02.712] ALL_SEND_CID 00000000
[c0|00:02.715] SEND_REL_ADDR 00000000
[c0|00:02.719] SEND_CSD AAAA0000
[c0|00:02.722] EMMC: setting clock speed to  017D7840
[c0|00:02.726] CARD_SELECT AAAA0000
[c0|00:02.730] APP_CMD AAAA0000
[c0|00:02.733] SEND_SCR 00000000
[c0|00:02.739] SET_BLOCKLEN 00000200
sdTransferBlocks read blk 00000000 len 00000001 addr 00020D90
[c0|00:02.747] READ_SINGLE 00000000
sdTransferBlocks read blk 00002000 len 00000001 addr 00020D90
[c0|00:02.760] READ_SINGLE 00002000
[c0|00:02.773] ---------------> sdInitCard [term]
[c0|00:02.777] SD Card working.
[c0|00:02.780] ...
[c0|00:02.782] Init complete. Please hit any key to continue.
```

<hit enter>

```
[c0|00:49.260] test_read - SD Card example usage
sdTransferBlocks read blk 00003DCA len 00000001 addr 00005F70
[c0|00:49.270] READ_SINGLE 00003DCA
LocateFATEntry: [kernel7.im]
sdTransferBlocks read blk 00003DCB len 00000001 addr 00005F70
[c0|00:49.284] READ_SINGLE 00003DCB
sdTransferBlocks read blk 0002ABCA len 00000001 addr 00005F70
[c0|00:49.295] READ_SINGLE 0002ABCA
[c0|00:49.302] Reading file into buf at 00009860
[c0|00:49.306] kernel7.img
sdTransferBlocks read blk 0002ABCB len 00000001 addr 00005F70
[c0|00:49.314] READ_SINGLE 0002ABCB

00000000: EA000006 EA00000E EA00002B EA00000C EA00000B EA00000A EA000009 EA000008
00000020: EE110F10 E3800A01 E3C00A02 EE010F10 EE100FB0 E7E10050 E3500000 0A000002
00000040: EAFFFFFF E320F003 EAFFFFFD E3A00000 E169F000 E162F300 E166F300 E16EF300
00000060: E164F300 E160F300 E16EF200 E3A00000 E12EF300 F1020012 E3A0DA23 F1020013
```

```
00000080: E3A0DA22 F102001F E3A0DA21 EB000C29 EB000ECC F1020010 E59FD010 E59F0008
000000A0: E1A0F000 EAFFFFE6 00000000 00000000 00000000 00000000 00000000 E8CD7FFF
000000C0: E58DE03C E14FC000 E58DC040 E28DDA02 EB000ECD E24DDA02 E58D0000 E59D0040
000000E0: E16FF000 E59DE03C E8DD7FFF E1B0F00E E5801000 E12FFF1E E5900000 E12FFF1E

[c0|00:49.379] Compare output to first 256 bytes of kernel7.list
[c0|00:49.384] Done.
sdTransferBlocks read blk 00003DCA len 00000001 addr 00005F70
[c0|00:49.392] READ_SINGLE 00003DCA
LocateFATEntry: [app.bin]
sdTransferBlocks read blk 00003DCB len 00000001 addr 00005F70
[c0|00:49.406] READ_SINGLE 00003DCB
sdTransferBlocks read blk 0002B40A len 00000001 addr 00005F70
[c0|00:49.417] READ_SINGLE 0002B40A
sdTransferBlocks read blk 0002B40B len 00000001 addr 00005F70
[c0|00:49.429] READ_SINGLE 0002B40B
[c0|00:49.435] create_thread - successful file read into 00040000

00:49.441 <pc=00003C30> - Trap Handler:
00:49.444 <pc=00003C40> -    r7 00000000
00:49.448 <pc=00003C50> -    r0 00000000
00:49.452 <pc=00003C60> -    r1 F00D0000
00:49.455 <pc=00003C70> -    r2 C0FFEE00
00:49.459 <pc=00003C80> -    r3 00020FB4
00:49.463 <pc=00003C90> -    SP 00023FD8
00:49.466 <pc=00003CA0> -   EPC 000000D4
00:49.470 <pc=00003CB0> -  SPSR 200001DF
…
```

When you build the code and run it, this is exactly what you should see. If not, there is a problem, and it is most likely with your SD card or the timing between your laptop and your SD card.

This shows the normal initialization sequence, with a new addition: the use of the SD card, which is initialized at the beginning, and then it is read at the end. The initialization sequence is heavily dependent on relative timing of the commands, and **you will need a Class-10 card**, for starters. Please check as soon as possible to see if your system works correctly, because debugging timing issues with drivers and devices can take an *enormous* amount of time, and it is not something you will want to be doing the weekend that the project is due.

The code that initializes the SD card looks like this:

```
sdInitCard(NULL, NULL, true);
```

And because it may not work initially, we have put it into a ***while()*** loop that keeps trying until successful. The code that reads the SD card looks like this:

```
fh = sdCreateFile(filename, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

sdReadFile(fh, (void *)buf, 1024, &bytesRead, 0)

sdCloseHandle(fh);
```

The variable "fh" is a "file handle," which happens to be an integer index into an array of data structures in the ***SDCard.c*** module. The ***sdCreateFile()*** function opens the file up and populates a data structure with information about the file, including where it is located in the file system, and how big it is, etc. The main argument you will use is the first one: the file name, a string with the name of one of the files at the root directory of your SD card.

When the file handle that ***sdCreateFile()*** returns is passed to the ***sdReadFile()*** function, the ***sdReadFile()*** function can use the previously discovered and stored information about the file to go find it and load it. This means that you do not have to know anything about sectors, blocks, or even the FAT filesystem structure.

The arguments of the ***sdReadFile()*** function are as follows:

- ***file handle*** - data value returned by the ***sdCreateFile*** function
- ***buffer*** - address into which the data should be read

- ***size*** - the amount of bytes to read from the file into the buffer
- ***return: bytes read*** (a pointer to a ***uint32_t*** variable) - a return value indicating the amount of data actually read by the function
- ***unused*** (leave as 0)

The first three arguments are the ones you will care the most about.

Lastly, the ***sdCloseHandle()*** function should be fairly self-explanatory, and it should be called as soon as you are done using the file.

Build It, Load It, Run It

Implement the following system calls:

- **read**()
- **write**()
- **open**()
- **close**()

Implement the following devices:

- LED — writing 1/0 turns the LED on/off.
- CLOCK — reading the clock gets the current time of day.
- LOG — writing to this device puts a string into the kernel log.
- DISK — you can open and read files on the SD card.

Your implementation is up to you, and how you handle illogical pairings (e.g., reading, opening, or closing the LOG; writing the CLOCK; etc.) is also up to you.

Once you have it working, show us.