**Purpose**

In this project, you will get a taste of what dealing with multicore is like. You will create a new system call to get the Core ID of the core you are running on (for debugging purposes), and then experiment with turning on all four cores of the CPU.

**Create a System Call for get_coreid()**

Your first task is to get the ID of the core your code is running on. If you try to call `core_id()` directly from the shell, you will find that nothing happens. This is because the shell runs in user mode, whereas getting access to the special control registers such as the one holding the Core ID is a privileged action. Therefore, you will have to ask the operating system for the ID of the core you are on.

Do this by adding a new system call. You have two primary options:

- Create a new system call just for getting the core ID, which involves creating a new entry in the list of system calls, a new function to execute, and a new entry in the `switch()` statement in the kernel’s trap-handler routine.
- Define a new device and call the `syscall_read_word()` system call on it, which involves creating a new device type and adding a new entry to the `devtab[]` array in the `io.c` module.

The choice is yours; either is fine. In the mean time, to get your code to build, you can simply define the function `get_coreid()` to be a constant, and use that until your system call is working.

**Verify the Code Works**

You have been given what is effectively a solution to Project 4. It is my Project 5 solution, with the mechanisms for non-blocking I/O stripped out. It also has a few other additions.

You will see that, in the project directory, there is no `1_boot.s` file, but there are instead a few other similarly named files. Here is the primary difference between the first two. Note that the `res_handler` block is the code called right at initialization.

### `1_boot.good`

```
res_handler:
  mrc p15, 0, r0, c1, c0, 0 @ Read System Control Register
  @ orr r0, r0, #(1<<2) @ dcache enable
  orr r0, r0, #(1<<12) @ icache enable
  and r0, r0, #0xFFFFFDFF @ turn on vector table at 0x0000000 (bit 12)
  mcr p15, 0, r0, c1, c0, 0 @ Write System Control Register

    // check core ID
  mrc    p15, 0, r0, c0, c0, 5
  ubfx  r0, r0, #0, #2
  cmp  r0, #0 // is it core 0?
  beq  core0  // if so, branch to core0

    // it is not core0, so do things that are appropriate for SVC level as opposed to HYP
    // like set up separate stacks for each core, etc.

  b    hang
  .globl hang
  hang:  wfi
    b    hang
```
core0:
    // Initialize SPSR in all modes.
    MOV  R0, #0
    MSR  SPSR, R0
    MSR  SPSR_svc, R0
    MSR  SPSR_und, R0
    MSR  SPSR_hyp, R0
    MSR  SPSR_abt, R0
    MSR  SPSR_irq, R0
    MSR  SPSR_fiq, R0
    .
    .

1_boot.multi

res_handler:
    _mrc p15, 0, r0, cl, c0, 0 @ Read System Control Register
    @    orr r0, r0, #{1<2} @ dcache enable
    @    orr r0, r0, #{1<12} @ icache enable
    @    and r0, r0, #0FFFFFDFFF @ turn on vector table at 0x00000000 (bit 12)
    mcr p15, 0, r0, cl, c0, 0 @ Write System Control Register

    // Initialize SPSR in all modes.
    MOV  R0, #0
    MSR  SPSR, R0
    MSR  SPSR_svc, R0
    MSR  SPSR_und, R0
    MSR  SPSR_hyp, R0
    MSR  SPSR_abt, R0
    MSR  SPSR_irq, R0
    MSR  SPSR_fiq, R0
    .
    .

The file 1_boot.good is just the 1_boot.s file you have seen before. In it, immediately after turning on the instruction cache and setting up the vector table, the code checks to see what core it is running on, and if the core ID is 0, the software branches to the core0 label and continues with the initialization. If the core ID is not 0, the software goes into an infinite loop.

The second file removes all of that and simply goes straight into initialization.

Use the file 1_boot.good to begin with (copy it to 1_boot.s and compile). When you run your code, you should see the familiar start-up:

```
[c0] 00:02.022] ...
[c0] 00:02.024] System is booting, kernel cpuid = 00000000
[c0] 00:02.029] Kernel version: [p6, Mon Apr 1 14:04:32 EDT 2019]
[c0] 00:02.035] create_thread:
[c0] 00:02.038] NULL thread 00000000
[c0] 00:02.041] stack = 00021000
[c0] 00:02.044] start = 00000044
[c0] 00:02.047] tcb = 000DDE18
[c0] 00:02.050] create_thread:
[c0] 00:02.053] shell 00000001
[c0] 00:02.055] stack = 00022000
[c0] 00:02.058] start = 00022E8
[c0] 00:02.061] tcb = 000DDE7C
[c0] 00:02.064] ...
[c0] 00:02.066] Init complete. Please hit any key to continue.
```

Running the eggshell on core 0.

Available commands:
    RUN = 004E5552
    PS = 00053535
    TIME = 454D4954
    LED = 0044454C
    LOG = 00474F4C
    EXIT = 54495845
    HANG = 474B4148
    DUMP = 504D5544

Please enter a command.
[c0]
At this point, if you list out the processes, you will see the following:

Please enter a command.
cd> PS
CMD_PS
[c0] 00:26.650] PS: Active processes ...
[c0] 00:26.654] Dumping TCB for thread 00000001
[c0] 00:26.658] shell 00000001
[c0] 00:26.661] stack 00022000
[c0] 00:26.664] tcb @ 0000DE7C
[c0] 00:26.666] r0 00000001
[c0] 00:26.669] r1 0000000A
[c0] 00:26.672] r2 00005350
[c0] 00:26.675] r3 00005350
[c0] 00:26.678] r4 00021FC8
[c0] 00:26.680] r5 00000000
[c0] 00:26.683] r6 00000000
[c0] 00:26.686] r7 00000009
[c0] 00:26.689] r8 00000000
[c0] 00:26.692] r9 00008458
[c0] 00:26.695] r10 00000000
[c0] 00:26.697] r11 474B4148
[c0] 00:26.700] r12 00021FC2
[c0] 00:26.703] sp 00021FA4
[c0] 00:26.706] lr 000026B8
[c0] 00:26.709] pc 00019BC
[c0] 00:26.711] spsr 00000150

Please enter a command.
cd>

We can invoke the blinker loop, and if you look in the kernel7.list file, you should see that it is located at address 0x21c4 (verify this, though … your compiler may produce different results than mine):

```
000021c4 <do_blinker>:
  21c4:  e92d41f0  push   {r4, r5, r6, r7, r8, lr}
  21c8:  e3a04000  mov   r4, #0, 0
  21cc:  ela0b004  mov   r8, r4
  21d0:  e99f505c  ldr   r5, [pc, #92] ; 2234 <do_blinker+0x70>
  .......
```

You can invoke the blinker by using the RUN command:

Please enter a command.
cd> RUN LED 0X21C4
CMD RUN [LED, 000021C4]
[c0700:51.549] SYSCALL_START THREAD name = 0044454C
[c0] 00:51.553] SYSCALL_START THREAD addr = 000021C4
[c0] 00:51.558] LED 000021C4
[c0] 00:51.561] create_thread:
[c0] 00:51.563] LED 00000002
[c0] 00:51.566] stack = 00023000
[c0] 00:51.569] start = 000021C4
[c0] 00:51.572] tcb = 0000DEE0

Please enter a command.
cd>

The command reports back that it worked, but you will notice that the LED does not start blinking. As you type, you might notice it changing from on to off and back, but it will not do the 1/2/3/4 pattern as it should. You can verify that the RUN command has created a thread by listing the processes out again, which should produce the following result:

Please enter a command.
cd> PS
CMD PS
[c0701:19.300] PS: Active processes ...
[c0] 01:19.304] Dumping TCB for thread 00000001
[c0] 01:19.308] shell 00000001
[c0] 01:19.311] stack 00022000
[c0] 01:19.314] tcb @ 0000DE7C
[c0] 01:19.317] r0 00000001
[c0] 01:19.320] r1 0000000A
[c0] 01:19.322] r2 00005350
So the “LED” thread has indeed been created. It is on the list of active threads. It has thread ID 2. But the LED is not blinking. Why is it not blinking?

Recall that this is the problem Project 5 is intended to solve. The LED is not blinking because the CPU is currently blocking, waiting on you to type into the keyboard. Each time you enter a command, a little bit of work is done, and the shell goes right back to asking you to input a new command — at which point the kernel stalls, waiting for your keyboard input.

Here is how we can get around that … it is a hack that will let you see that two threads are indeed being run and switched between:

```plaintext
Please enter a command.
c0> HANG 10
CMD_HANG ...
00000000A
00000009
00000008
00000007
00000006
00000005
00000004
00000003
00000002
00000001

Please enter a command.
c0>
```

The “HANG” command counts down from whatever number you give it and does nothing, simply printing to the screen roughly once per second as it counts down to zero. Once it reaches zero, it goes back into the normal control loop, waiting for input. But while it is counting down, it is not blocking on I/O, which means that the kernel can schedule other threads. So while it is counting down, you should see the LED going through its 1/2/3/4 blinking cycle.
**Turn on Multicore**

Once you get to this point, rebuild the kernel using the file `1_boot.multi` as `1_boot.s`. What happens when you boot this kernel? Does it behave the same way every time you boot? What is causing its behavior?

You might think that the main problem is that the cores are all using the same stacks. That is true, but it is not the main problem. A more extensive boot module can be found in `1_boot.stacks`, which has each core do a different initialization routine. Each gets a different set of stack pointers, and there is no overlap. You can even change things so that each core starts up a different thread. There will still be a problem with the use of shared resources, and this fundamental problem will not go away easily.

**Your assignment for this portion of the project:** think about how you would solve this problem and write up a solution. It only needs to be a page, at a high level of detail … but how would you deal with this problem? With multicore, you now have the problem at a kernel level that you previously had at the application level on a single processor: you cannot easily share resources without having everyone step on each others’ toes.

Here is how the operating system’s kernel steps in to manage application access to the various hardware devices:

![Diagram of kernel management](image)

Rather than giving all applications direct access to all devices, which could easily turn chaotic, the applications go through a single access point: the kernel. Funneling all requests through a single point serializes the requests, so that a known order can be determined, and so that a given device is not manipulated incorrectly or out of sequence. Rather than having applications know the hardware-device protocols, all device-specific information is encapsulated in the kernel (in its drivers), and the application-level interface to the array of hardware devices becomes, instead, just the system-call interface, which invokes the kernel on the application’s behalf.

We created the operating system’s kernel itself to solve that problem at the application level: the kernel is there to manage access to shared resources, so that applications need not worry about it. However, with multicore we have the same problem, but at the kernel level. Once we move to multicore the problem...
returns, because within a multicore CPU, there are multiple processor cores sharing a single set of hardware devices. While some hardware devices may be replicated across each core, as in the timer example below, the majority of the devices are grouped as a set of shared resources that all of the CPU cores must manage together. This is the setup:

How would you solve this problem? There is no right/wrong answer … the point is to give it some serious thought and write up a description of your solution.

**Build It, Load It, Run It**

Once you have it working, show us.