# Project 5: Multicore Madness (4%)

**ENEE 447: Operating Systems** — Spring 2021
**Assigned:** Tuesday, Apr 6; **Due:** Sunday, Apr 18

## Purpose

In this project, you will get a taste of what dealing with multicore is like. You will experiment with turning on all four cores of the CPU, which will create a bit of chaos. Once you stare at the chaos for a bit, you are to perform a thought experiment to see how one might design an operating system kernel to run on multiple cores.

## Verify That the Code Works

You have been given what is effectively a solution to Project 4 (my Project 4 solution), with some changes made to the ***1_boot.s*** file.

In the project directory, there is no ***1_boot.s*** file, but there are instead two other similarly named files. Here is the primary difference between the 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, #0xFFFFDFFF   @ 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
    .
    .
    .
```

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 jumps to an infinite loop, effectively taking cores 1, 2, and 3 off-line. Only core 0 proceeds, so the kernel to date has only run on a uniprocessor.

The second file removes all of that and simply goes straight into initialization. The file ***1_boot.multi*** starts up all of the cores:

## *1_boot.multi*

```
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, #0xFFFFDFFF   @ turn on vector table at 0x0000000 (bit 12)
    mcr p15, 0, r0, c1, 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
    .
    .
    .
```

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|00:02.265] ...
[c0|00:02.266] System is booting, kernel cpuid = 00000000
[c0|00:02.272] Kernel version: [p5, Tue Apr 6 18:27:07 EDT 2021]
[c0|00:02.277] Initializing SD Card ...
[c0|00:02.281] ---------------> sdInitCard [init]
[c0|00:02.286] EMMC: reset card.
[c0|00:02.289] EMMC: setting clock speed to  00061A80
[c0|00:02.294] GO_IDLE_STATE 00000000
[c0|00:02.297] SEND_IF_COND 000001AA
[c0|00:02.300] APP_CMD 00000000
[c0|00:02.303] SD_SENDOPCOND 50FF8000
[c0|00:02.707] APP_CMD 00000000
[c0|00:02.710] SD_SENDOPCOND 50FF8000
[c0|00:02.714] ALL_SEND_CID 00000000
[c0|00:02.717] SEND_REL_ADDR 00000000
[c0|00:02.720] SEND_CSD AAAA0000
[c0|00:02.723] EMMC: setting clock speed to  017D7840
[c0|00:02.728] CARD_SELECT AAAA0000
[c0|00:02.731] APP_CMD AAAA0000
[c0|00:02.734] SEND_SCR 00000000
[c0|00:02.740] SET_BLOCKLEN 00000200
sdTransferBlocks read blk 00000000 len 00000001 addr 00020D70
[c0|00:02.749] READ_SINGLE 00000000
sdTransferBlocks read blk 00002000 len 00000001 addr 00020D70
[c0|00:02.761] READ_SINGLE 00002000
[c0|00:02.776] ---------------> sdInitCard [term]
[c0|00:02.780] SD Card working.
[c0|00:02.783] ...
[c0|00:02.785] Init complete. Please hit any key to continue.
```

At this point, if you hit <***enter***> you will see the Project 4 behavior: blinking light, and two repetitive threads printing to the log.

## 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 correctly?

Does it behave as you expected it to?

Does it behave the same way every time you boot? What does that tell you?

Here is just one example of a boot sequence that you might see (note that the initial "c#" at the beginning of a log entry indicates which core number is generating the entry):

```
[
 cc0|0::0..65]  ..

                    [c1|0::2..66]]SSystm  ssboooiing,keenee cppuid==000000030
                                                                              [c2||0::2..22]
KKernelversion:: [p5, TueApr 6 18:224:47 EDT20211

[[c||0::2..22] ---------------- sddnntCCar  [iitt]

[21|0::0..88]  EMM: rrsett arr.
cc1|0::2..29]  MMM: seetinn  cocckssee to  00001188
[[c0|0::2..94]]GO_IDDL_STATT  00000000
[[cc32||00:c1|29:] SD_TI SOUTI0EO00 020
                                       [c02
a[d3|0:] .MMC] rese: rardt
:c1|30:] EMMC: sMtting tlong speed soeed00o1A80
1A80
000000:] GO_I] G_STATE_S000E 00
T[c0000:02.04.3[c] SD:TI.EOU] 0D000000
set3|ard. .[c] EMMC:.rese EMaC:
           [c] EM:C: settiEMMCl cktspeed lo k00p61A 0o [c0061:80.
                                                                  [c] GO:IDLE_S]
TE_IDLE_00ATE 00000000
EMMC| setting clEMkC: sed tn  c0061Asp
[c3o  :0061A8]
_STATE 00000000Oc1DLE_STATE ] 0000DL
SDc3|ME:UT.000] 002CD_TIMEO|T 0000000]
                     ce|et:ca.d. ] EcMC: :es.t c] E.M
 sp2|d to  0006 GO_IDLE_STATE 0000]000_
                                       [c3|ST:TE.000]00EN
IF_COND 000001AA
[c1|[c0|00::7..55]]SDD_TIMOU[c2|00000.

] 0PP_CM. 00000EMC
[c1||0::0.366] EMMC: settiing lloc sspedd t   0061AA0

c10|0::0..66]]GGO_DDL_STAAEE 00000000
                                     [c20|0::7..70] SD_D__IFPCOND0000F1AA0
[c1|00:07.373] APP_CMD 00000000
[c0|00:07.376] APP_CMD 00000000
[c1|00:07.379] SD_SENDOPCOND 50FF8000
[c0|00:07.382] SD_SENDOPCOND 50FF8000

…
```

What do you think is causing this odd interlaced-core behavior?

At the very least, it is comforting to see that all of the cores are working correctly, but nobody likes to see gibberish in the log file. :)

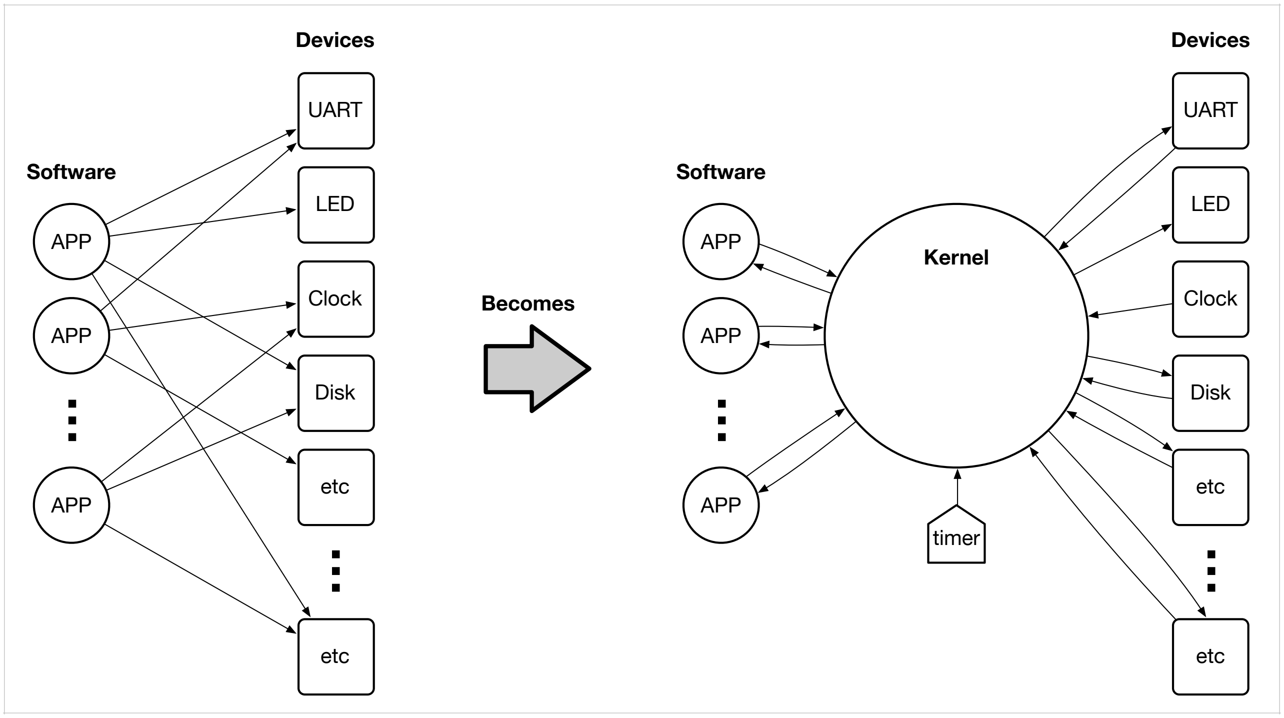
You might think that the main problem is that the cores are all using the same stacks. They are not: each gets a different set of stack pointers, and there is no overlap. The problem is the use of shared resources, and this fundamental problem will not go away easily.

## Your Assignment

**Your assignment for this 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.

Let's illustrate that directly.

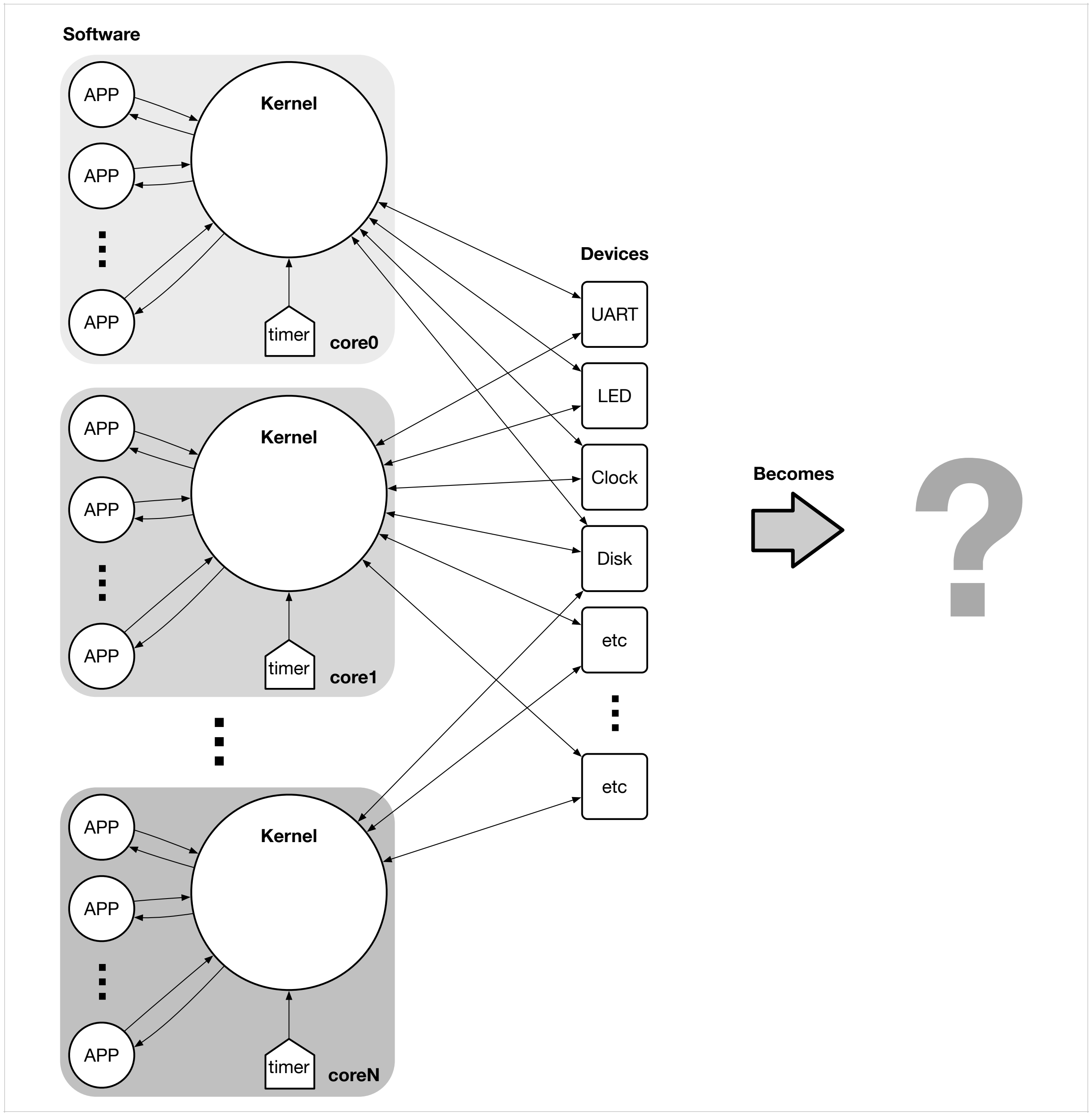
In the following diagram we show how the operating system's kernel steps in to manage application access to the various hardware devices. On the left side is shown a system in which every user-level application has direct access to every shared hardware resource. As one might imagine, this results in chaos, because you will frequently encounter scenarios in which multiple threads want to access the same hardware resource at the same time. On the right side is shown how the kernel acts as a gate-keeper that effectively serializes access to the various shared resources. User applications must go through through the kernel to get at the hardware devices, and the kernel ensures that only one request at a time is actively using each resource.



Rather than giving all applications direct access to all devices, 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 hardware 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.

In the following diagram we show the setup. On the left side is shown a multicore system, with each core running a copy of the kernel. While each core may have its own timer, there is still a large number of shared hardware resources (UART, LED, Clock, Disk, Network, etc.) that are not found on a per-core basis but are instead shared among all of the cores. This creates the same problem as in the diagram. above: you have multiple kernels that all potentially want to access the same hardware device at the same time. On the right side is your solution to the problem … what does it look like?



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 — As a Thought Experiment

Once you have it written up, share with us.