Project 4: Preemptive Context Switch (4%)

ENEE 447: Operating Systems — Spring 2021
Assigned: Tuesday, Mar 9; **Due:** Sunday, Mar 28

Purpose

In this project you will implement context switching on the Raspberry Pi, using perhaps the simplest possible scheduling algorithm: on every timer tick you will round-robin between three processes (i.e., if thread 0 is running, change to thread 1; if thread 1 is running, change to thread 2; if thread 2 is running, change to thread 0). The three threads will be in the same address space, so we will not have to worry about saving and restoring anything other than the register file contents (for instance, once we have virtual memory running, you will have to save special control registers related to that). Context switching obviously represents the underpinning of all multitasking and multiprocessing and is thus one of the operating system's most fundamental and powerful mechanisms. From this point, you will be able to implement much more sophisticated scheduling algorithms and juggle any number of simultaneous threads.

Context Switch in ARM

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

The IRQ vector shares a number of registers with code running in USR mode: **r0–r12** and the **program counter** are **common**, while the IRQ vector runs in a mode that has its own **stack pointer** (**r13**) and **link register** (**r14**).

Among many other things, what this means is that, assuming you have a register-save area of sufficient size, located at **threadSave**, then the following code will save all of the registers visible in USR and SYS modes:

```
save_r13_irq: .word 0

irq_handler:

    ldr     sp, =threadsave
    stmia   sp,{r0-lr}^     @ Save all user registers r0-lr
                      @ (the ^ means user registers)

    str lr,[sp,#60]     @ store saved PC on stack

    str     lr, save_lr_irq           @ save the SVC lr
    mrs lr, SPSR        @ load SPSR (assume ip not a swi arg)
    str lr,[sp,#64]     @ store on stack
    ldr     lr, save_lr_irq           @ save the SVC lr

    // regs saved, we can now destroy stuff

    //
    // clear timer interrupt (we get here from timer)
    //
    mov     sp, #SVCSTACK0
    bl      clear_timer_interrupt

    @ clobber the user stack - simulates effect of another thread running
    @ clobber the user stack - simulates effect of another thread running
    @ clobber the user stack - simulates effect of another thread running

    mov     r2, # SYS_mode
    msr     cpsr_c, r2
    ldr     r0,badval
    ldr     r1,badval
    ldr     r2,badval
    ldr     r3,badval
    ldr     r4,badval
    ldr     r5,badval
    ldr     r6,badval
    ldr     r7,badval
    ldr     r8,badval
    ldr     r9,badval
    ldr     r10,badval
    ldr     r11,badval
    ldr     r12,badval
    ldr     r13,badval
    ldr     r14,badval
    mov     r2, # IRQ_mode
    msr     cpsr_c, r2

    @ clobber the user stack - simulates effect of another thread running
    @ clobber the user stack - simulates effect of another thread running
    @ clobber the user stack - simulates effect of another thread running

    // reset the timer
    bl      set_timer

    // restore the registers
    ldr     sp, =threadsave
    ldr r0,[sp,#64]     @ pop saved CPSR
    msr SPSR_cxsf, r0       @ move it into place

    ldr lr,[sp,#60]     @ restore address to return to

    @ Restore saved values.  The ^ means to restore the userspace registers
    ldmia   sp, {r0-lr}^
    subs    pc, lr, #4                  @ return from exception
```

This code does several things. First, it saves the thread context on an array of words pointed to by **threadSave**. Once those values are saved, the code is free to destroy the register file contents (which simulates a context switch to another thread). The handler changes to SYS mode, which shares the same register file as USR mode, and it loads a garbage value into registers 0–14. Then it jumps back into the IRQ handler's mode, restores the previously saved state, and exits.

This code is given to you in the project source directory for **p4**. The entire project, as presented to you, compiles and runs. With it, you will build a preemptive context switch.

Here's how it works. The code begins by performing the following save-register functions:

- The address of the save location is loaded into the ***sp*** register, which does not destroy the USER mode's copy of the ***sp*** register (see previous write-ups on the ARM register file).
- Registers r0–r14 are stored upwards starting at this address. These are the user registers, so the ***sp*** and ***lr*** registers are the user's copies.
- The return address is stored at the next address, which would correspond to the location for r15. This is because the return address is r15, as that ***is*** the Program Counter in the ARM32 architecture.
- Last, at the next location beyond that, we store the process's saved SPSR.

Therefore, one can think of the register set being saved as looking like the following in a data structure:

```
    REG_r0,
    REG_r1,
    REG_r2,
    REG_r3,
    REG_r4,
    REG_r5,
    REG_r6,
    REG_r7,
    REG_r8,
    REG_r9,
    REG_r10,
    REG_r11,
    REG_r12,
    REG_sp,
    REG_lr,
    REG_pc,
    REG_spsr,
```

That is exactly the data that is save and restored for a context switch. These values, in that order, are stored in the following structure, for which there is one for every process in the system:

```
struct tcb {
    char    name[NAMESIZE];
    long    threadid;
    long    stack;
    long    regs[17];   // 17th reg is the SPSR saved by context switch
} tcbs[ NUM_THREADS ];
```

Note that we have a statically-declared set of thread structures, ***tcbs[NUM_THREADS]***. The kernel in this project loads three apps into the first three TCBs for you.

Note that the ***name***, ***thread ID***, and ***stack*** are static values: they should not change. When a thread runs, all of its context is stored in the ***regs*** portion of the TCB. For instance, the ***stack*** member of the struct is the ***statically assigned starting point*** for the threads stack: as will be discussed in the next section, user thread stacks begin at 0x30000, and each thread is given its own 4KB segment: thread 0 gets the first 4KB; thread 1 gets the next; etc. This will change once we have virtual memory, but just remember not to modify this value, and when a new thread starts up, use that as its initial stack pointer (for instance, you could put the value in ***REG_sp***).

Back to the assembly code; once the registers are saved, the code calls C-language routines to do the work. Because the handlers cannot be interrupted by themselves, we simply assign the same static starting stack location for each handler invocation. This works because we are running single-core for the moment.

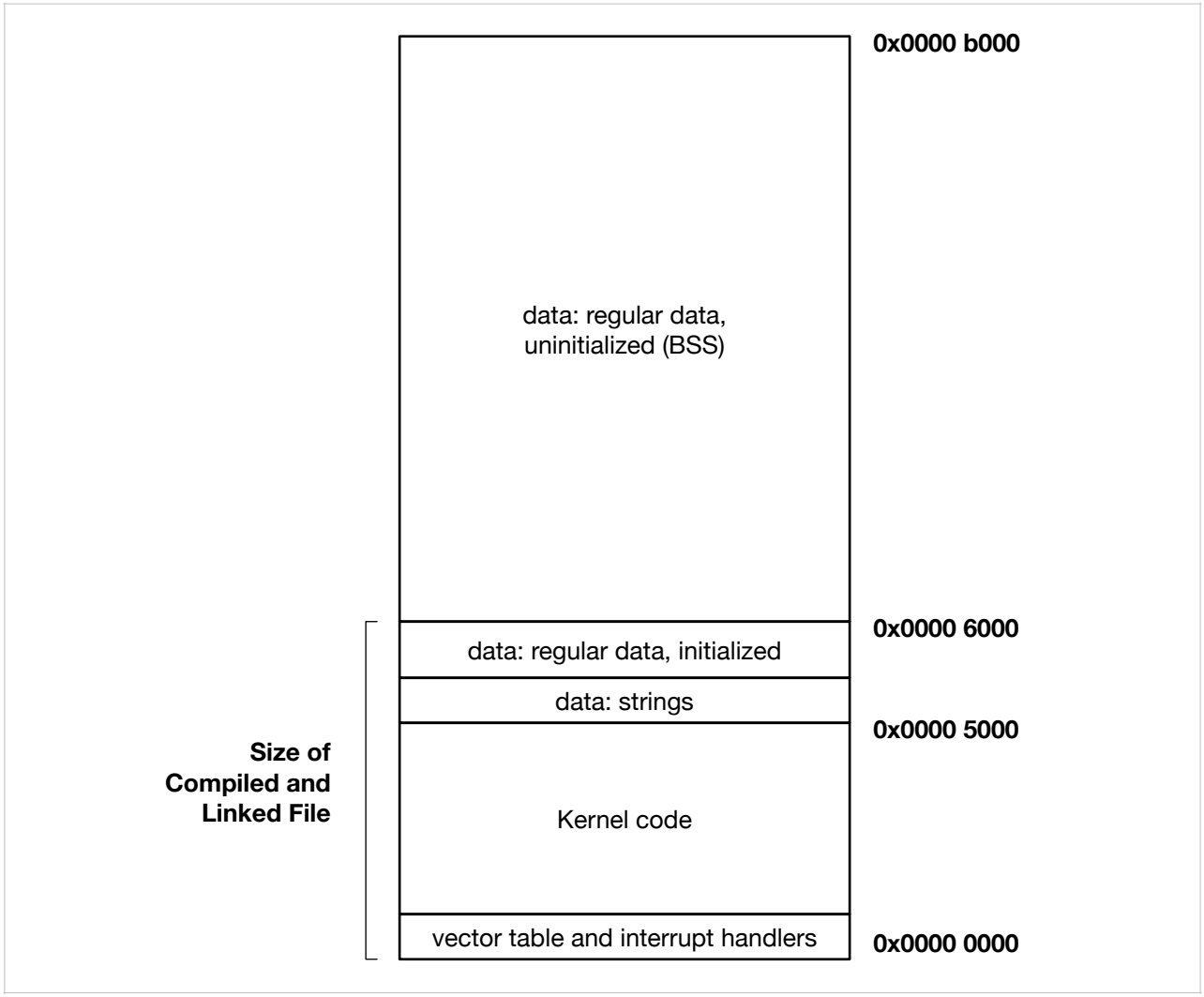
After the C-language routines finish, we restore state from the thread control block:

- The address of the save location is loaded into the ***sp*** register.

- First, from the topmost location (corresponding to what would be r16 in the TCB), we restore the process's saved SPSR.
- Next we retrieve the user process's return address from the TCB location corresponding to r15. This is held temporarily in the ***lr*** link register, which is not the user link register.
- Lastly, registers r0–r14 are restored upwards starting at the bottom address. These are the user registers, so the ***sp*** and ***lr*** registers are the user's copies and do not overwrite the "sp" or "lr" registers being used by the handler code.

What Address?

One important issue in this project is figuring out where to put things. Here is a basic structure for the kernel executable file. This is what is in ***kernel7.img*** and what is shown in human-readable form in ***kernel7.list***.



To find this information out, you must look through the file ***kernel7.list***. This is extremely important, in general, because it is the easiest way to figure out your code size and code layout. Note that if you simply rely upon the listed file size for the kernel binary, you might be misled into thinking that its size is something that it is not. When you look at the compiled size of the kernel file, your laptop will report that the size of the file ***kernel7.img*** (or ***kernel7.bin***) is roughly 23K.

Why is this worth paying attention to?

This is why:

0xb000 ≠ 23,000

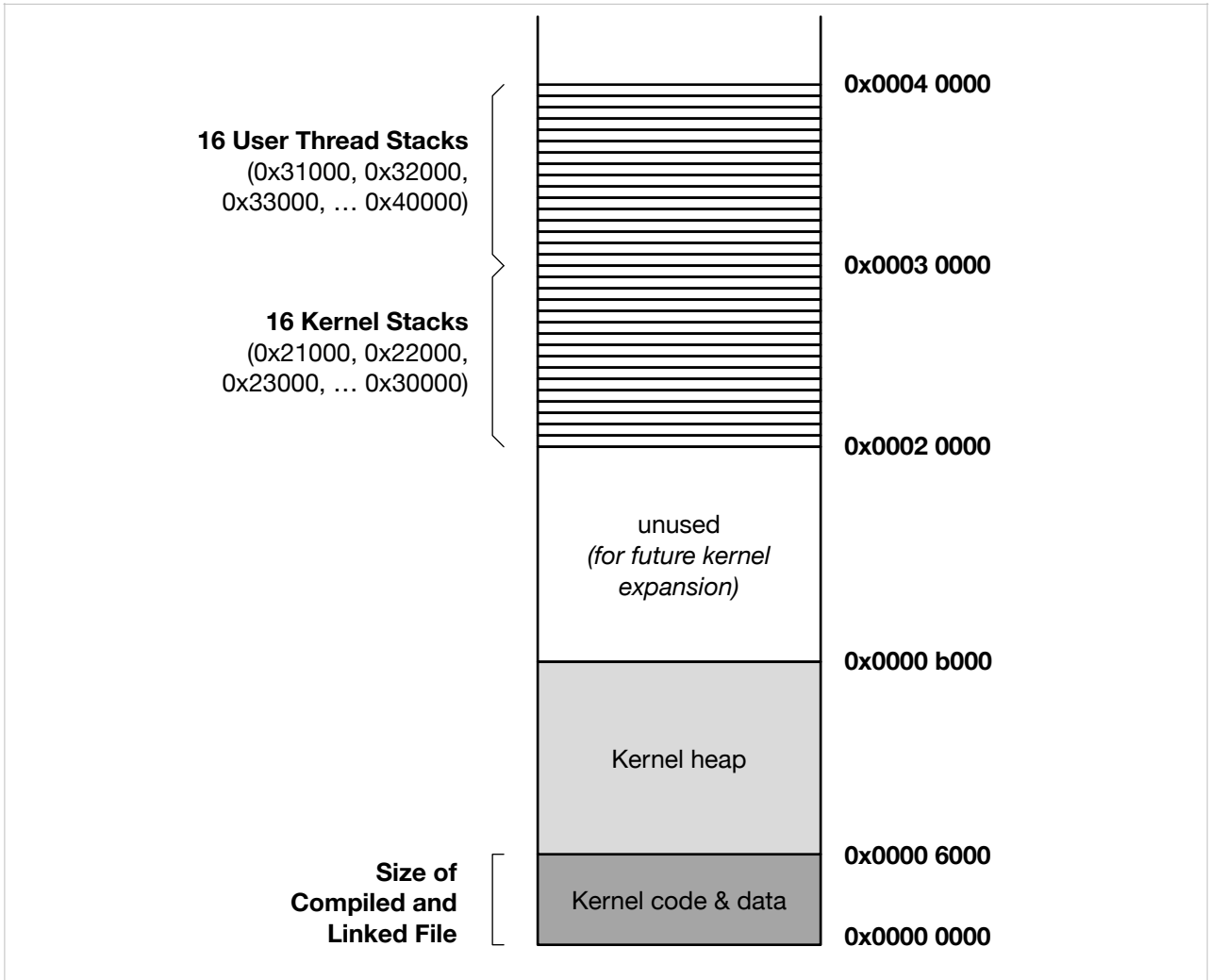
The decimal value of 0xb000 is closer to 43K, not 23K. The amount of memory that the kernel uses is roughly double the size of the file as stored on disk. If you tried to put the various stacks and things right after the 23K mark, you would be interfering with the kernel's heap.

Moral of the story: don't ever use the binary size as an indication of memory footprint.

Anyway, we have to address the following questions:

- Where should the kernel stacks go?
- Where should the thread stacks go?
- Where should the user application binaries go?

We have placed the kernel stacks in the 0x0002xxxx range (there are sixteen, but only a few are used at the moment), and we have placed the thread stacks (there are only 16 of them, for now) in the 0x0003xxxx range. This is shown below:



The kernel stacks are assigned statically in the ***1_boot.s*** module, and the user-thread stacks are assigned statically in the ***threads.c*** module.

One thing that you will have to do is tell the compiler where the apps are; for now, and until we implement virtual memory, this needs to be a static decision. So, for example, if you decide that an app should be loaded at location 0x40000, then you need to edit the shell's ***memmap*** file to reflect this. This has been done for you, for all three applications (at 0x40000, 0x60000, and 0x80000, which is where kernel.c loads them at init time).

The ***memmap*** files have the following format:

```
MEMORY
{
    ram : ORIGIN = 0x0000, LENGTH = 0x400000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
}
```

This is an extremely simple linker file (do a little research, and you will see … this is wonderfully simple, all thanks to David Welch). The main thing you should work with is the ORIGIN variable in the top part. This tells the linker where the application will start. Because all of the addresses will be different for each application (we are not yet implementing virtual memory), each will have to be loaded into a region that does not overlap with anything, and you will need to modify each application's ***memmap*** file to reflect the location into which it will be loaded. Yes, this is a pain in the butt, and it is one of the reasons that virtual memory is so awesome. But, again, it's already been done for you.

Implement Context Switch

There are three "user applications" to switch between:

- ***app1*** — This blinks the LED on and off at roughly 1-second intervals.
- ***app2*** — This repeatedly gets the time of day and then sends that value to the kernel log, at roughly 1-second intervals.
- ***app3*** — This repeatedly writes a string to the kernel log at roughly 2/3-second intervals.

The code you are given loads the three binaries into thee different sections of the memory system, turns on the system timer to wake up periodically (roughly 1/10-second intervals) and then starts up the first application, which is thread 0. You should see the LED blinking on and off regularly, and the periodic timer will wake up and print a dot to the screen every 16th invocation, so you should see a line of dots forming across your screen as the LED blinks. When you see this happening, it indicates to you that everything is working correctly: your RPi is running the first thread regularly; the SVC interrupt is working correctly; the timer is running correctly; and the timer and SVC interrupts do not interfere with each other.

Your task is to write code that will swap between all three apps. Knowing that the code above works, this should be straightforward, as the code above is a context-switch code. If your code is implemented correctly, it should look like all threads are running "simultaneously." If you slow the timer interrupt down in **time.c**, for instance once every second or even every ten seconds, you should see only one app working at a time. If you speed the timer up, you should see problems as the cost of handling the interrupts grows significant.

Build It, Load It, Run It

Once you have it working, show us.

