Project 4: Context Switch (4%)

ENEE 447: Operating Systems — Spring 2012
Assigned: Monday, Feb 22; **Due:** Friday, Mar 4

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 switch back and forth between two processes (i.e., if thread 0 is running, change to thread 1; if thread 1 is running, change to thread 0). The two 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. 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

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:

```
irq_nop:
    str     r13, save_r13_irq          @ save the IRQ stack pointer
    ldr     r13, =threadSave           @ load the IRQ stack pointer with address of TCB
    add     r13, r13, #56              @ jump to middle of TCB for store up and store down
```



```
stmia   sp, {sp, lr}^              @ store USR stack pointer & link register, upwards
push    {r0-r12, lr}               @ store USR regs 0-12, IRQ link register, downwards

    @@@@
    @@@@ DO WHATEVER IS NECESSARY TO CLEAR THE INTERRUPT (if anything)
    @@@@

mov r0, #1
bl  clear_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

ldr     r13, =threadSave           @ load the IRQ stack pointer with address of TCB
pop     {r0-r12, lr}               @ load USR regs 0-12 and IRQ link register, upwards
ldmia   sp, {sp, lr}^              @ load USR stack pointer & link register, downwards
nop                                @ evidently it's a good idea to put NOP after LDMIA
ldr     r13, save_r13_irq          @ restore the IRQ stack pointer from way above
subs    pc, lr, #4                 @ return from exception
```

This code does several things. First, it saves the stack pointer **sp/r13** into a known location. Then, it saves the thread context on an array of words pointed to by **threadSave**: this is done by first jumping into the middle of the array, storing two values upward, and then storing 14 values downwards. Once those values are saved, it 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 (note that the "application" code is running in SYS mode, so that it has direct access to the GPIO registers that drive the LEDs), 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 the **kernel** loop on **core0** interrupting the application loop on **core1** 1000 times per second, and **core1** running this handler code on every interrupt. The user code is a simple app called **entry_t0** that blinks the green LED one, twice, three times, then four times, and repeats. The idea is that this sequence should always repeat and never restart mid-stream (in which case you jumped to the beginning of the thread instead of saving it and restoring it).

Feel free to set the kernel's interrupt frequency to different values to see what happens — for instance, how fast can you interrupt **core1** before it becomes noticeable?

Implement Context Switch

Your task is to write code that will swap between two different apps: one that blinks the green LED and one that blinks the red LED (these are found in the **z_applications.c** file). Knowing that the code above works, this should be straightforward, as the code above is a context-switch code. It is a bit more involved, however, as you must perform the following functions:

- Every interrupt, you must save the currently executing context and restore the other
- You must start up the second thread (**entry_t1**) if it is not already running

If it is done correctly, both the green and red LEDs should cycle on the sequence of blink once, blink twice, blink three times, blink four times, repeat. If you slow the kernel's interrupt time to every second or slower, then you should see that only one thread runs at a time: when the green LED is blinking, the red LED is not, and *vice versa*. When the kernel interrupts many times per second, it will look like both LEDs are blinking from simultaneously running threads.

Build It, Load It, Run It

Once you have it working, show us.

