Project 4: Preemptive Context Switch (4%)

ENEE 447: Operating Systems — Spring 2019

Assigned: Tuesday, Mar 5; **Due:** Sunday, Mar 17

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 (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_nop:
    // save the registers
    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

    @ regs saved, we can now destroy stuff

    //
    // clear timer interrupt (we get here from timer)
    //
    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     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, 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 exception of the code you are currently writing for Project 3. With it, you will build the penultimate feature of The Simplest Possible Operating System™ — the preemptive context switch.

The Simplest Possible Operating System™

As said before, 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 implemented a system-call facility and provided an extremely rudimentary shell that runs in user mode — but still within the kernel proper, for now. Project 4 adds to that the preemptive context switch and a rudimentary scheduler: switch processes every time quantum. The code for saving and restoring state is provided above, and your job is to use this to create the scheduler and context-switch mechanism.

The scheduler will be invoked through a timer interrupt that comes into the kernel through the IRQ vector. Routines in the ***time.c*** module have been written for you, to interface with the timer. The IRQ interrupt is invoked 100 times per second by setting the appropriate value in the *set_timer()* routine in ***time.c:***

```
void
set_timer()
{
    PUT32(TIMER_Load, 10);  // time in millisecs

    PUT32(Enable_IRQs_1, 64);
    PUT32(Enable_Basic_IRQs, 0x1);
    enable_irq();
    return;
}
```

If you put a smaller/larger value in the first line, the IRQ interrupt will fire less/more frequently. Feel free to set the kernel's interrupt frequency to different values to see what happens — for instance, how fast can you interrupt the core before it becomes noticeable? You can speed the timer up faster than 1/ms, but it requires you to comment out the line that does the "1000x downsample." At some point, the frequency of interrupts will overwhelm the processor.

Implement Context Switch

There are two "user applications" to switch between:

- ***run_shell()*** in ***z_shell.c*** — This is the shell that you have been given for Project 3, except that the while() loop has been edited so that it simply prints out the prompt, waits about a second, and skips back to the top of the loop. This means that the shell simply prints something to the screen at a regular time interval. We will override that at the end.
- ***do_blinker()*** in ***z_blinker.c*** — This application blinks the LED in a repeating 1, 2, 3, 4, 1, 2, 3, 4 pattern (it blinks once, then twice, then three times, etc.).

Thus we have two tasks that should run in USR mode and which perform regular I/O, but to non-conflicting devices (one is to the LED, the other is to the UART). Your task is to write code that will swap between two different apps. 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 (**do_blinker**) if it is not already running

If your code is implemented correctly, it should look like both 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.

The Problem: Long-Latency Input

Once you have the project working, you are done writing code, but there is one more step to perform, so that you understand a big problem that operating systems face. Comment out the code in ***z_shell.c*** that loops back to the top of the ***while()*** loop without doing any input. What happens? The reason ***why*** it happens can be seen in ***uart.c*** … when you ask the kernel to READ the UART on your behalf, it ultimately calls ***uart_recv()*** — which **blocks**. A short section of the ***uart.c*** module is shown below:

```
//------------------------------------------------------------------------
unsigned int uart_recv ( void )
{
    while(1)
    {
        if(GET32(AUX_MU_LSR_REG)&0x01) break;
    }
    return(GET32(AUX_MU_IO_REG)&0xFF);
}
//------------------------------------------------------------------------
unsigned int uart_recvcheck ( void )
{
    if(GET32(AUX_MU_LSR_REG)&0x01) return(1);
    return(0);
}
//------------------------------------------------------------------------
unsigned int uart_sendcheck ( void )
{
    if(GET32(AUX_MU_LSR_REG)&0x20) return(1);
    return(0);
}
//------------------------------------------------------------------------
void uart_send ( unsigned int c )
{
    while(1)
    {
        if(GET32(AUX_MU_LSR_REG)&0x20) break;
    }
    PUT32(AUX_MU_IO_REG,c);
}
```

The first thing that ***uart_send()*** and ***uart_recv()*** do is hang indefinitely.

The IRQ interrupt is set up not to interrupt the kernel in SVC mode, so the entire time this code is blocking, the timer interrupt is being ignored.

Next project, we will address that. Start thinking now about how **you** would address it.

Build It, Load It, Run It

Once you have it working, show us.