**Project 5: Non-Blocking I/O (4%)**

**ENEE 447: Operating Systems — Spring 2019**

**Assigned:** Tuesday, Mar 12; **Due:** Sunday, Mar 31

**Purpose**

In this project, your main task is to implement non-blocking I/O for just two scenarios: reading and writing the DEV_WORD devices, which includes the LED (write-only) and the Console (read/write). You also need to implement the C-language scheduler, which is now more complex than in the previous project, as it handles a set of threads, not just two, and it maintains a TCB for each thread that is running. Much has been improved since the last project: you will work with a slightly more fleshed-out shell; you will invoke threads from the shell; and you can get a list of running processes. The thread creation and invocation facilities are more generalized and therefore support a more extensive system. Thus, you are being exposed to techniques that better exemplify the way things are normally done, and you will build the hardest parts (95% of the work has been done for you: the easy part that nonetheless would take quite a while to implement).

**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 are 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 added to that the preemptive context switch and a rudimentary scheduler: the scheduler simply switches processes every time quantum, involved by a timer interrupt. Project 5 makes the I/O non-blocking so that the kernel can better respond to multiple sources of input “simultaneously” arriving from different sources.

**Interrupt Handlers and Thread Context**

Firstly, we have provided a simplified mechanism for saving and restoring state. In the *boot.s* code, you will find the following assembly-code routines:

```assembly
save_lr_irq : .word 0

// courtesy of Prof Vince Weaver, U Maine
irq_handler:

ldr sp, tcb_address_runningthread
stmia sp, {r0-r1}^ @ Save all user registers r0-lr
ldr sp, {r0-r1}^ @ (the ^ means user registers)
str lr, [sp, #60] @ store saved PC on stack
str lr, save_lr_irq @ save the SVC lr
mrs lr, SPSP ^ @ load SPSP (assume ip not a swi arg)
str lr, [sp, #64] @ store on stack
ldr lr, save_lr_irq @ save the SVC lr

@ Call the C version of the handler
mov sp, #SVCSTACK0
b clear_timer_interrupt
b periodic_timer
b set_timer

ldr sp, tcb_address_runningthread
ldr r0, [sp, #64] @ pop saved CPSR
msr SPSP_cxsr, 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-r1}^ @ return from exception
```

© Copyright 2016, 2019 Bruce Jacob. All Rights Reserved
These two routines are the only interrupt handlers, and the IRQ will neither interrupt itself (unless we set the timer too fast), nor will it interrupt the SVC mode. On a single core, the SVC mode will not interrupt itself. Therefore, none of this code will conflict with itself. What that means is that you won't have to worry about the `tcb_address_runningthread` value changing out from underneath you, and potential weirdness of that sort.

The code begins by performing the following save-register functions:

- The address of the TCB for the currently running thread 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:

```c
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 it. This is stored in the following structure, for which there is one for every process in the system:
struct tcb {
    LL_PTRS;
    char name[NAMESIZE];
    long tid;
    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. Thread 0 is the “NULL thread,” and the rest of the thread numbers can be assigned to anything.

Note that the name, ID, and stack are static values: they do 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: user thread stacks begin at 0x20000, 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 TCB for the currently running thread 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.

Here's the weirdness. The ARM32 architecture often used the following snippet to return from IRQ and FIQ exceptions:

    subs pc, lr, #4 @ return from exception

It is clearly a hack. It is inelegant. It is how IRQ and FIQ interrupts generally work. However—and here is where the weirdness comes in—it is NOT how ARM's SVC trap handlers generally work. If you try to subtract 4 from the SVC handler's link register on return from exception, you will get a stalled processor. But nonetheless, we would like to structure our kernel so that at both entry points into the kernel, a process is saved so that it can potentially be swapped out as a result of calling the interrupt. Thus, the process needs to be saved and restored identically, no matter which interrupt handler does the saving/restoring.

Thus, what gets stored by the register-save protocol is a number that will work with the “-4” on the return-from-exception at the end of both interrupt handlers (we do it in the SVC handler even though we don't need to, just to ensure symmetry). This is expected for the IRQ, but it is not expected for the SVC trap handler. Thus, you see that, when the SVC handler stores the return value, it first adds 4 to it. Yes, this is inelegant. Yes, this is stupid. Welcome to interfacing with hardware.

The upside: you need not write any assembly code for this project, unless you want to.
**General Kernel Structure**

The idea is that on either a trap (SVC handler) or a periodic timer interrupt (IRQ handler), a process can get swapped out. On a trap to the SVC handler, a process can get swapped out if it is a long-latency I/O operation (for this project, a SYSCALL_RD_WORD or SYSCALL_WR_WORD). On an invocation of the IRQ handler (due to a periodic timer interrupt), we will always swap processes, if there is another to swap to.

The threads.c module maintains three lists of processes:

- The **free list** (contains TCB structures not being used)
- The **runq** (TCB structures for threads that can run but are not the currently active thread)
- The **sleepq** (TCB structures for threads that are waiting on I/O to finish)

The `create_thread()` routine sets up a thread and moves its TCB structure onto the **runq**, where it is likely to begin executing (become the currently active thread), starting with the next periodic timer interrupt.

The `scheduler()` function takes as an input one of three values:

- **THREAD_INIT** — indicates that there is no active thread to remove; we simply need to select someone from the **runq** and initialize the TCB pointer, as well as stack and start addresses that are going to be used by the *boot.s* assembly code (see where core0 starts up the first user thread).
- **THREAD_RUN** — indicates that the currently active thread should be preemptively swapped out for another thread, provided that there is another to run. If there is no other thread on the **runq**, then the currently active thread should simply continue to execute.
- **THREAD_SLEEP** — indicates that the currently active thread needs to be put onto the **sleepq**, because it is waiting on long-latency I/O operations to finish. If the **runq** is empty, the NULL thread is run as a default; that is the only reason it exists.

**Handling I/O Operations in a Non-Blocking Manner**

As described in class, when your trap handler sees that the system call is SYSCALL_RD_WORD or SYSCALL_WR_WORD, you invoke the timeout queue. The mechanism for using the timeout queue has changed slightly. First, the `create()` routine looks like this:

```c
void create_timeoutq_event(int timeout, int repeat, int max, pfv_t function, namenum_t data);
```

The timeout is the time until the event fires; the repeat value, if non-zero, indicates when after that it should re-fire. The max value indicates the maximum number of repeats, and the function/data pair acts like before.

Another difference, as you will see if you look in callout.c, is that the input to the user-specified timeout function is the data structure itself:

```c
ep = (struct event *)LL_DETACH(timeoutq, ep);
ep->go(ep);
if (ep->repeat_interval > 0 && ep->max_repeats-- > 0) {
    ep->timeout = ep->repeat_interval;
    insert_event(ep);
} else {
    LL_PUSH(freelist, ep);
}
```

This allows your user-defined timeout function to determine for itself if it wants to continue repeating. If it wants to continue, it can manually set the struct's `repeat` and `max` values to appropriate non-zero values, and if it wants to terminate, it sets them to zero. This is what you will implement do in the function
**do_dev_word** in the *io.c* module. Other facilities that this code will use are the new read and write check() functions in the device structure.

You will notice that the device table has been redefined slightly, with the addition of these two new structure members:

```c
struct GdevG {
    char devname[8];
    int devtype;
    pfv_t init;
    pf_t read;
    pf_t write;
    pf_t rcheck;
    pf_t wcheck;
};
```

There are now the rcheck and wcheck members, which are used to check and see if the device is ready to read or write. These functions have been provided for you, at least for the UART device (the main device for which you will be using them in this project). They are in the uart.c module:

```c
int uart_recvcheck ( void )
{
    if (GET32 (AUX_MU_LSR_REG)&0x01) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x01) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x01) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x01) return(1);
    return(0);
}

int uart_sendcheck ( void )
{
    if (GET32 (AUX_MU_LSR_REG)&0x20) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x20) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x20) return(1);
    if (GET32 (AUX_MU_LSR_REG)&0x20) return(1);
    return(0);
}
```

The repetitive checks is a hack to improve responsiveness. Recall that this replaces a while() loop, which suggests that sometimes it may take a while to get the UART’s attention. If we only checked once, then we would have to wait potentially 1/100 to 1/10 of a second to try again … might as well give it a bunch of tries (kind of like replacing the while() loop with a short for() loop that can be retried periodically).

Your code, which will implement the *do_dev_word()* function, will make use of these functions, testing to see if the device is ready to read or write — if so, perform the function, but if not, reschedule the task to try again in the near future. Whenever this function wakes up, it checks to see if the word-granularity I/O device is ready. If the device is ready, it performs the I/O function and sets the event structure not to repeat; however, if the device is not ready, the function sets the repeat values in the event structure appropriately to repeat again in the future. For WRITE system calls, the data to output to the device will be in the data value of the io structure. For READ system calls, you need to figure out how to get the value back to the user process. Hint: use the TCB structure!

**Example Interaction with the Shell**

The following shows how the interaction with the shell might go, indicating what sort of things might come next in terms of thread invocation, etc.

```
[00:02:023] ...
[00:02:025] System is booting, kernel cpuid = 00000000
[00:02:037] create_thread:
[00:02:039] NULL threadIdx 00000000
[00:02:043] stack = 00021000
[00:02:046] start = 00000044
```
At this point, there are two threads that have been created: the NULL thread and the shell. However, when I do a PS, the NULL thread doesn’t show up, because, in my implementation, it isn’t kept on the runq. You can do whatever is easiest or most logical for you.

At this point, we can start up do_blinker by invoking it from the shell. Looking at the kernel7.list file, we find that do_blinker is located at address 0x00002194. The inputs for RUN are a short name (1–3 chars) & address. Thus we have the following:

```
P:SYs
D:SYs
P:SYs
```
At this point, if we do a PS, we should see both threads:

Please enter a command.
> PS
CMD_PS
[c0] 00:58.115  PS: Active processes ...
[c0] 00:58.118  Dumping TCB for thread 00000001
[c0] 00:58.122  shell 00000001
[c0] 00:58.125  stack 00022000
[c0] 00:58.128  tcb @ 0000DC1C
[c0] 00:58.131  r0  00000001
[c0] 00:58.133  r1  0000000A
[c0] 00:58.136  r2  00005350
[c0] 00:58.139  r3  00005350
[c0] 00:58.142  r4  00021FD4
[c0] 00:58.145  r5  00000000
[c0] 00:58.147  r6  00021FD4
[c0] 00:58.150  r7  00000009
[c0] 00:58.151  r8  54495845
[c0] 00:58.156  r9  454D4954
[c0] 00:58.159  r10 00002C84
[c0] 00:58.161  r11 00000000
[c0] 00:58.164  r12 00021FC0
[c0] 00:58.167  sp  00021FA4
[c0] 00:58.170  lr  000025D0
[c0] 00:58.171  pc  0001A4A4
[c0] 00:58.175  spsr 00000150
[c0] 00:58.178  Dumping TCB for thread 00000002
[c0] 00:58.183  BLK 00000002
[c0] 00:58.185  stack 00023000
[c0] 00:58.188  tcb @ 0000DC80
[c0] 00:58.191  r0  00000003
[c0] 00:58.193  r1  00022FE0
[c0] 00:58.196  r2  00000008
[c0] 00:58.199  r3  00000000
[c0] 00:58.202  r4  00000001
[c0] 00:58.205  r5  000AA660
[c0] 00:58.208  r6  037EAA64
[c0] 00:58.210  r7  00000004
[c0] 00:58.213  r8  00000000
[c0] 00:58.216  r9  00000000
[c0] 00:58.219  r10 00000000
[c0] 00:58.222  r11 00000000
[c0] 00:58.224  r12 00000000
[c0] 00:58.227  sp  00022FDC
[c0] 00:58.230  lr  000021F0
[c0] 00:58.231  pc  00019E00
[c0] 00:58.236  spsr 80000150

Please enter a command.
> TIME
CMD_TIME = [00000000 047DD3DD]

Please enter a command.
>

This should make it relatively clear that, as far as the operating system goes, there is little difference between two different threads running in two different processes and two different threads running in the same process. Multithreading via spawning at the function level would be a trivial extension. Perhaps in an upcoming project …

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

Once you have it working, show us.