This page last updated: April 12, 2016.

Context Switching and Timer Interrupts

Spring Term, 2016

In these exercises, we'll be modifying a version of the `example-idt` demo, which is part of the code in the `prot.tar.gz` package on D2L. If you are on a LinuxLab machine, you should also be able to grab a copy of that file from the LLP stash at `/stash/cs410_LLP/prot.tar.gz`. Either way, once you have a copy of `prot.tar.gz` in your llp directory (i.e., the one that you share with the virtual machine), you should be able to unpack it and prepare for the exercises described on this page using the following commands:

```
tar xvzf prot.tar.gz
make
cp -r example-idt switching
cd switching
make run
```

Of course, at this point in the process, you should see exactly the same behavior as if you were running the original `example-idt` example, as demonstrated in class. To help confirm that you are working in the right directory in later steps, it would probably be a good idea to edit the `grub.cfg` file so that this program shows up in the GRUB boot menu using a different description: `"Context Switching Demo"` is probably a good choice ...

Ok, let's get to work!

Step 1: "hlt is a privileged instruction"

In this first step, we'll demonstrate that `hlt` is a *privileged* instruction, which means that it cannot be executed in user mode. (Or, at least, you can try to execute `hlt` in protected mode, but it won't actually halt the system!)

Start by adding a new function called `yield` to the user mode program. Implement this function in assembly code (in `userlib.s`), using a simple body that attempts to halt the machine by using a `hlt` instruction. For safety, you might also add a `jmp` instruction so that your program will jump back to retry the `hlt` if, for some reason, the program ever gets to execute any following code. You should also add a prototype for this function (`extern void yield(void);`) at the top of `user.c`, and insert a call to this function as the last statement in the body of the `for` loop in `cmain()`.

Try `make run`, and explain the results ...

This might be a good time to introduce the QEMU monitor. While you are running a program in QEMU, you can hit Ctrl-Alt-Shift-2 to switch to QEMU's monitor window. (You can switch back to the main QEMU display using Ctrl-Alt-Shift-1.) This allows you to type in simple commands to control and inspect the state of the virtual machine. For example:

- `"info cpus"` lists current CPU status
- `"info registers"` displays the values of all CPU registers (Use Ctrl up arrow and Ctrl down arrow to scroll)
- `"print $eax"` displays the value of the `eax` register. More generally, you can use `"print expr"`. (Note that `"print"` can be abbreviated as `"p"`.)
- `"xp /fmt addr"` displays the contents of (physical) memory at the specified address (which can be a specific number or a register name, such as `$eip`). The `fmt` parameter determines how the data is displayed. For example, `/5i` will display the next five instructions.

Step 2: "Privileged instructions can be executed in Ring 0"

We'll start by arranging for the user mode program to ask the kernel to halt the system, instead of trying to execute the halt instruction itself. For this, you should modify your implementation of `yield` to use `int $129` instead of `hlt`. Don't forget to add a `ret` instruction after the `int $129` so that your program will return to the code where `yield()` was called after the interrupt has been handled.

Now we'll need to make sure that the kernel can respond to this interrupt. In the `kernel` directory, specifically in `init.s`, create an interrupt handler stub for `yield` that will execute the `hlt` instruction. In addition, you'll need to make sure that the IDT will contain an appropriate entry. To a large degree, you can model your implementation on what was done for `"syscall"` ...

Now what happens? (Make sure, in particular that there is no message indicating that an exception has occurred.)

Step 3: "A (slightly) more interesting system call"

Now let's provide a (slightly) more interesting implementation of yield by making the interrupt handler jump to a label `"yieldimp"`. Again, you can use `"syscall"` as a model. We'll write the `yieldimp` function in C as part of the code for `kernel.c`:

```
void yieldimp() {
    printf("Yielding ...");
    switchToUser(&user);
}
```

The first line in this implementation allows us to see that the kernel is running (the output appears on the left hand side of the screen). The second line sends us back in to the user mode program. (The kernel never actually returns from `yieldimp()`!)

Verify that this works in the way you would expected ... (and if you're not sure what to expect, then ask for some help and try to talk it through ...)

Step 4: "Multiple user processes"

In the user directory, create a new version of the `user` program called `user2`. You'll need to make copies of the `user.c` source file and the `user.ld` script. I would suggest modifying the two `.c` files so that you can tell which one is running by looking at the output messages that they produce. As for the loader

script, we'll want to load both user programs in memory at the same time, so they can't both start at the same address. I suggest configuring `user2` to load at address `0x500000`, corresponding to 5MB. Edit the `Makefile` so that you can build both the `user` and `user2` executables by using the `make` command. (It's ok to copy, paste, and then edit the relevant parts of the rules for building `user` and `user.o`.) While we're editing Makefiles, be sure to add `user/user2` to the `mimgmake` command in the main `Makefile` (i.e., the `Makefile` in your `switching` folder). (Also, in case you're not already aware of this detail, it might be worth pointing out that all of the command lines in a `Makefile` must begin with at least one tab character.)

We'll also take this opportunity to clean up the display a little. I suggest modifying the start-up code for the two user programs so that they set the text windows shown below:

For `user`:

```
  setWindow( 1, 11, 47, 32);   // user1 process on upper right hand side
```

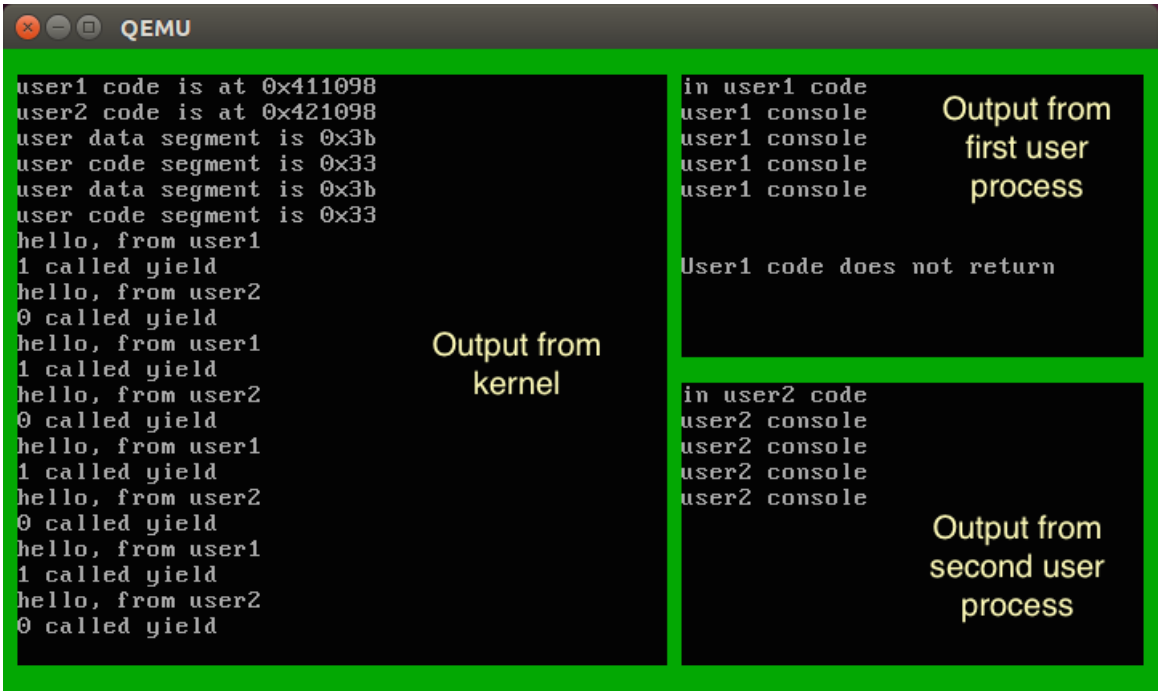
For `user2`:

```
  setWindow(13, 11, 47, 32);   // user2 process on lower right hand side
```

In addition, you can modify the code that configures the display at the start of the main `kernel()` function in `kernel.c`:

```
  setAttr(0x2e);               // Create a PSU Green border
  cls();
  setWindow(1, 23, 1, 45);     // kernel on left hand side
  setAttr(7);
  cls();
  printf("Protected kernel has booted!\n");
```

The following picture shows the corresponding regions on the screen for each of these three programs. (But note that **the output that you get right now won't actually look like this in all details until we get to the end of Step 6** and have managed to get both user programs running.)



Verify that your program runs correctly. Look at the memory map and header information that is displayed to confirm that both user programs have been loaded at the expected addresses, and that the entry points that are displayed make sense.

You can (and should) experiment with modifying the kernel so that it will run the second user program instead of the first. Make sure that you are using the header information from the `BootData` information rather than hardwiring a specific address in to your code: those addresses might change if we do more work on the user programs later on ...

At this point in time, however, the `kernel` code will only be able to run one of the two user programs. Wouldn't it be nice if we could run both? At the same time?

Step 5: "Multiple contexts"

Add some new global variables to `kernel.c` so that we can store two `Context` structures, one for each of the two user programs:

```
struct Context user[2];
struct Context* current;
```

Replace the code that initialized the original `user` context with code that will initialize the two separate structures in the new user array, one for each of our two user mode programs. Note that you can use the expressions `(user+0)` and `(user+1)` to get the addresses of the two context structures. Of course, you can simplify the first of those to just `user`, if you prefer.

Update the rest of the program so that any remaining references to `user` are replaced with appropriate uses of `"current"` instead. Note that `"current"` is a pointer, so you won't have to take its address using `&`, but you will need to use the `->` operator instead of the `.` operator to access its fields. Check that you get the expected results when you initialize `current`, either to `(user+0)`, or to `(user+1)`.

You might be tempted to add context switching at this point, but make sure you've completed the steps above and that everything is working correctly before you move on ...

Step 6: "Context Switching"

Update your `yieldimp()` implementation so that it changes the value of `current` each time it is called to alternate between the two user contexts. Something like the following should do the trick:

```
current = (current==user) ? (user+1) : user;
```

Test to see if you get the desired behavior. Hint: it will probably be pretty close, but not exactly perfect ...

Step 7: "May I interrupt you (on a regular basis)?"

Now we're going to use the hardware to force context switching at regular intervals, even if the user programs don't `yield()`. To get started, you can prevent `yield` from having any effect at all by commenting out the line we just put in to change contexts. Now that's gone, we'll know that any context

switching that occurs will be the result of whatever new code we add. If you really want to be sure, you can delete the calls to `yield()` altogether in both user programs. (If you keep the calls to `yield()` in your user programs and don't comment out the context switching line in the `yieldimp()` function, then you might notice some significant, negative impacts on performance; by all means feel free to come back later on to experiment with this and see if you can figure out why this is happening ...)

Unfortunately, however, we are going to need a fair bit of new code to complete this step:

- Put a copy of the `initpic.s` assembly code (available to download via that link) in your kernel directory. Modify the `Makefile` so that `initpic.s` is compiled (strictly speaking, assembled) and linked in with the `kernel` executable. (You can follow the pattern that is used for `init.s`.) You don't need to read the code in `initpic.s`, but if you do, then you'll discover that its purpose is to implement a function that could be declared as:

  ```
  extern void initPIC();
  ```

  in C, that initializes a standard hardware component on the PC called the PIC, or *Programmable Interrupt Controller*. [Hint: It probably wouldn't hurt to add the above function prototype somewhere near the top of `kernel.c`.]

- Put a copy of the `"hardware.h"` file (again, available to download via the link) in your kernel directory. This file provides implementations of some functions for interacting with the PIC, as well as some code for configuring a second hardware device, the PIT, or *Programmable Interval Timer* that is also part of standard PC systems. Again, you're welcome to peruse the code, but not expected to understand all of the details. But one thing you will need to do is to `#include "hardware.h"` at the top of your `kernel.c` source file.

- Add a new interrupt handler in `init.s` with the following code:

  ```
  timerHandler:
     sub     $4, %esp
     push    %gs
     push    %fs
     push    %es
     push    %ds
     pusha
     leal    stack, %esp
     jmp     timerInterrupt
  ```

  You will also need to add an entry for this handler in the IDT; be sure to use slot 32 for this because that is the slot that the timer hardware is configured to use. Note, however, that you won't need to specify a `dpl` parameter this time; the default value of `0` will work just fine.

- Next, add an implementation of the `timerInterrupt` function (which appeared as the target of the `jmp` instruction in the previous chunk of assembly code) in your `kernel.c` file. Something along the following lines should do the trick:

  ```
  static void tick() {
      static unsigned ticks = 0;
      ticks++;
      if ((ticks&15)==0) {
          current = (current==user) ? (user+1) : user;
      }
  }
  ```

```
void timerInterrupt() {
  maskAckIRQ(TIMERIRQ);
  enableIRQ(TIMERIRQ);
  tick();
  switchToUser(current);
}
```

I've separated this code in to two sections so that you can focus on the code in the `tick()` function and not worry too much about the details of interrupt handling in the main `timerInterrupt()` function. We'll learn more about how the latter function works in the coming weeks.

- The final step is to enable interrupts. (You might want to check that all of the code above is compiling and running as expected before you do this.) There are two parts to this. First, we need to make sure that user processes can be interrupted. We do this by making sure that the `IF` bit is set in the `eflags` register for each user program. This can be done at the time we initialize the associated contexts: look at the definition of `INIT_USER_FLAGS` in `context.h`, and add in the `1 << IF_SHIFT` component (or just use the `#define` for `INIT_USER_FLAGS` that has been commented out)!

  Last, but not least, add the following two lines, in the `kernel()` function, right before the point where you start running a user mode program for the first time:

  ```
  initPIC();
  startTimer();
  ```

With all these changes in place, your kernel will configure the PIC and PIT to interrupt the system 100 times a second, executing the `tick()` function each time that happens. And once every 16 calls, it will automatically context switch between the two user programs, ensuring that both run to completion! (We don't need to context switch as frequently as 100 times a second—that might create unnecessary overhead. And so we choose to switch at 1/16 of that frequency instead.)

For demonstration purposes, you might like to increase the number of loop iterations in the two user programs (and have the loop bodies print out the current value of `i` at each step so that you can see it increasing). I'd suggest trying 300 iterations in `user.c` and 600 in `user2.c` so that you can see both running together, as well as what happens when one stops. (Adjust these numbers up or down, depending on how fast your VM seems to be running; so long as there is a noticeable gap between the times that the two user processes finish doing "useful work", the exact numbers are not important.)

Step 8: "Are you sure we're really protected?"

Although we are running in "protected" mode, there is, in fact, no real protection between the code in the different programs. To illustrate this, we'll modify `user.c` and `user2.c` to see how one can "interfere" with the other.

First, in `user.c`, add the definition of a variable:

```
volatile unsigned flag = 0;
```

and then insert the following lines immediately after the main loop:

```
printf("My flag is at 0x%x\n", &flag);
while (flag==0) {
```

```
    /* do nothing */
}
printf("Somebody set my flag to %d!\n", flag);
```

Clearly, `flag` is initialized to zero, and there are no other assignments to `flag` in the code for `user.c`, so we would not expect the while loop seen here to terminate. Try building and running the program to confirm that the `"Somebody set my flag ..."` message does not appear.

Next, pay careful attention to the address that the user program displays in the `"My flag is at ADDR"` message, and then add the following lines after the body of the loop in `user2.c`:

```
unsigned* flagAddr = (unsigned*)ADDR;
printf("flagAddr = 0x%x\n", flagAddr);
*flagAddr = 1234;
```

where `ADDR` is replaced with the previously observed flag address. Make and run to see what happens now. Oh dear. And what is special about addresses that are part of the `user` program? In other words, is there anything that might prevent `user2` from interfering with (or even rewriting) the `kernel` code? Oh dear indeed. We'll have to work on this some more ... :-)

[Of course, this shouldn't really be a surprise: our kernel and user programs have been happily writing in to the same video RAM ever since we got started with these exercises; it's only the careful choice of parameters to `setWindow()` that have prevented them from interfering with one another before now ...]

But to have just a little more fun before we get there ...

Step 9: "What time is it?"

As a stretch goal, you might like to try adding a clock that provides a continuous display of the current time (i.e., the number of `ticks` since the kernel started) on the top line of the video RAM. The fine details of how you do this are up to you. To my mind, however, an eight digit hexadecimal display seemed like a good fit, not to mention an opportunity to reuse some code that I had already written ...
