ECE531: Advanced Operating Systems – Homework 5

Syscalls and Userspace

Due: Friday, 6 October 2023, 5:00pm

This homework involves setting up a system call handler and moving code to execute in userspace.

1. **Download the homework code template**

   - Download the code from:
     `http://web.eece.maine.edu/~vweaver/classes/ece531/ece531_hw5_code.tar.gz`
   - Uncompress the code. On Linux or Mac you can just
     `tar -xzvf ece531_hw5_code.tar.gz`

2. **Convert the shell to a user application (2pt)**

   - You can use either the provided `shell.c` or else the one you wrote yourself for HW#4.
   - The `shell.c` file now lives in the `user` subdirectory, so make sure you are editing it there.
   - We will need to convert it so that it no longer makes function calls into the kernel directly, but instead uses system calls. To make this easier I've provided a "vlibc" library that implements a few simple system calls.
     (a) First add the `#include "vlibc.h"` header. You can also remove any headers that reference the kernel directly.
     (b) Change all calls to `printk()` to `printf()`
     (c) Change `uart_getc()` and `uart_putc()` to the vlibc routines `getchar()` and `putchar()`
     (d) You might want to take a look at the `vlibc.c` and `syscalls.h` code to see how userspace makes a system call.
     (e) Edit `kernel_main.c` and remove the `#if 0` and `#endif` at the end to activate the calls to setup the user stack and switch to userspace. Be sure you understand why those assembly calls are needed!
     (f) Boot your system, if all went well your shell should be operating like before, but now in userspace!

3. **Add a time system call (3pt)**

   - We are going to add the `TIME` system call, `time()`. On Linux `time()` takes one argument, a pointer to where in memory you want to store the time value (number of seconds since the epoch, which for Linux is Jan 1 1970). It also returns the time value as the result. If the argument is NULL it still returns the time but doesn't write anything to memory.

     Our Pi doesn't have a real-time count, so we are instead going to return the number of seconds since our OS was booted.
   - First, modify `interrupt_handler()` in `interrupts.c` so that it increments the (already provided) variable `tick_counter` each time a timer interrupt happens. (`tick_counter` is defined for you in `timer.c`).
   - Next, edit `syscalls.h` and uncomment the `SYSCALL_TIME` define. We are using numbers that match the Linux syscall numbers.

- Now edit `syscalls.c`. In the swi_handler() code note that r7 (the syscall number according to the kernel EABI) is used in a switch statement to pick what code to run. Add a case for `SYSCALL_TIME`.
- Modify the code there so that it puts the current value of `tick_counter` into the address pointed to by r0. There are various ways to do this. To avoid breaking your head with too much C pointer math you might try to break things into multiple steps, like this:

```
int *int_ptr;        // int_ptr can point to an int
int_ptr=(int *)r0;   // int_ptr now points to address
                     // passed in r0
```

  and then use `int_ptr` to write the `tick_counter` value to the memory pointed to by r0.

  Remember, `int_ptr` is an address but `*int_ptr` is the value (the dereferenced pointer).

  Be sure to check to make sure the pointer is not NULL before writing it.

  Also set the "result" value so that the time value gets passed back in the r0 result.
- Add code to your command parser in `shell.c` so that it recognizes the `time` command. When you give the time command, it should print the number of seconds since boot. It can get this by calling the time syscall you just implemented. Something like the following (that's syscall followed by the number 1):

```
syscall1(SYSCALL_TIME,(long)&ticks);
```

  where `ticks` is a 32-bit variable that you create.
- Test your new syscall and make sure it works.

4. **Something Cool (1pt)**

   Be sure you document in your README what your something cool does

   - Easy: have your parser handle commands other than just "print" (i.e. port your HW#4 something cool to this one).
   - Hard: update the vlibc library so it has a wrapper for time() much like it does for read() and write() and then have your shell call that rather than calling syscall1() directly.
   - Harder: add another system call of your own design. One idea might be a `SYSCALL_BLINK` that disables or enables the LED blinking.

5. **Answer the following questions (4pt)**

   Put your answers to these questions in the README file.

   (a) I had to modify the provided codes so it uses a non-blocking read for the `read()` syscall used by `getchar()`. Why is this necessary? (hint, what would happen if we called `read()` and it sat waiting in the SWI handler for keys to be pressed?)

   (b) What is one reason to run some code in userspace rather than running everything in supervisor mode?

   (c) When running in user mode you cannot change the mode bits in the CPSR register. What are the two ways remaining that can cause your Pi to switch from userspace back to supervisor mode?

   (d) What is an ABI and why do they exist?

(e) When the time() syscall is called, our kernel writes the time to whatever pointer the user passes in. Are there any security implications to this? (i.e. how could a hacker use this to cause trouble)

(f) Look up a Linux system call, that's not read/write/open/close. Give a brief, few line discussion of what it does. The manual pages can be useful for finding out this information.

6. **Submit your work**

- Run `make submit` in your code directory and it should make a file called `hw5_submit.tar.gz`. E-mail that file to me as well as the document with the answers to the questions.