CS/COE 1550 – Introduction to Operating Systems

Project 1: Syscalls[1]

Upload the files sys.c, syscall_table.S, unitstd.h, and sem.h into GradeScope.

Due Date: Friday, September 27, 2019 @11:59pm
Late Due Date: Sunday, September 29, 2019 @11:59pm with 10% reduction per late day

Table of Contents

[1] Based upon Project 2 of Dr. Misurda's CS 1550 course.

Project Overview

Anytime we share data between two or more processes or threads, we run the risk of having a race condition where our data could become corrupted. In order to avoid these situations, we have discussed various mechanisms to ensure that one program's critical regions are guarded from another's.

Project Details

Syscalls for Synchronization

We need to create a semaphore data type and the two operations we described in class, down() and up(). To encapsulate the semaphore, we'll make a simple struct that contains an integer value and a queue of processes:

```
struct cs1550_sem
{
   int value;
   //Some queue of your devising
};
```

We will then make two new system calls that each has the following signatures:

```
asmlinkage long sys_cs1550_down(struct cs1550_sem *sem)

asmlinkage long sys_cs1550_up(struct cs1550_sem *sem)
```

to operate on our semaphores.

Sleeping

As part of your down() operation, there is a potential for the current process to sleep. In Linux, we can do that as part of a two-step process.

1) Mark the task as not ready (but can be awoken by signals):
   `set_current_state(TASK_INTERRUPTIBLE);`

2) Invoke the scheduler to pick a ready task:
   `schedule();`

Waking Up

As part of up(), you potentially need to wake up a sleeping process. You can do this via:

```
wake_up_process(sleeping_task);
```

Where `sleeping_task` is a `struct task_struct` that represents a process put to sleep in your down(). You can get the current process's `task_struct` by accessing the global variable `current`. You may need to save these someplace.

Atomicity

We need to implement our semaphores as part of the kernel because we need to do our increment or decrement and the following check on it **atomically**. In class we said that we'd disable interrupts to achieve this. In Linux, this is no longer the preferred way of doing in kernel synchronization due to the fact that we might be running on a multicore or multiprocessor machine. Instead, we'll use something somewhat surprising: *spin locks*.

We can create a spinlock with a provided macro:

```
DEFINE_SPINLOCK(sem_lock);
```

We can then surround our critical regions with the following:

```
spin_lock(&sem_lock);

spin_unlock(&sem_lock);
```

For each, feel free to draw upon the text and handouts for this course as well as 449.

Adding a New Syscall

To add a new syscall to the Linux kernel, there are three main files that need to be modified:

1. `linux-2.6.23.1/kernel/sys.c`

This file contains the actual implementation of the system calls.

2. `linux-2.6.23.1/arch/i386/kernel/syscall_table.S`

This file declares the number that corresponds to the syscalls

3. `linux-2.6.23.1/include/asm/unistd.h`

This file exposes the syscall number to C programs which wish to use it.

Testing the syscalls

As you implement your syscalls, you are also going to want to test them via a user-land program. The first thing we need is a way to use our new syscalls. We do this by using the `syscall()` function. The syscall function takes as its first parameter the number that represents which system call we would like to make. The remainder of the parameters are passed as the parameters to our syscall function. We

have the syscall numbers exported as `#defines` of the form `__NR_syscall` via our `unistd.h` file that we modified when we added our syscalls.

We can write wrapper functions or macros to make the syscalls appear more natural in a C program. For example, you could write:

```
void down(struct cs1550_sem *sem) {
       syscall(__NR_cs1550_down, sem);
}
```

Hints

Setting up the Kernel Source (To do in recitation)

1. Copy the linux-2.6.23.1.tar.bz file to your local space under `/u/OSLab/username`

   ```
   cp /u/OSLab/original/linux-2.6.23.1.tar.bz2 .
   ```

2. Extract

   ```
   tar xfj linux-2.6.23.1.tar.bz2
   ```

3. Change into linux-2.6.23.1/ directory

   ```
   cd linux-2.6.23.1
   ```

4. Copy the .config file

   ```
   cp /u/OSLab/original/.config .
   ```

5. Build

   ```
   make ARCH=i386 bzImage
   ```

You should only need to do this once, however redoing step 2 will undo any changes you've made and give you a fresh copy of the kernel should things go horribly awry.

Rebuilding the Kernel

To build any changes you made, from the `linux-2.6.23.1/` directory, simply:

```
make ARCH=i386 bzImage
```

QEMU Version

We will be using an x86-based version of Linux and QEMU for this project. The disk image and a copy of QEMU for windows are available on CourseWeb (qemu.zip). For Mac users, you can download an

older but GUI-based application (Q.app) available on CourseWeb as well. Point it at the tty.qcow2 disk image in the above zip.

The username and password are both the word **root**.

For Linux users (and Mac users wanting to use the homebrew version), you can find on CourseWeb a test version of the disk image and a start.sh script to run it (qemu-test.zip). It should be identical to the above version in terms of functionality, but actually boot with a recent version of QEMU. IF THE ORIGINAL WORKS FOR YOU, DON'T BOTHER WITH THIS ONE.

On Mac OS X, if you don't have Homebrew, open a terminal and type:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Go through the install steps. When done, install qemu by typing:

```
brew install qemu
```

That will install qemu. Now you can run `start.sh` from the terminal **in the unzipped folder** to launch qemu.

On Linux, using your appropriate package manager, install qemu-system-arm, likely part of your distro's qemu package.

Then run `start.sh` **in the unzipped folder** to launch qemu.

Copying the Files to QEMU

From QEMU, you will need to download two files from the new kernel that you just built. The kernel itself is a file named `bzImage` that lives in the directory `linux-2.6.23.1/arch/i386/boot/`. There is also a supporting file called `System.map` in the `linux-2.6.23.1/` directory that tells the system how to find the system calls.

Use `scp` to download the kernel to a home directory (`/root/` if root):

```
scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/USERNAME/linux-2.6.23.1/arch/i386/boot/bzImage .

scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/USERNAME/linux-2.6.23.1/System.map .
```

Installing the Rebuilt Kernel in QEMU

As root (either by logging in or via `su`):

```
cp bzImage /boot/bzImage-devel

cp System.map /boot/System.map-devel
```

and respond 'y' to the prompts to overwrite. Please note that we are replacing the -devel files, the others are the original unmodified kernel so that if your kernel fails to boot for some reason, you will always have a clean version to boot QEMU.

You need to update the bootloader when the kernel changes. To do this (do it every time you install a new kernel if you like) as root type:

```
lilo
```

lilo stands for LInux Loader, and is responsible for the menu that allows you to choose which version of the kernel to boot into.

Booting into the Modified Kernel

As root, you simply can use the `reboot` command to cause the system to restart. When LILO starts (the red menu) make sure to use the arrow keys to select the linux(devel) option and hit enter.

Building and Running test programs

If we try to build your test program using `gcc`, the `<linux/unistd.h>` file that will be preprocessed in will be the one of the kernel version that `thoth.cs.pitt.edu` is running and we will get an undefined symbol error. This is because the default `unistd.h` is not the one that we changed. What instead needs to be done is that we need to tell `gcc` to look for the new include files with the `-I` option:

```
gcc -m32 -o trafficsim -I /u/OSLab/USERNAME/linux-2.6.23.1/include/ trafficsim.c
```

We cannot run our `test` program on `thoth.cs.pitt.edu` because its kernel does not have the new syscalls in it. However, we can test the program under QEMU once we have installed the modified kernel. We first need to download the test program using `scp` as we did for the kernel. However, we can just run it from our home directory without any installation necessary.

File Backups

One of the major contributions the university provides for the AFS filesystem is nightly backups. However, the `/u/OSLab/` partition on thoth is not part of AFS space. Thus, any files you modify under your personal directory in `/u/OSLab/` are not backed up. If there is a catastrophic disk failure, all of your work will be irrecoverably lost. As such, it is my recommendation that you:

Backup all the files you change under `/u/OSLab` or QEMU to your `~/private/` directory frequently!

BE FOREWARNED: Loss of work not backed up is not grounds for an extension.

Notes

- `printk()` is the version of `printf()` you can use for debugging messages from the kernel.
- In general, you can use some library standard C functions, but not all. If they do an OS call, they may not work

Requirements and Submission

We will use an automatic grader for Project 1. You can test your code on the autograder before the deadline. You get unlimited attempts until the deadline. It takes about two minutes to grade your solution.

You need to submit the following files into Gradescope:

- The three, well-commented, files (sys.c, syscall_table.S, unitstd.h) that you modified from the kernel and
- A header file, named sem.h, that contains the declaration of your struct cs1550_sem. To make sure that you put all required declarations into the file, try compiling one of the test case files and make sure that it compiles. The sem.h file should be in the same folder as the test case file when compiling.

Grading Sheet/Rubric

The rubric items can be found on the project submission page on Gradescope. A non-compiling code gets **zero** points.