University of Pittsburgh

# CS/COE 1550 – Introduction to Operating Systems

# Project 1: Syscalls[1]

Upload the files cs1550.h, cs1550.c, syscall_table.S, unitstd.h, and the writeup into GradeScope.

**Due Date**: Friday, September 25, 2020 @11:59pm
**Late Due Date**: Sunday, September 27, 2020 @11:59pm with 10% reduction per late day

## Table of Contents


**SEMAPHORE DATA TYPE AND SEMAPHORE LIST** ..... 2
**SYSCALLS FOR SYNCHRONIZATION** ..... 3
**SLEEPING** ..... 4
**WAKING UP** ..... 4
**ADDING A NEW SYSCALL** ..... 4
**TESTING THE SYSCALLS** ..... 5
**REQUIREMENTS AND HINTS** ..... 5
**SETTING UP THE KERNEL SOURCE (TO DO IN RECITATION)** ..... 6
**REBUILDING THE KERNEL** ..... 6
**RUNNING THE MODIFIED KERNEL AND THE TEST PROGRAMS** ..... 7
RUNNING QEMU ON THOTH ..... 7
RUNNING QEMU ON YOUR LOCAL MACHINE ..... 7
*QEMU installation on Mac OS X and Linux* ..... 7
COPYING THE FILES TO QEMU ..... 8
INSTALLING THE REBUILT KERNEL IN QEMU ..... 8
BOOTING INTO THE MODIFIED KERNEL ..... 8
HINTS ON USING THE QEMU VIRTUAL MACHINE ..... 8
RUNNING THE TEST PROGRAMS ..... 9
**DEBUGGING THE MODIFIED KERNEL** ..... 9
**FILE BACKUPS** ..... 10
**SUBMISSION** ..... 10
**GRADING RUBRIC** ..... 10


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

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 critical regions are mutually exclusive.

In this project, we will create a semaphore data type and implement the two semaphore operations that we described in class, namely down() and up().

## Semaphore data type and semaphore list

To encapsulate the semaphore, we'll make a simple struct that contains an integer value, an identifier, a spinlock, a security key, and a FIFO queue of processes:

```
struct cs1550_sem
{
   int value;
   long sem_id;
   spinlock_t lock;
   char key[32];
   char name[32];
   //Some FIFO queue of your devising
};
```

Each semaphore has a name and a unique long integer identifier. You are free to generate the identifier at random or use a global integer value that gets incremented with the creation of each semaphore. The FIFO queue stores pointers to the task control blocks of the processes waiting on the semaphore. The spinlock is used to do the increment or decrement of the semaphore's value and the following checks on it **atomically**. An alternative to spinlocks would be disabling interrupts (this is viable because the semaphore implementations is part of the kernel). However, in Linux, disabling interrupts is no longer the preferred way of doing in-kernel synchronization since we might be running on a multicore or multiprocessor machine. The security key protects access to the semaphore, so that only authorized processes can use the semaphore.

We must initialize the spinlock with the following function that takes a **pointer** to the spinlock.

```
spin_lock_init(spinlock_t*);
```

We can then surround our critical regions with calls to the following functions:

```
spin_lock(spinlock_t*);

spin_unlock(spinlock_t*);
```

Your implementation must store a system-wide list of semaphores. Access to this list of semaphores should be protected using a system-wide spinlock, which needs to be declared and initialized using the macro:

```
DEFINE_SPINLOCK(lock);
```

## Syscalls for Synchronization

We will then add five new system calls with the following signatures to create, open, and operate on our semaphores.

```
/* This syscall creates a new semaphore and stores the provided key to protect
access to the semaphore. The integer value is used to initialize the
semaphore's value. The function returns the identifier of the created
semaphore, which can be used to down and up the semaphore. */

asmlinkage long sys_cs1550_create(int value, char name[32], char key[32])

/* This syscall opens an already created semaphore by providing the semaphore
name and the correct key. The function returns the identifier of the opened
semaphore if the key matches the stored key or -1 otherwise. */

asmlinkage long sys_cs1550_open(char name[32], char key[32])

/* This syscall implements the down operation on an already opened semaphore
using the semaphore identifier obtained from a previous call to
sys_cs1550_create or sys_cs1550_open. The function returns 0 when successful
or -1 otherwise (e.g., if the semaphore id is invalid or if the queue is
full). Please check the lecture slides for the pseudo-code of the down
operation. */

asmlinkage long sys_cs1550_down(long sem_id)

/* This syscall implements the up operation on an already opened semaphore
using the semaphore identifier obtained from a previous call to
sys_cs1550_create or sys_cs1550_open. The function returns 0 when successful
or -1 otherwise (e.g., if the semaphore id is invalid). Please check the
lecture slides for pseudo-code of the up operation. */

asmlinkage long sys_cs1550_up(long sem_id)

/* This syscall removes an already created semaphore from the system-wide
semaphore list using the semaphore identifier obtained from a previous call to
sys_cs1550_create or sys_cs1550_open. The function returns 0 when successful
or -1 otherwise (e.g., if the semaphore id is invalid or the semaphore's
process queue is not empty). */

asmlinkage long sys_cs1550_close(long sem_id)
```

## Sleeping

As part of the 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 that was put to sleep in the down() operation.

For each of these syscalls, feel free to draw upon the text and handouts for this course as well as CS 0449.

## Adding a New Syscall

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

**1 and 2:** `linux-2.6.23.1/include/linux/cs1550.h` and `linux-2.6.23.1/kernel/cs1550.c`

These files contain the struct definitions and the actual implementation of the system calls.

**3:** `linux-2.6.23.1/arch/i386/kernel/syscall_table.S`

This file declares the numbers that correspond to the syscalls

**4:** `linux-2.6.23.1/include/asm/unistd.h`

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

The contents of these files should be self-explanatory. You may find it useful to look at the definitions of some of the existing syscalls in the file `linux-2.6.23.1/kernel/sys.c`.

## 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:

```
int down(long sem_id) {
       return syscall(__NR_cs1550_down, sem_id);
}
```

There are multiple test programs already provided with this description on Canvas. Please check these to get an understanding of how your semaphores will be used.

If we try to build the test programs 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:

1. `cd /u/OSLab/`whoami``
2. `cp /u/OSLab/original/trafficsim* .`
3. `gcc -g -m32 -o trafficsim -I /u/OSLab/`whoami`/linux-2.6.23.1/include/ trafficsim.c`
4. `gcc -g -m32 -lm -o trafficsim-mutex -I /u/OSLab/`whoami`/linux-2.6.23.1/include/ trafficsim-mutex.c`
5. `gcc -g -m32 -o trafficsim-strict-order -I /u/OSLab/`whoami`/linux-2.6.23.1/include/ trafficsim-strict-order.c`

## Requirements and hints

- There should be no artificial limit on the size of the semaphore list and the queue of each semaphore. One way of achieving that is by allocating memory as needed using kmalloc and freeing it using kfree. Check the manual pages for these two functions and make sure that you use a correct option for the allocated memory.
- There must be no memory leaks.
- You can get a pointer to the current process's `task_struct` by accessing the global variable `current`. You may need to save these pointers someplace (hint: in the semaphore's queue).

- Avoid calling `spin_unlock` on an already unlocked semaphore.
- Be careful not to send a process to sleep while the process is holding a spinlock. This may lead to a situation known as a deadlock.
- `printk()` (not `printf()`) is what you should use for printing debugging messages from kernel code. Here is an example:

```
printk(KERN_WARNING "cs1550_down syscall: pid=%d entered the critical
                section.\n",  current->pid);
```

- In general, you can use some library standard C functions, but not all. If they do an OS call, they may not work.
- You should put some logic to catch potential errors before they propagate further, especially if they will propagate into kernel code. So, if the client code is wrongly initializing the semaphore to a negative value or if the queue logic had had a bug, we do not want that error to cause a segmentation fault in kernel code.

## Setting up the Kernel Source (to do in recitation)

1. **On thoth**, copy the linux-2.6.23.1.tar.bz2 file to your local space under `/u/OSLab/USERNAME`

   ```
   cd /u/OSLab/`whoami`
   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 steps 1-4 once. Redoing these steps 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 type:

```
make ARCH=i386 bzImage
```

## Running the modified kernel and the test programs

We will be using an x86-based version of the Linux kernel and the QEMU emulator for this project to run the modified kernel. The username and password for the QEMU virtual machine are both the word **root**.

### Running QEMU on thoth

You can run QEMU on the thoth server as follows. On thoth type the following commands:

1. `cd /u/OSLab/`whoami``
2. `cp /u/OSLab/original/qemu.tar.gz .`
3. `tar xzvf qemu.tar.gz`
4. `cd qemu`
5. `make qemu`

### Running QEMU on your local machine

If you choose to run QEMU on your local machine, please follow the below steps depending on your machine's operating system.

**For Windows users.** The disk image and a copy of QEMU for windows are available on Canvas (qemu.zip).

**For Mac users.** For Mac users, you can download an older but GUI-based application (Q.app) available on Canvas as well. Point it at the tty.qcow2 disk image in the qemu.zip file.

**For Linux users (and Mac users wanting to use the homebrew version of QEMU).** You can find on Canvas a version of the disk image and a start.sh script to run it (the file name is qemu.tar.gz). It should be identical to the above version in terms of functionality, but boots with a recent version of QEMU. You will need to install QEMU as follows.

### QEMU installation on Mac OS X and Linux

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 that you get from extracting qemu.tar.gz** to launch qemu.

On Linux, using your appropriate package manager, install qemu-system-i386, likely part of your distro's qemu package.

Then run `start.sh` **in the unzipped folder that you get from extracting qemu.tar.gz** to launch qemu.

## Copying the Files to QEMU

Inside 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/`whoami`/linux-2.6.23.1/arch/i386/boot/bzImage .

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

## Installing the Rebuilt Kernel in QEMU

Inside 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. Ignore the warning that you may get.

## 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.

## Hints on using the QEMU virtual machine

- To properly shutdown the VM, please use the command: poweroff
- To properly restart the VM, please use the command: reboot

- If you decide to run QEMU on your local machine, you should be able to scroll up and down using shift+page up/page down on Windows and shift+fn+up/down arrow on Mac.

### Running the test programs

We cannot run our `test` program on `thoth` 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. Start QEMU with the modified kernel and download a test program (e.g. trafficsim) using `scp` as we did for the kernel. We can just run the test program from our home directory without any installation necessary.

```
scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/`whoami`/trafficsim .

./trafficsim
```

## Debugging the modified kernel

You can use GDB to debug the modified kernel. The following instructions assume that you run QEMU on thoth.

1. cd /u/OSLab/`whoami`/qemu
2. cp /u/OSLab/`whoami`/linux-2.6.23.1/vmlinux .
3. make qemu-gdb
4. Open a new SSH connection to thoth.
5. cd /u/OSLab/`whoami`/qemu
6. gdb -iex "set auto-load safe-path ."

   **Step 2-6 has to be repeated every time you modify and recompile the kernel.**

7. Feel free to use your GDB skills. For example, you can place a breakpoint at a given function in the kernel source code:
   (gdb) b sys_cs1550_down
8. Then continue running the kernel using the continue command
   (gdb) c
9. To debug a user program (e.g., trafficsim), you can add the symbol table inside the user program to the GDB session as follows
   (gdb) add-symbol-file /u/OSLab/`whoami`/trafficsim 0x08048430
10. You will need to replace the 08048430 value by the address of the .text section that you can get by running readelf -WS /u/OSLab/`whoami`/trafficsim
11. You would then be able to add breakpoints in the user program:
    (gdb) b trafficsim.c:main
12. Using the next command in the GDB prompt (n) you can trace inside kernel code and out to the user code.

13. Here is a nice page about gdb commands: https://visualgdb.com/gdbreference/commands/

## 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.

## 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 four, well-commented, files (cs1550.h, cs1550.c, syscall_table.S, unitstd.h) that you modified from the kernel and
- **Create a report to answer the following question:** *Explain the pros and cons of using a FIFO queue (as compared to a priority queue) within a semaphore.*

## Grading Rubric

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

| Item | Grade |
|---|---|
| **Test cases on the autograder** | 60% |
| **Correct implementation of the semaphore list (using kmalloc and kfree)** | 5% |
| **Correct implementation of the process queue (using kmalloc and kfree)** | 5% |
| **Correct usage of the spinlocks** | 5% |
| **Defensive programming to catch invalid initialization of the semaphore** | 5% |
| **Comments and style** | 10% |
| **Report** | 10% |

Please note that the score that you get from the autograder is not your final score. We still do manual grading. We may discover bugs and mistakes that were not caught by the test scripts and take penalty points off. Please use the autograder only as a tool to get immediate feedback and discover bugs in your program. Please note that certain bugs (e..g, deadlocks) in your program may or may not manifest when the test cases are run on your program. It may be the case that the same exact code fails in some tests then the same code passes the same tests when resubmitted. The fact that a test once fails sh ould make you try to debug the issue not to resubmit and hope that the situation that caused the bug won't happen the next time around.