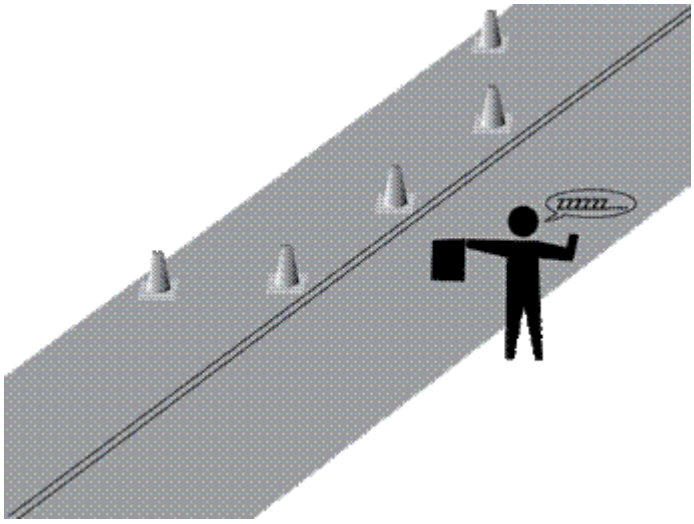
CS 1550 – Project 2: Syscalls and IPC
Due: Sunday, July 1, 2018, by 11:59pm

Project Description

Anytime we have shared 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.



One place that we might use parallelism is to simulate real-world situations that involve multiple independently acting entities, such as people or automobiles. In this project, you will be modeling a common roadway occurrence, where a lane is closed and a flagperson is directing traffic.

The figure above shows the scenario. We have one lane closed of a two-lane road, with traffic coming from the North and South. Because of traffic lights, the traffic on the road comes in bursts. When a car arrives, there is an 80% chance another car is following it, but once no car comes, there is a 20 second delay before any new car will come.

During the times when no cars are at either end, the flagperson will fall asleep, requiring the first car that arrives to blow their horn. When a car arrives at either end, the flagperson will allow traffic from that side to continue to flow, until there are no more cars, or until there are 10 cars lined up on the opposing side, at which time they will be allowed to pass.

Each car takes 2 seconds to go through the construction area.

Your job is to construct a simulation of these events where under no conditions will a deadlock occur. A deadlock could either be that the flagperson does not allow traffic through from either side, or lets traffic through from both sides causing an accident.

Simulation

Create a program, trafficsim, which runs the simulation.

- Treat the road as two queues, and have a producer for each direction putting cars into the queues at the appropriate times.
- Have a consumer (flagperson) that allows cars from one direction to pass through the work area as described above.
- To get an 80% chance of something, you can generate a random number modulo 10, and see if its value is less than 8. It's like flipping an unfair coin.

- Use the syscall nanosleep() or sleep() to pause your processes
- Make sure that your output shows all of the necessary events. You can sequentially number each car and say the direction it is heading. Display when the flagperson goes to sleep and is woken back up.

  Print out messages in the form:

```
The flagperson is now asleep.

The flagperson is now awake.

Car %d coming from the %c direction, blew their horn at time %d.

Car %d coming from the %c direction arrived in the queue at time %d.

Car %d coming from the %c direction left the construction zone at time %d.
```

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 the integer value:

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

We will then make two new system calls that each have 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);
```

Implementation

There are two halves of implementation, the syscalls themselves, and the trafficsim program.

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

Shared Memory in our simulation

To make our buffer and our semaphores, what we need is for multiple processes to be able to share the same memory region. We can ask for N bytes of RAM from the OS directly by using mmap():

```
void *ptr = mmap(NULL, N, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0);
```

The return value will be an address to the start of this page in RAM. We can then steal portions of that page to hold our variables much as we did in the malloc() project from 449. For example, if we wanted two integers to be stored in the page, we could do the following:

```
int *first;
int *second;
first = ptr;
second = first + 1;
*first = 0;
*second = 0;
```

to allocate them and initialize them.

At this point we have one process and some RAM that contains our variables. But we now need to share that to a second process. The good news is that a mmap'ed region (with the MAP_SHARED flag) remains accessible in the child process after a fork(). So all we need to do for this to work is to do the mmap() in main before fork() and then use the variables in the appropriate way afterwards.

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.

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
```

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.

Implementing and Building the `trafficsim` Program

As you implement your syscalls, you are also going to want to test them via your co-developed `trafficsim` 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(cs1550_sem *sem) {
       syscall(__NR_cs1550_down, sem);
}
```

However if we try to build our code 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
```

Running `trafficsim`

We cannot run our `trafficsim` 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 `trafficsim` 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 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` to your `~/private/` directory frequently!

Loss of work not backed up is not grounds for an extension. YOU HAVE BEEN WARNED.

Hints and 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
- Try different buffer sizes to make sure your program doesn't deadlock

Requirements and Submission

You need to submit:

- Your well-commented `trafficsim` program's source
- The three, also well-commented, files that you modified from the kernel

Make a tar.gz file as in the first assignment, named `USERNAME-project2.tar.gz`

Copy it to `~jrmst106/submit/1550` by the deadline for credit.