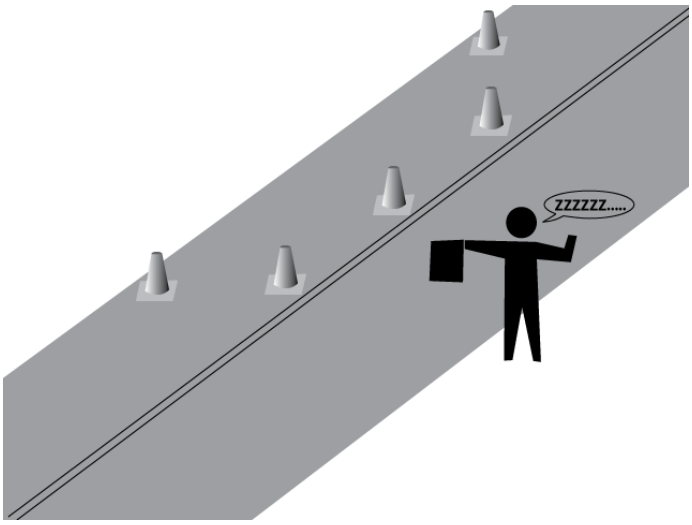
CS 1550 – Project 4: DIY Semaphores and IPC
Due: Sunday, April 7, 2024, by 11:59pm

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 a 75% chance another car is following it, but once no car comes, there is an 8 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 8 cars lined up on the opposing side, at which time they will be allowed to pass.

Each car takes 1 second 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 a 75% chance of something, you can generate a random number modulo 100, and see if its value is less than 75. 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 and a queue:

```
struct cs1550_sem
{
   int value;
   struct mutex *lock;    //So the kernel can lock on this instance
   //Some process queue of your devising
};
```

Memory in the kernel can be allocated by `kmalloc()`. You could use it to make nodes for a linked list, for example. Use `GFP_KERNEL` as the second parameter.

We will then make three new system calls that each have the following signatures (See below for how these should be declared in the kernel source files):

```
asmlinkage long sys_cs1550_seminit(struct cs1550_sem *sem, int value)

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 and potentially put the calling process to sleep while still being able to release the lock. The kernel provides two synchronization primitives: spinlocks and mutexes. For implementing system calls, we should use the provided **mutex** type and operations.

We can allocate a mutex in the `cs1550_seminit` system call with a `kmalloc` and initialize it:

```
mutex_init(sem->lock);
```

We can then surround our kernel critical regions with the following:

```
mutex_lock(sem->lock);

mutex_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 slides for this course.

Shared Memory in trafficsim

Just as we did in project 3, to make our buffer and our semaphores we need 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, -1, 0);
```

The return value will be an address to the start of this region in RAM. We can then steal portions of that page to hold our variables. For example, if we wanted two integers to be stored in the region, we could do the following:

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

to allocate them and initialize them. Do NOT call `mmap` more than once. Its minimal unit of allocation is a page (usually 4KB) and this would be very wasteful.

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.

Setting up, building, and installing the Kernel

Follow the exact same steps as in project 2. I'd suggest starting from the original kernel source (you can download/extract [if you kept the download] a new VM image if you want, or simply delete the old linux kernel folder and extract the source anew).

Note this means you'll have the same two hour-long builds as anytime we add system calls the entire kernel will be rebuilt. Make sure you start this setup early.

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 can write wrapper functions or macros to make the syscalls appear more natural in a C program. For example, since we are on the 32-bit version of x86, you could write:

```
void down(cs1550_sem *sem) {
	syscall(441, sem);
}
```

And something similar for `up()` and `init()`.

Running `trafficsim`

Make sure you run trafficsim under your modified kernel. If you try to invoke a system call that isn't part of the kernel, it will appear to return -1.

File Backups

Backup all the files you change under `VirtualBox` to your `~/private/` directory frequently!

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

Copying Files In and Out of VirtualBox

Once again, you can use scp (secure copy) to transfer files in and out of our virtual machine.

You can backup a file named sys.c to your private folder with:

```
scp sys.c USERNAME@thoth.cs.pitt.edu:private
```

Hints and Notes

- 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
- sys.c containing your implementation of the system calls

Make a tar.gz file named `USERNAME-project4.tar.gz`

Upload it to Canvas by the deadline for credit.