CS 1550 – Project 4: DIY Semaphores
Due: Thursday, November 16, 2023, by 11:59pm

Project Description

We've have been learning about synchronization and solving the producer/consumer problem using semaphores. In this project, we will again modify the Linux kernel to add our own implementations of down() and up() as system calls and then use them to solve the producer/consumer problem.

How It Will Work

There will be a userspace application called `prodcons` that will implement the producer/consumer problem using processes. We will be using the fork() system call to create additional processes, with the number of consumers and producers specified on the command line followed by the size of the buffer.

If we run the executable as: `./prodcons 2 3 1000` (number of producers, number of consumers, buffer size), we would see something like this output:

```
Producer A Produced: 0
Producer B Produced: 1
Producer A Produced: 2
Producer B Produced: 3
Producer A Produced: 4
Consumer A Consumed: 0
Consumer A Consumed: 1
Consumer B Consumed: 2
Consumer C Consumed: 3
```

Basically, we will be producing sequential integers and then consuming them by printing them out to the screen. The program should run as an infinite loop and never deadlock. All producers and consumers share the same buffer (i.e., there is only one buffer total).

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

We will then make three new system calls that each have the following signatures:

```
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 pointer 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 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 `prodcons` program.

For each, feel free to draw upon the text and slides for this course.

Shared Memory in prodcons

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 prodcons Program

As you implement your syscalls, you are also going to want to test them via your co-developed `prodcons` program. The first thing we need is a way to use our new syscalls. We again 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 prodcons

Make sure you run prodcons under your modified kernel.

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 `prodcons` 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.