Assignment 6 (Synchronization)

06 April 2011

due 20 April at noon

In this assignment, you will use semaphores to achieve synchronization between multiple user processes. The semaphores have already been implemented for you within the kernel; check `src/geekos/semaphore.c` if you want to see the implementation. Also, in case you didn't get your shared variables working on the last assignment, that is now provided for you in the kernel too; see `src/geekos/sharedvar.c`.

At the user level, the semaphore API looks like this:

```
int Open_Semaphore(const char *name, int value);
int Wait_Semaphore(int sem);
int Signal_Semaphore(int sem);
int Close_Semaphore(int sem);
```

It should look a lot like the API for shared variables. Semaphores also have globally unique names, so cooperating processes just need to know the names of the semaphores they are sharing. The first time a process opens a semaphore with a particular name, the kernel will set aside some space for it and return a handle (an index into an array kept in kernel memory). Other processes that open the semaphore with the same name will receive a handle to the same semaphore.

This time, all of your work will be at the user level. Recall that when you ran the work program last time, it creates two copies of the compute process which share a variable. One of them increments it 800 times, the other increments it 900 times. In the end, the final value was rarely 1700. This was because some updates were lost.

Mutual exclusion

The first task is to fix this program so that updates to the shared variable are mutually exclusive — while one process is updating the counter, the other is excluded from it. You can do this by incorporating semaphore operations into `src/user/compute.c`. Just open the semaphore at the beginning and close it at the end. Call the semaphore whatever you want, but `mutex` is a common name for a semaphore used in this fashion. Then, enclose the read/write operations in a wait/signal pair.

Now, when you run `work` from the prompt, you should find that the result always comes out to 1700, no matter how many times you run it.

Producer/consumer

Your next task is to use semaphores to implement the producer/consumer problem (also known as bounded-buffer problem). This is started for you in `src/user/procon.c`. Besides `main()`, there are three functions: `initialize()`, `producer()`, and `consumer()`. The producer and consumer run as two separate processes, and initialize is called by both of them.

The idea is that the one process produces pieces of data and puts them into a shared memory buffer. The other process consumes them by reading the shared buffer. Also, the buffer has a small, fixed size, which cannot hold all the data at once. Therefore, the processes need to cooperate in order to ensure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The way `procon.c` is written, the producer will produce the integers 1, 2, 3, and so on, up to and including 100. The consumer will read all these integers and keep a running sum. It prints the sum at the end, which should be 5050. If you run `procon` as it is from the shell prompt, it will probably just output 0, or maybe some other small number. This is because the processes are not cooperating properly.

To get them to cooperate, we'll use two semaphores. These semaphores are not for mutual exclusion, like in the previous problem. Instead, they are called counting semaphores. They will keep track of the number of free and used spaces in the buffer:

```
#define BUFFER_SIZE 4
int elements_sem;  /* semaphore holding # data elements in buffer */
int spaces_sem;    /* semaphore holding # free spaces in buffer */
int buffer[BUFFER_SIZE];
```

You should open these semaphores in the `initialize` function. Then, the producer should wait until a free space is available before writing to the shared buffer, and signal that a new element is available after writing. Meanwhile, the consumer will wait until an element is available before reading it from the shared buffer, and then signal that free space is available after reading.

If you get both of these cooperating nicely, then there will be no deadlock between the producer and consumer, and the consumer will print the final sum of **5050**.

That's it: make sure your two user programs compile, and test them at the shell prompt. Commit your changes before the deadline. Good luck!