CIS–4020 Lab
Kernel Threads

© Copyright 2015 by Peter C. Chapin

Last Revised: September 13, 2015

1 Introduction

In this lab you will write a module that demonstrates the producer/consumer problem and its solution. Your module will create two kernel threads, one to be a producer of data and the other to be a consumer of that data. These threads will run periodically but with different periods (that you will vary). Your module will need to use kernel semaphores and mutex objects to keep the threads properly synchronized.

In addition your module will provide a simple `/proc` file that reports on the behavior of the threads. This is needed so you can check to see if they are working properly.

The precise data produced and consumed is entirely artifical in this exercise. The purpose of the lab is educational, of course, and to give you experience with thread synchronization issues inside the kernel. It also gives you more practice writing modules.

2 Creating Threads

Start with a skeleton module file. In the module initialization function, create two kernel threads using the names "thread-A" and "thread-B." Use the macro `kthread_run` to create the threads and start their execution.

The two threads should loop, calling the `msleep` function to sleep for a specified number of milliseconds. Start by having both of the threads sleep for 500 ms (you will experiment with changing this value later). The threads should loop exactly 256 times. The threads must terminate before the module that created them can be removed. Thus it is important to arrange a way for them to terminate gracefully or you will be forced to reboot each time you wanted to update the module!

Check your module to be sure it behaves as expected. The threads should show up in the process list with square brackets around their names. What happens if you try to remove the module while the threads are running?

3 Producer/Consumer

Let thread-A be the producer thread. It should use `kmalloc` to allocate a page of kernel memory, fill that page with `PAGE_SIZE` copies of an eight bit count, and then install a pointer to that page into a small buffer shared with the consumer. The producer should also increment a global counter of the number of pages it has produced.

For our purposes the buffer can be declared as a small array of pointers to unsigned characters. For example:

```
#define BUFFER_SIZE 8
static unsigned char *buffer[BUFFER_SIZE];
static int count = 0;
```

```
static int next_in = 0;
static int next_out = 0;
```

Here `next_in` is the index of the next available slot in the buffer, and `next_out` is the index of the next item to be removed from the buffer. You will also want to declare in the same place whatever mutex and semaphore objects are required. Be sure those objects are properly initialized. It is important these things be global variables that can be shared by the threads.

Note that every byte of the pages installed in the buffer should be filled with the same count. The first page produced should be filled with `PAGE_SIZE` copies of `0x00`. The second page produced should be filled with `PAGE_SIZE` copies of `0x01`, and so forth. Since the producer loops 256 times, the last page produced should be filled with `PAGE_SIZE` copies of `0xFF`.

Let thread-B be the consumer thread. It should extract a pointer from the buffer shared with the producer, verify that the count stored on that page is correct, and then use `kfree` to release the page back to the kernel. The consumer should also update global counters of the number of verified pages it has consumed and the number of error pages it has consumed (pages that don't verify).

Be sure that the producer and consumer use the proper locking to ensure that they maintain mutual exclusion when manipulating the buffer and that the buffer neither overflows nor underflows. See the class notes on Locking for the producer/consumer pseudocode.

4 Thread Monitoring

You must arrange for your module to create a `/proc` file that exposes the three count values (pages produced, verified pages consumed, error pages consumed). Because the amount of data to be returned through the `/proc` file is small, it may be more convenient to not bother with the seq file API. The basic module skeleton shows an alternative way of creating a `/proc` file.

Run your module with the producer and consumer sleeping for the same amount of time (500 ms each). Run it again with the producer running faster (only a 250 ms sleep). Run it a third time with the consumering running faster. Observe the behavior in each case to verify that no problems arise. Note that when the threads end the `/proc` file should report that 256 pages were produced and 256 verified pages (0 error pages) were consumed. However, while the threads are running the number of pages consumed might be slightly less than the number produced. Why?

Experiment with different buffer sizes also. Can you arrange things so the producer produces all of its output before the consumer processes even one item?

5 Optional

In this section I describe a few optional things to try that might interest some of you. Unless stated elsewhere these steps are not a required part of the lab.

- Modify your module so the count of items in the buffer is also shown in the `/proc` file. *Be careful!* The thread that tries to read the `/proc` file is different than either thread-A or thread-B. It should acquire the mutex before looking at the shared count variable. Will this cause any problems for the producer or consumer? What happens to the reading process if it hangs indefinitely while locking that mutex (due, for example, to an error in the program's logic)?

- Kernel threads in Linux are similar to ordinary threads and have independent process ID numbers. This means they run in a "process context" that is different from that used by the process that reads the `/proc` file.

  Have both the producer and consumer threads store their process ID numbers in separate global

variables (use the `current` pointer to look up the process ID in the thread's `task_struct`). Modify the handling of the `/proc` file to display these values along with the process ID of the calling process (obtained the same way).

- Linux has some mutex debugging features. Consult the configuration of the kernel to see which, if any, are currently activated. If necessary turn some mutex debugging features on and rebuild the kernel. Next modify your code to elicit the kinds of errors the debugging features can detect and see if they are, in fact, detected.

6 Report

Write a report for this lab following using LaTeX. Include highlights of your module code. Be sure to include a description of your code that explains what it does and how it works. Comment on the behavior of the system in your various experiments.