Assignment 5 – Process Synchronization (100 points)

In this assignment, we are going to use POSIX threads, semaphores and a mutex to illustrate the Producer-Consumer Problem.

You may want to read about the Producer-Consumer problem in the textbook, and you may want to look at the following web sites for POSIX threads, semaphores and mutexes on Linux:

http://faculty.cs.niu.edu/~hutchins/csci480/pthreads.htm
http://faculty.cs.niu.edu/~hutchins/csci480/semaphor.htm

You may want look at the program "thread_simple.cpp" at:
http://faculty.cs.niu.edu/~hutchins/csci480/pthread_simple.cpp
Try compiling and running it.

You can find more about the involved functions at die.net:
https://linux.die.net/man/2/

How does the Producer-Consumer problem work?

We will have several consumer threads and several producer threads and one buffer. Each producer creates widgets and inserts them into the buffer, one at a time. Each consumer removes widgets from the buffer, one at a time. As the buffer is a fixed size, we need to ensure that:

(a) no producer tries to insert a widget into the buffer when it is full, and

(b) no consumer tries to remove a widget from the buffer when it is empty.

We will not actually deal in widgets and a buffer. Instead, we will simply maintain an integer called Counter to keep track of how many widgets are in the buffer at any moment. Thus, the action of inserting a widget is incrementing Counter and the action of removing a widget is decrementing Counter.

We do not want to have two threads attempting to change the value of the counter at the same time, so we will use a mutex to limit access to it. Thus we can describe the actions as:

Insert a widget:
   Lock the mutex
   Increment Counter
   Unlock the mutex

Remove a widget:
   Lock the mutex
   Decrement Counter
   Unlock the mutex

We will also use two semaphores, NotFull and NotEmpty. Before we start, NotFull is initalized to BUFFER_SIZE and NotEmpty is initialized to 0. (The textbook refers to NotFull and NotEmpty as Empty and Full, respectively.)

If NotFull is not 0, (think of that as "true") then the buffer is not full, so there is space left in the buffer and a producer can insert a widget. If NotFull is 0 (think of that as "false"), then the buffer is full, so the producer must wait until NotFull is positive before it can insert a widget and then decrement NotFull.

If NotEmpty is not 0 (think of that as "true"), then the buffer is not empty, so there is at least one Widget in the buffer and a consumer can remove a widget. If NotEmpty is 0 (think of that as "false"), then the buffer is empty and the consumer must wait until NotEmpty is not 0 before it can remove an object and then decrement NotEmpty.

Notice that whenever we insert or remove a widget, Counter and NotFull and NotEmpty are all affected.

It may sound as if we have three counters for the buffer. In a sense, we do, but they are for different purposes. Everyone shares Counter. The NotFull semaphore is oriented to the producer's point of view: is there space to insert a widget? The NotEmpty semaphore is oriented to the consumer's point of view: is there a widget to remove? The fact that these are semaphores gives us the convenience of the wait() function (go into a wait state instead of busy-waiting).

Here is what the producer is doing each time:

```
wait(NotFull)
Insert()
post(NotEmpty)
```

Here is what the consumer is doing each time:

```
wait(NotEmpty)
Remove()
post(NotFull)
```

What do we need in the program?

The program should have #defines for 5 constants:

P_NUMBER = the number of producers = 6

C_NUMBER = the number of consumers = 4

BUFFER_SIZE = the maximum size of the buffer = 12

N_P_STEPS = the number of iterations for each producer thread = 4

N_C_STEPS = the number of iterations for each consumer thread = 6

(Try running your program several times with different values. You probably want P_NUMBER * P_STEPS = C_NUMBER * C_STEPS.)

You need to create an array of C_NUMBER threads to be consumers and an array of P_NUMBER threads to be producers. These are of type pthread_t.

You need functions: Insert(), Remove(), Produce() and Consume(). Each of them has one argument, which is the ID number of a thread.

When you create a producer thread, the start_routine argument should be Produce(), and when you create a consumer thread, the start_routine argument should be Consume().

In Produce(), you need a loop to call Insert() P_STEPS times. In Consume(), you need a loop to call Remove() C_STEPS times. When each thread reaches the end of its work, it should exit with pthread_exit().

The main program is also a thread. After creating the threads, its principal job is to wait for the other threads to terminate. It does this using pthread_join() in a loop. Once it is the only remaining thread, it should use sem_destroy() to get rid of the semaphores and mutex_destroy() to get rid of the mutex and then terminate itself.

Since we want to be able to follow what is happening, Insert() and Remove() should print out a message whenever they are called, stating the ID number (0 to P_NUMBER-1 or 0 to C_NUMBER-1) of the calling thread, whether it is a producer or a consumer, what it has just done, and the value of Counter, as in:

"Consumer 1 removed a widget. Total is now 2"

(As this refers to the value of Counter, it may be well do this before unlocking the mutex.)

The thread knows its ID number because we pass it to Produce() and Consume() as an argument (the argument of start_routine()), and then the ID is passed in turn to Insert() and Remove().

(There is a concept in LINUX of a thread ID, which we could obtain with the gettid() function, but we are not using this; we are just numbering our own threads in our own program.)

To make all this easier to follow, we will insert some delays. Each time we call Insert() or Remove(), we will then have the thread sleep for 1 second: sleep(1). Put this in Produce() and Consume().

Most of these functions will give us an integer return value which is zero for success and nonzero for failure. Whenever you use one of them, check the return value and if it is not 0, print an error message (using cerr or stderr) identifying what when wrong and at what point, and then exit with the value -1.

Comments

You may use other variable and function names if you want, but they should be reasonable names reflecting their purposes.

Your program should be appropriately indented and well documented as usual.

You may use the standard template libraries (STL) if you wish.

You should have a makefile. The name of the executable file should be "Assign5".

When you run the program, it will seem to run slowly because of the use of sleep().

If you run your program several times and compare the output, you probably will find that the order of the lines is not always the same. This happens because the threads are executing independently. For this reason, your output is unlikely to duplicate the sample output.

When you are done, you need to submit your work on Blackboard. As in the other assignments, you should create a tar file containing the files involved: the program file(s), header file(s) and the makefile. To do this, you need the "tar" utility.

Do the following (replacing "Znumber" with your own Z-ID):

(a) Create a subdirectory named Znumber_A5_dir.

(b) Copy the files into it (source code files, headers, input file -- if there’s any, and makefile).

(c) In the parent directory of Znumber_A5_dir, use this command:

```
tar -cvf Znumber_A5.tar Znumber_A5_dir
```

Use an FTP program to retrieve the tar file and then submit it on Blackboard. The TA will move it to hopper, extract the files and run your makefile, as in:

```
tar -xvf Znumber_A5.tar
cd Znumber_A5_dir
make
Assign5
```

As usual, if your makefile does not run or your program does not compile and run, you will receive no credit.