CS105 – Computer Systems Fall 2019

Assignment 6: Sync Lab

Due: Tuesday, November 19, 2019 at 11:59pm

A *ring buffer*, also called a *circular buffer* or a *bounded buffer*, is a common method of sharing information between producers and consumers. As we saw in class, producer/consumer programs can run into a variety of synchronization problems. In this lab, you will implement a simple producer/consumer program and synchronize it with condition variables.

As usual, you should complete this lab with a partner of your choice.

As usual, the lab is available by downloading a tar file. Unpacking the file with "`tar xvf ringbuf.tar`" will create a subdirectory named `ringbuf` containing the writeup, a `makefile`, and some test files.

Specifications

You are to write a program named `ringbuf.c` from scratch. *Be sure to document the names of all team members in comments at the top of the file.*

Your program must be implemented using POSIX threads. There will be a `main` thread, one producer thread, and two or more consumer threads. The producer will read information from standard input and place it into the ring buffer. The consumers will independently extract information from the buffer and perform operations on the values.

The number of consumers will be between two and eight, specified by a command-line argument. Your program must create the consumer threads and then a producer thread. It will then collect all three threads with `pthread_join`, print a summary of the information they return, and then exit. The consumer threads *must* be created first or your output might not match our test cases.

You must use the POSIX mutex and condition functions listed on page 6-4. You may *not* use semaphores of any type. Further, you may not implement a solution that uses any type of polling, regardless of whether or not the polling wastes the CPU.

All library and system calls should be checked for errors. If an error occurs, print an informative message and terminate the program.

The Shared Buffer

The producer and consumer will communicate through a shared buffer that has 10 slots. (The size should be set by a `#define` so that it is easy to change.) Each slot in the buffer must have the following structure.

```
struct message {
    int value;          /* value to be passed to consumer     */
    int consumer_sleep; /* time (in ms) for consumer to sleep */
    int line;           /* line number in input file          */
    int print_code;     /* output code                        */
    int quit;           /* non-zero if consumer should exit   */
};
```

The fields have these purposes:

`value` A data value to be passed to the consumer.

`consumer_sleep` A time, expressed in milliseconds, that the consumer will expend in consuming the buffer entry.

`line` The line number in the input file that this data came from. *Line numbers start at 1.*

`print_code` A code indicating whether the consumer should print a status report after consuming this line. The print codes are interpreted as follows:

0 No messages are printed for this input line.

1 The producer generates a status message.

2 The consumer generates a status message.

3 Both the producer and consumer generate status messages.

`quit` For all buffer entries seen by the consumer except the last, this value will be zero. For the last entry, it will be nonzero. The consumer should *not* look at any of the other fields in the message if `quit` is nonzero.

Besides the shared buffer itself, you will need a number of auxiliary variables to keep track of the buffer status. These might include things like the index of the next slot to be filled or emptied. You will also need some `pthread` *conditions* and *mutexes*. The exact configuration is up to you, but you should make sure that the entire shared buffer can be used.

We recommend that you write functions `enqueue` and `dequeue` to read and manipulate the shared buffer and its auxiliary variables. By doing that, you will isolate the critical sections and can easily manage access to them.

The Producer

The producer is to read one line at a time from the standard input. Each line consists of four numbers, as follows.

- A value to be passed to the consumer.
- An amount of time the producer should sleep, given in milliseconds.
- An amount of time the consumer should sleep, given in milliseconds.
- A "print code" indicating what sorts of status lines should be printed.

For each line, the producer will sleep for a time given in the line, pass a value to the consumer via the ring buffer, and (optionally) print a status message. Note the order: the producer sleeps first, then places information in the buffer, and finally prints. Since printing is slow, the producer must not hold any mutexes while it is sleeping or printing.

The producer may read the four numbers on a line using the C library function `scanf`. See "`man scanf`" for more information. When `scanf` returns an `EOF` indication, the producer will enter a number of quit messages into the queue. Each quit message will contain a nonzero `quit` field; the other fields are irrelevant and ignored. There must be one quit message for each consumer, so the producer must know how many consumers there are.

The producer's status message, if required, should be produced by calling `printf` with the following format argument.

```
"Producer: value %d from input line %d\n"
```

The Consumers

Each consumer waits for a message to appear in the buffer, extracts it, and then executes it. Be sure that the consumer does not act on a message until *after* it has been removed from the buffer. That way, the other threads can continue to work while the consumer is processing a message.

The argument (of type `void*`) to the thread when it is created will be the address of an integer in `main`. Each consumer thread will have a distinct integer that is used for communication in two directions. When the thread is created, the integer value will be the index of the consumer thread, a value between 0 and 8, inclusive. When the thread is finally finished, it will set the integer to the value of the sum that has been accumulated.

When a consumer extracts a message with a zero `quit` field, it processes the information in the message. It begins by sleeping the specified time; then adds the `value` field to a running total (initialized to zero); and if the `print_code` is 2 or 3, prints a status message . The status message must be generated by calling `printf` with the following format argument.

```
"Consumer %d: %d from input line %d; sum = %d\n"
```

The first integer argument is the index of the consumer.

When a consumer extracts a message with a nonzero `quit` field, it records the total it has calculated, prints that value using the following `printf` format, and then terminates its thread.

```
"Consumer %d: final sum is %d\n"
```

Again, the first argument is the index of the consumer.

The `main` Function

As mentioned earlier, the `main` function will

- extract the number of consumers from the command line, using `strtol`;
- initialize any necessary values;
- create the consumer threads, starting with consumer 0;
- create the producer thread;
- wait for all the threads to terminate; and
- print the grand total.

After it has "joined" with all the threads, it will add the results from the consumer threads and print the the last line of the program with the grand total, using this format argument.

```
"Main: grand total is %d\n"
```

Feedback

Create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

Submission

Use the course submission site to submit your program, which will consist of the single file `ringbuf.c`, and your feedback file. Include all team members as collaborators when you submit. Also, be sure the names of all team members are *clearly* and *prominently* documented in the comments at the top of both submitted files.

Useful Information

System and Library Calls

You will need to use a number of Unix system and C library calls. Read the documentation on them by using `man`. For example, to learn about `pthread_mutex_lock`, type "`man 3 pthread_mutex_lock`". (The "3" specifies that the manual page should come from section 3 of the manual, which describes the C library. You can usually omit it, but sometimes "man" will give you the wrong manual page and you have to be explicit.) The calls you will need to use are all documented in sections 2 and 3 of the `man` pages.

Become familiar and comfortable with the style of Unix manual pages. For example, many man pages have a "SEE ALSO" section at the bottom, which will lead you to useful related information.

To make sure you get the best grade possible even if there are bugs in your solution, we suggest that you include the following line at the top of your `main` function.

```
setlinebuf(stdout);
```

Doing so will ensure that when your program is run with standard output redirected to a file, any partial output will appear even if your program hangs. Speaking of that, you should test your program with redirected output; that changes the timing and reveals some bugs that will not appear if you only test with output to the terminal.

Pthreads Features

You will need to familiarize yourself with the following pthreads functions, at a minimum.

```
pthread_create
pthread_join
pthread_mutex_lock
pthread_mutex_unlock
pthread_cond_wait
pthread_cond_signal
```

You may choose to use other functions as well. Remember that you are *not* allowed to use the pthread semaphore functions (`sem_*`).

Sleeping

For historical reasons, there are many ways to get a thread to sleep for a specified time. The preferred method is `nanosleep`; see "`man 2 nanosleep`" for documentation. Note that you cannot simply convert

milliseconds to nanoseconds, because `nanosleep` requires that the nanoseconds field be less than 10^9. It is best to write a wrapper function that accepts sleep times in milliseconds. If the specified sleep time is zero, your program should *not* call `nanosleep`.

Compiling and Testing

To compile your program, you will need to `#include` several header files—to use threads, input and output facilities, sleeping, and error reporting. We recommend these:

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Use `gcc`, as usual, to compile your program. When you link your program, you will need to specify `-lpthread` on the command line. Without that library, your program *will* compile, load, and excute, but *will not* run correctly. We have supplied a `Makefile` that includes a sufficient set of directives.

To test your program, run it with the standard input redirected from a test file and, perhaps, the standard output redirected as well. Here is an example.

```
% ./ringbuf 4 < testinput1.txt > testoutput1.txt
```

Input Files

The handout includes seven test files for your experiments. Because of indeterminacies in the system, a file may produce different results from run to run. Changing the number of consumers will change the behavior as well. The total sum should be 42 in all cases.

`testinput0.txt` A small test case with no sleeping. You can use it to verify that your program is interpreting the `print_code` correctly.

`testinput1.txt` The test case from `testinput0`.txt, with 1-second sleeps for the producer and no sleeping for the consumers. We recommend that you begin testing with this file, because it generates results that are easy to interpret.

`testinput2.txt` The test case from `testinput0`.txt, with 1-second sleeps for the consumers and no sleeping for the producer. This file tests your ability to deal with situations where the producer runs far ahead of the consumers, so that the buffer is always full.

`testinput3.txt` A test case with randomly generated sleep times. At times, the producer will run ahead; at other times one or more of the consumers will catch up.

`testinput4.txt` Another test case with randomly generated sleep times, and also with randomly generated `print_code`s.

`testinput5.txt` A test case similar to the previous one, but with widely varying sleep times for the consumers. One consumer will "hang back" while the others do most of the work.

`testinput6.txt` A test case similar to `testinput0.txt`, but with long sleep times for the first two items. With two consumers, the correct behavior is that the last two items processed will be from line 26 and line 1, in that order. With more than two consumers, the last items will be from lines 26, then line 2, and finally line 1.

Sample Output

Here is one result of running our sample solution with two consumers and the test case `testinput4.txt`.

```
Producer: value -8 from input line 2
Consumer 0: value 3 from input line 1; sum = 3
Producer: value 1 from input line 3
Producer: value 10 from input line 4
Consumer 1: value 1 from input line 3; sum = -7
Consumer 1: value 4 from input line 5; sum = -3
Producer: value 0 from input line 6
Consumer 0: value 0 from input line 6; sum = 13
Producer: value -1 from input line 8
Consumer 0: value -1 from input line 8; sum = 12
Consumer 1: value 8 from input line 9; sum = -1
Consumer 0: value 5 from input line 12; sum = 24
Producer: value 10 from input line 14
Consumer 1: value 1 from input line 15; sum = 6
Producer: value 10 from input line 16
Producer: value 5 from input line 17
Producer: value -2 from input line 20
Consumer 0: value -2 from input line 20; sum = 46
Producer: value 1 from input line 21
Consumer 0: value 3 from input line 24; sum = 50
Consumer 1: value 9 from input line 23; sum = 6
Producer: value 6 from input line 26
Consumer 1: value 6 from input line 26; sum = 12
Producer: value -3 from input line 27
Producer: value -8 from input line 32
Consumer 1: value -4 from input line 30; sum = 0
Consumer 0: value -7 from input line 31; sum = 40
Consumer 1: value -8 from input line 32; sum = -8
Consumer 1: value -4 from input line 34; sum = -12
Producer: value -7 from input line 36
Producer: value -1 from input line 39
Consumer 1: value 3 from input line 40; sum = -5
Consumer 0: value 1 from input line 41; sum = 51
Producer: value 10 from input line 42
Producer: value 0 from input line 43
Consumer 1: value -2 from input line 44; sum = 3
Producer: value -8 from input line 46
Consumer 1: value -8 from input line 46; sum = -5
Producer: value 1 from input line 49
Consumer 0: value -8 from input line 47; sum = 36
Consumer 0: value 1 from input line 49; sum = 37
Consumer 1: value -1 from input line 48; sum = -6
Consumer 1: final sum is -6
Consumer 0: value 11 from input line 50; sum = 48
Consumer 0: final sum is 48
Main: grand total is 42
```