CS 105
Ring Buffer
70 points

1 Introduction

A *ring buffer*, also called a *circular buffer*, is a common method of sharing information between a producer and a consumer. Your textbook contains an elegant implementation using semaphores (see p. 1007). In this lab, you will implement a simple producer/consumer program *without* using semaphores. Instead, you will use `pthread` mutexes and condition variables. In so doing, you will learn some of the basics of synchronization and threads.

In real life, a producer does some amount of work to actually produce an item to place in the buffer. Similarly, a consumer would work on an item taken from the buffer. In your program, you will use "sleeping" to simulate the time delay incurred when producing and consuming items.

As usual, you are to work in groups of two. It is an Honor Code violation to work on the program without your partner present.

2 Specification

You are to write a program named `ringbuf.c`. We have provided a Makefile and starter code for you; you should familiarize yourself with the starter code and especially the sections marked with NEEDSWORK comments, which are the places where you will need to make changes. **Be sure to document the names of both team members in comments at the top of the file.** Useful information for your implementation appears in Section 3 of this writeup.

Your program must be implemented using POSIX threads (`pthreads`). There will be two threads: a producer and a consumer. The producer will read information from standard input (see Section 2.3 of this writeup) and place it into the ring buffer. The consumer will extract information from the buffer and perform certain operations.

2.1 Important Rules

- You may **NOT** use semaphores of any type to implement your solution. This includes implementing a semaphore construct yourself by building on more primitive thread constructs. You also may not implement a solution that uses any type of polling, regardless of whether or not the polling wastes the CPU. (In other words, your implementation cannot repeatedly check whether the buffer is full or empty, then "wait a while" before checking again. If your producer is capable of seeing two buffer-full conditions in a row without inserting anything in the

buffer, or if your consumer can see two buffer-empty conditions without removing anything, you have implemented polling and must come up with a different solution.)

- The main program we have provided creates a thread that runs the consumer and then calls the producer's procedure directly; this effectively makes the main thread the producer thread. After the consumer terminates, the main thread collects it with `pthread_join` and exits. (In an alternate design, the main program could create and collect two new threads.)
- All library and system calls should be error-checked. If an error occurs, print an informative message and terminate the program.

2.2 The Shared Buffer

The producer and consumer will communicate through a shared buffer that has 10 slots (the size is set by a `#define` so that it's easy to change). Each slot in the buffer has 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; see below */
    int quit;           /* Nonzero if consumer should exit ("value" ignored) */
    };
```

These fields have the following purposes:

`value` The actual data to be passed to the consumer; in this example the consumer will sum the values passed in.

`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, not zero!

`print_code` A code indicating whether the consumer or producer should print a status report after consuming or producing this line.

`quit` For all buffer entries except the last, this value should be zero. For the last entry, it should 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 such as the index of the next slot to be filled or emptied. You will also need some `pthread` "conditions" and "mutexes." The exact set is up to you.

2.3 The Producer

The basic task of the producer is to read one line at a time from the standard input. For each line, it will first sleep for a time given in the line, then pass the data to the consumer via the ring buffer. Finally, *after* the message has been placed in the ring buffer, the producer will optionally print a status message. Printing is slow, so the producer **must not hold any mutexes** while it's printing.

Each input line consists of four numbers, separated by spaces, as follows:

- The value to be passed to the consumer.
- An amount of time the producer should sleep, given in milliseconds. Note that the sleep must be done *before* placing information in the ring buffer.
- An amount of time the consumer should sleep, given in milliseconds.
- A "print code" integer that indicates what sorts of status lines should be printed.

The producer reads these four numbers using the C library function "`scanf`" (see "`man scanf`" for more information. Feel free to do an Internet search for examples of using `scanf`). When `scanf` fails, either by returning an `EOF` indication[1] or by returning fewer the four values, the producer should enter one more message in the ring buffer, without sleeping first. This message should contain a nonzero `quit` field; the other fields will be ignored by the consumer.

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.

The producer's status message should be generated *after* the data has been passed to the consumer via the ring buffer. It must print the value and line number from the input line, and be produced by calling `printf` with **exactly** the following format argument:

```
"Produced %d from input line %d\n"
```

[1] I.e., when the return value from `scanf` is the value `EOF`

2.4 The Consumer

The consumer waits for messages to appear in the buffer, extracts them, and then processes them (the "processing" just involves adding the value to a running sum). Note that the consumer does not act on the message until *after* it has been removed from the buffer, so that the producer can continue to work while the consumer is processing the message.

If the extracted message has a nonzero `quit` field, the consumer does not add the value to the sum, but simply prints the total it has calculated, using the following `printf` format:

```
"Final sum is %d\n"
```

It then terminates its thread without sleeping.

Otherwise (`quit` is zero), the consumer then sleeps for the specified time and *then* adds the `value` field to a running total (initialized to zero), and finally optionally prints a status message if the `print_code` is 2 or 3. The status message must be generated by calling `printf` with exactly the following format argument:

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

3 Useful Information

You will need to use a number of Unix system and C library calls. You can read the documentation on these calls by using "man". For example, to learn about `pthread_mutex_lock`, type "`man pthread_mutex_lock`". (Sometimes you need to specify the section of the manual you want, for example with "`man 3 printf`," but usually that's not needed. The calls you will need to use are all documented in sections 2 and 3 of the manual.) If some manual pages are not available on `wilkes`, you can usually find them online using an Internet search.

You should try to develop familiarity 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.

NOTE: Some of the pthread manual pages are taken directly from the Posix standard, which is written in a style that no rational human can be expected to understand. If you encounter a manual page that makes heavy use of the word "shall" and is otherwise incomprehensible, we suggest that you search the Internet for a clearer discussion. For example, we had good luck searching for "`pthread_cond_wait example`".

To make sure you get the best grade 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 won't appear if you only test with output to the terminal.

3.1 Downloading

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, Makefile, skeleton code, and test files.

3.2 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 pthreads semaphore functions (`sem_*`).

3.3 Sleeping

For historical reasons, there are many ways to get a thread to go to sleep for a specified time period. The preferred method is `nanosleep`; see "`man 2 nanosleep`" for documentation. **Note that if the sleep time exceeds 999 milliseconds**, you cannot simply convert milliseconds to nanoseconds because `nanosleep` requires that the nanoseconds field be less than 10^9. Also note that you will not need the second argument to `nanosleep`; you can set it to `NULL`. We have provided a wrapper function named `thread_sleep`, which accepts an argument in milliseconds and converts it into a correct `nanosleep` call. However, note that as given, the wrapper function DOES NOT WORK CORRECTLY: it always sleeps for 0.25 seconds. You will have to modify it to calculate a correct value.

Note that if the specified sleep time is zero, the wrapper function doesn't call `nanosleep`.

3.4 Compiling and Testing

We have provided a sample `Makefile` that will compile your program. **Pay attention to compiler warnings—and fix them!**

To test your program, run it with standard input redirected to a test file. For example:

```
% ./ringbuf < testinput1.txt
```

The lab `tar` file includes five test files for you to try out:

`testinput0.txt` A small test case with no sleeping. Note that because of indeterminacies in the system scheduler, this test file may produce different results from run to run. However, only it and `testinput4.txt` will ensure that you are interpreting `print_code` correctly.

`testinput1.txt` The test case from `testinput0`.txt, with 1-second sleeps for the producer and no sleeping in the consumer. **We recommend that you begin testing with this file, because it generates results that are easy to interpret.** Also, be sure that it produces exactly one pair of output lines per second. If the output comes along too quickly, or if your program appears to hang, you may be calling `nanosleep` incorrectly. (Hint: if you run this command:

```
% time ./ringbuf < testinput1.txt
```

it should report a "real" time of about 25 seconds.)

`testinput2.txt` The test case from `testinput0`.txt, with 1-second sleeps for the consumer and no sleeping in the producer. This file tests your ability to deal with situations where the producer runs far ahead of the consumer, 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 the consumer will catch up.

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

We also provide two sample outputs; your output should match them *exactly*. You can check for correctness with the following command:

```
% ./ringbuf < testinput1.txt | diff testoutput1.txt -
```

and similarly for `testinput2.txt`. The "`diff`" command will be silent if things match; otherwise it will tell you what lines are different. Watch out for whitespace errors!

4 Submitting

Use `cs105submit` to submit your program, which should consist of the single file `ringbuf.c`:

```
cs105submit -a 05 ringbuf.c
```

Be sure the names of both team members are **CLEARLY** and **PROMINENTLY** documented in the comments at the top of the file.

5 Grading

The lab is graded based on your program's behavior when run against the five provided test inputs. For each input, there are 10 points available for matching the expected sum. In addition, for `testinput1.txt` and `testinput2.txt`, there are an additional 10 points if your output *exactly* matches our sample output for those two cases. (The other three test cases have nondeterministic output, so we don't expect an exact match there.)

WARNING: When we test your code, we will run some background processes in another window to encourage the OS to schedule your threads in a less deterministic order. We suggest that you do the same so that some of your race conditions can be uncovered.

6 Sample Output

The following is the result of running our sample solution on the test case `testinput4.txt` (note that your interleaving of "Produced" and "Consumed" may differ since this test input has randomness built in):

```
Produced -8 from input line 2
Consumed 3 from input line 1; sum = 3
Produced 1 from input line 3
Produced 10 from input line 4
Consumed 1 from input line 3; sum = -4
Consumed 4 from input line 5; sum = 10
Produced 0 from input line 6
Consumed 0 from input line 6; sum = 10
Produced -1 from input line 8
Consumed -1 from input line 8; sum = 3
Consumed 8 from input line 9; sum = 11
Consumed 5 from input line 12; sum = 20
Produced 10 from input line 14
Consumed 1 from input line 15; sum = 40
Produced 10 from input line 16
Produced 5 from input line 17
Produced -2 from input line 20
Produced 1 from input line 21
Consumed -2 from input line 20; sum = 48
Consumed 9 from input line 23; sum = 53
Consumed 3 from input line 24; sum = 56
Produced 6 from input line 26
Consumed 6 from input line 26; sum = 55
Produced -3 from input line 27
Produced -8 from input line 32
Consumed -4 from input line 30; sum = 47
Consumed -7 from input line 31; sum = 40
Consumed -8 from input line 32; sum = 32
Consumed -4 from input line 34; sum = 34
Produced -7 from input line 36
Produced -1 from input line 39
Consumed 3 from input line 40; sum = 45
Consumed 1 from input line 41; sum = 46
Produced 10 from input line 42
Produced 0 from input line 43
Consumed -2 from input line 44; sum = 54
Produced -8 from input line 46
Consumed -8 from input line 46; sum = 39
Produced 1 from input line 49
Consumed -8 from input line 47; sum = 31
Consumed -1 from input line 48; sum = 30
```

```
Consumed 1 from input line 49; sum = 31
Consumed 11 from input line 50; sum = 42
Final sum is 42
```

7 P.S.

The answer:

http://en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker%27s_Guide_to_the_Galaxy