CS 105, Fall 2005
Ring Buffer

November 27, 2005

1 Introduction

A *ring buffer*, also called a *circular buffer*, is a common method of sharing information between a producer and a consumer. In class, we have seen an elegant implementation of a ring buffer using semaphores.

In this lab, you will implement a simple producer/consumer program *without* using semaphores. In so doing, you will learn some of the basics of synchronization and threads.

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 Specifications

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

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

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.

The main program must create a thread to run the consumer, and then call the producer directly. After the consumer terminates, the main thread should collect it with `pthread_join` and then exit. (Alternatively, the main program could create and collect two new threads. However, the consumer thread **must** be created first or your output might not match our test cases.)

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

2.1 The Shared Buffer

The producer and consumer will communicate through a shared buffer that has 10 slots (this should be given by a `#define`). 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;           /* NZ if consumer should exit */
    };
```

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.

`print_code` A code indicating whether the consumer should print a status report after consuming 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 pthreads "conditions" and "mutexes." The exact set is up to you.

2.2 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 sleep for a time given in the line, and 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. Since printing is slow, the producer must not hold any mutexes while it is printing.

Each input line consists of four numbers, 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" indicating what sorts of status lines should be printed.

You can read these four numbers using the C library function "`scanf`" (see "`man scanf`" for more information.

When `scanf` returns an EOF indication, your program 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.

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. It must be produced by calling `printf` with the following format argument:

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

2.3 The Consumer

The consumer waits for messages to appear in the buffer, extracts them, and then executes them. 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 prints the total it has calculated, using the following `printf` format:

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

It then terminates its thread.

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

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

3 Useful Information

You will need to make use of 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 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 manual.)

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 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. You may find it useful to write a wrapper function that accepts sleep times in milliseconds.

If the specified sleep time is zero, your program should *not* call `nanosleep`.

3.4 Compiling and Testing

To compile your program, you will need to `#include` several header files. To use threads, you need `<pthread.h>`. Any C program that uses `printf` or `scanf` needs `<stdio.h>`. Finally, as the `nanosleep` manual page informs you, you will need `<time.h>`.

To link your program, you will need to specify `-lpthread` on the command line. For example, you could put the following in your Makefile:

```
$(CC) $(CFLAGS) -o ringbuf ringbuf.c -lpthread
```

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

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

The lab kit 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.

`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.

4 Submitting

Use `cs105submit` to submit your program, which should consist of the single file `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 Sample Output

The following is the result of running our sample solution on the test case `testinput4.txt`:

```
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
```