CSE 2431 **Lab 2**
SP 23

Submissions Due
Wednesday, March 29th, 2023, 11:30 p.m. on Carmen

1. Goal

Before reading this lab description, be sure that you first read the Bounded Buffer Producers-Consumers pdf file on Carmen in the Lab2 folder. This lab assignment is based on the Producer-Consumer problem for a bounded circular buffer described in that document. You are provided with code in the Lab folder on Carmen which you will need to modify to solve the problem.

2. Introduction

In this lab, we will design a programming solution to the bounded-buffer problem using producer and consumer threads. One way to solve the problem uses three semaphores: two of these semaphores are *empty* and *full*, which count the number of empty and full slots in the buffer (empty and full are *counting* semaphores), and the third semaphores is *mutex*, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. In our solution for this lab, standard counting semaphores will be used for *empty* and *full*, but, rather than a binary semaphore used to provide mutual exclusion, in our solution, ***a mutex lock (*actually, three different locks, see below*)*** will be used in place of a *mutex* semaphore.

The producers and consumers - running as separate threads - will move items to (producers) and from (consumers) a buffer that is synchronized with these concurrency mechanisms (semaphores and locks): *empty (semaphore), full (semaphore), prod_mutex (for producers), cons_mutex (for consumers), and output_mutex (to ensure output values are ordered correctly)*. Using C, you are required to use the pthread package on stdlinux to solve this problem in this lab. After creating a lab2 directory, download the buffer.c source code file which is posted on Carmen in the Lab folder, and copy it to your lab2 folder/directory on stdlinux. You can do this by downloading the buffer.c file from Carmen after starting a firefox web browser in stdlinux, with the following command:

$ firefox https://carmen.osu.edu/#

After the Carmen opens, enter your login information, and go to our course page in Carmen. The buffer.c source code is in the Lab > Lab2 folder.

When you download buffer.c, it will be in your Downloads folder on stdlinux. To copy it to the lab2 folder, cd to that folder:

$cd ~/cse2431/lab2

Now, use this command to copy the file:

$cp ~/Downloads/buffer.c ./

This will copy the file from your Downloads folder/directory to the folder you are currently in, ~/cse2431/lab2 (./ at the end of the command in Linux means your current directory, so that is why you want to be in your ~/cse2431/lab2 directory when you execute the command above).

3. The Buffer

Internally, the buffer will consist of a fixed-size array of type buffer_item (which is defined using *typedef in the source code file*). The array of *buffer_item* objects will be manipulated as a circular queue.

The buffer will be manipulated with two functions, insert_item() and remove_item(), which are called by the producer and consumer threads, respectively. You can see a description of the code for the functions, which you must complete, in the buffer.c source file.

When you add code to the buffer.c source file which you have downloaded as described above, YOU SHOULD ONLY WRITE CODE FOR THREE FUNCTIONS: 1) main 2) insert_item (which is called by producer threads), and 3) remove_item (which is called by consumer threads).

The *insert_item*() and *remove_item*() functions will synchronize the producers and consumers using the algorithms described in the Bounded Buffer Producers-Consumers pdf file on Carmen in the Lab > Lab2 folder. The buffer.c code also requires code in main to initialize the mutual exclusion locks *prod_mutex and cons_mutex,* along with the *empty* and *full* semaphores (see below for information on how to do these initializations in main).

The *main*() function will create the specified number of separate producer and consumer threads (the number of threads of each type is specified in the command that is used to run the program as a process on the command line in Linux; see below). Once it has created the producer and consumer threads, the *main*() function will sleep for a period of time and, upon awakening, will terminate the application. The *main*() function will be passed three parameters on the command line, in the order specified below (after the name of the executable, which is always the very first parameter on the command line):

1. How long to sleep in seconds before terminating (use sleep()).
2. The number of producer threads.
3. The number of consumer threads.

In other words, if the user runs the program using:

$buffer 2 3 2

when the program runs as a process, the main thread will sleep for 2 seconds before awakening and terminating the process; 3 producer threads will be created in main; and 2 consumer threads will be created in main.

A skeleton for the main function appears as:

```
int main(int argc, char*argv[]) {
        /* 1. Get command line arguments argv[1], argv[2], argv[3] */
        /* 2. Initialize mutex locks and semaphores */
        /* 3. Create producer thread(s) */
        /* 4. Create consumer thread(s) */
        /* 5. Sleep */
        /* 6. (After awakening) return (0); (this will exit the program) */

}
```

4. **Producer and Consumer Threads:** The producer thread will alternate between sleeping for a random period and inserting a random integer into the buffer. Random numbers will be produced using the rand_r() function, which produces pseudo-random integers between 0 and RAND_MAX safely in multithreaded processes. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. Code for the producer and consumer threads appears in the buffer.c file on Carmen, and a description of the algorithm to be used to insert and remove items is given in the Bounded Buffer Producers-Consumers pdf file on Carmen.

Note: For pseudo-random number generation, you need to use rand_r(), which is reentrant (that is why it is called rand_r), and it is thread-safe. Here is a declaration of rand_r:

int rand_r(unsigned int **seedp)*;

It requires a seedp parameter, which is a pointer to a seed value. A seed value of 100 should be used (the code provided has this value for the seed). This will generate the same sequence of pseudo-random values each time the program is executed; this is important for the grader to be able to grade your lab output.

5. **Thread creation** in the pthread package

The following code sample demonstrates the pthread APIs for creating a new thread:

```
#include <pthread.h>

void *thread_entry(/*parameters*/) { /* the entry point of a new thread; this is the function in
                                        which the thread should begin execution. NOTE:
                                        producer threads should begin execution in the producer
                                        function, and consumer threads should begin execution in
                                        the consumer function. */
}

int main( ) {
        pthread_t tid;

        /* initialize mutex locks */

        /* initialize semaphores */

        /* get the default attribute */
        pthread_attr_init(&attr);
```

/*create a new thread – use two loops, one for the number of producer threads, and one for the number of consumer threads */

```
    pthread_create(&tid, &attr, thread_entry, NULL);
    /*NOTE: thread_entry is the name of the function in which the thread will begin execution
    */

}
```

The pthread package provides *pthread_attr_init*() function to set the default attributes for the new thread. The function *pthread_create*() creates a new thread, which starts the execution from the entry point specified by the third argument; **you must replace *thread_entry* by the name of the function in which a given thread is to begin execution.**

Note: Even though we will possibly create multiple producer threads and multiple consumer threads, we can use a single tid (thread id) and attr (attribute variable); that is, every time you create a thread, you can use the same tid and attr variable. The value in the variables for the various threads do not have to be preserved in the program; the OS will manage scheduling of the threads after they are created. You can use a for loop to create the number of producer threads specified on the command line, and a separate for loop for the specified number of consumer threads.

6. **Mutex locks** in the pthread package

The following code sample illustrates how mutex locks available in the pthread API can be used to protect a critical section using a mutex lock:

```
#include <pthread.h> pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

The pthread package uses the *pthread_mutex_t* data type for mutex locks. A mutex is initialized with the *pthread_mutex_init*(*&mutex, NULL*) function, with the first parameter being a pointer to the mutex. By passing *NULL* as a second parameter, we initialize the mutex to its default attributes.

The mutex is acquired and released with the *pthread_mutex_lock*() and *pthread_mutex_unlock*() functions. If the mutex lock is unavailable when *pthread_mutex_lock*() is invoked, the calling

thread is blocked until the current owner of the mutex lock invokes *pthread_mutex_unlock*(). All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero value.

In this lab, we will use **three separate mutex locks**: one for the producer threads, *prod_mutex*, one for the consumer threads, *cons_mutex*, and one for output, *output_mutex*.

7. **Semaphores** in the pthread package

The pthread package provides two types of semaphores - named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is declared:

```
#include <semaphore.h>
sem_t sem;

/* initialize a sempahore to BUFFER_SIZE; one semaphore will need
to be initialized to BUFFER_SIZE, and the other to 0; you need to
determine which is which. */
sem_init(&sem, 0, BUFFER_SIZE);
```

The *sem_init*() initializes a declared semaphore. This function is passed three parameters:

- A pointer to the semaphore
- A flag indicating the level of sharing
- The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value BUFFER_SIZE. Not all semaphores in our code will be initialized to this value (you need to determine what value each semaphore should be initialized to).

For the semaphore operations *wait* and *signal* discussed in class, the pthread package names them sem_wait() and sem_post(), respectively. The code example below creates a binary semaphore mutex with an initial value of 1 and illustrates its use in protecting a critical section: (Note: The code below is only for illustration purposes. Do not use this binary semaphore for protecting a critical section. Instead, you are required to use the mutex locks, as explained above, provided by the pthread package for protecting a critical section.)

```
#include <semaphore.h>
sem_t sem_mutex;

/* create the semaphore */
sem_init(&sem_mutex, 0, 1);
. . . .
/* acquire the semaphore */
sem_wait(&sem_mutex);
```

```
/*** critical section ***/

/* release the semaphore */
sem_post(&sem_mutex);
```

8. Compilation

You need to link two special libraries to provide multithreaded and semaphore support using the following compilation command:

```
$ gcc buffer.c -lpthread -lrt -o buffer
```

The grader will compile the code this way.

To run the lab, for example, with a sleep time of 4 seconds, 3 producer threads, and 2 consumer threads, use:

```
$ buffer 4 3 2
```

We ***strongly encourage you*** to start your testing with:

```
$ buffer 2 1 1
```

In other words, 1 producer thread and 1 consumer thread, with a sleep time for main of 2. Once this works correctly, you can increase to 2 or 3 threads of each type (but no more than 2 or 3).

9. Testing

You can start by using one producer thread and one consumer thread for testing, and gradually use more producer and consumer threads (up to 3 of each type of thread). The grader will grade the lab with a small number of threads of each type, for example, something such as 3 producer threads and 2 consumer threads, so if your code works correctly with that number of threads of each type, it should be fine. For each test case, you need to make sure the random numbers generated by producer threads should exactly match the random numbers consumed by consumer threads **(*BOTH THE VALUES AND THE ORDER OF THE VALUES*). BE VERY SURE that the values consumed by the consumer(s), starting with the first value consumed, are in EXACTLY the same order as the values produced by the producer(s). This is an essential characteristic of a correct solution to the bounded buffer problem. NOTE THAT several values may be output as produced by producers before any values are output as having been consumed by consumers, but this is correct behavior, as long as the order of the values output by producers is the same as the order of values output by consumers. Please see the examples of correct output on Carmen (which will be posted in a day or so).**

Any program that does not compile will receive a zero. **The grader will not spend any time to fix your code due to simple errors you introduce at the last minute (or before)**. It is your responsibility to leave yourself enough time to ensure that your code can be compiled, run, and tested well before the due date.

Submission

Please start a firefox web browser in stdlinux as directed for Lab 1 (See the instructions in that lab description). **For this lab, be sure to submit only the buffer.c source file (and be sure your name is placed inside the comment at the top of the source file with a place for your name)**. For this lab, please DO NOT ZIP YOUR FILE BEFORE SUBMITTING. You should only submit the buffer.c source file; be sure you DO NOT SUBMIT AN EXECUTABLE, OR YOU WILL GET NO CREDIT!!

A late penalty of 25% will be assessed on a lab after the deadline, but within 24 hours after the deadline. Any lab submitted later than that will receive no credit, and not be graded. Labs must be the student's own work. Start early, and work steadily. Also, seek help *early* if you encounter problems.

Additional Notes

You can post questions on Piazza, as mentioned below, ***but be sure to leave your post private if you post any of your code.***

You are welcome to use Piazza to discuss questions/problems you are having with the lab with other students, but DO NOT LOOK AT ANY OTHER STUDENT'S CODE, AND DO NOT ALLOW ANYONE ELSE TO LOOK AT YOUR CODE. ***Please, also, do not post code unless you leave your post private (what it is by default), so that it can only be seen by instructors.*** You are also welcome to use information regarding Pthreads or the Producer-Consumer problem that you may find on the Internet but, ***do not look for, or look at, any code for this problem written by others (whether they were students in this course or not)! Doing so is considered academic misconduct.***