EECE.4810/EECE.5730: Operating Systems

Spring 2018

Programming Project 2

Due **11:59 PM**, **Wednesday, 3/21/18**

(3/21 is the Wednesday after Spring Break, so if you don't want to work on the program over break or save all the work for those last three days, start it early!)

1. Introduction

This project covers fundamentals of multithreading and synchronization. You will use Pthreads to write a multithreaded version of the producer-consumer model, with producers and consumers sharing a bounded, circular buffer managed as a shared queue. The buffer will be part of a monitor—a structure holding shared data and the primitives to synchronize accesses to the data.

This assignment is worth a total of 100 points. The grading rubric given in Section 4 applies to students in both EECE.4810 and EECE.5730.

2. Project Submission and Deliverables

Your submission must meet the following requirements:

- You should submit three files:
  - `OS_program2.c` contains your `main()` function and global variables
  - `prod_cons_MT.h` contains your structure definitions and function prototypes
  - `prod_cons_MT.c` contains your function definitions. Global variables declared in `OS_program2.c` should be additionally declared as `extern` in this file—see Section 5: Hints for a brief description of the `extern` keyword.
- Programs must be submitted via e-mail sent to Dr. Geiger (Michael_Geiger@uml.edu), either as three separate attachments or as a .zip or .tar archive.
  - If you submit an archive file, please do not use the .rar format.
- You may work in groups of up to 3 on this assignment. If you work in a group, please list the names of all group members in a header comment at the top of each file in your submission, as well as in the email you use to submit your code.

3. Specification

General overview: As covered in class, the producer-consumer model covers any application in which one or more producers add data to some shared storage, and one or more consumers remove and process those data. A bounded buffer limits the amount of data that can be stored at any time; in a circular buffer, after reading or writing the highest-numbered location, a thread will return to the first location.

In your solution, the buffer should be managed as a FIFO queue—in other words, items will be removed from the buffer in the order they are added. You should not implement a linked queue (a collection of nodes in which each node holds the address of the next) for this assignment.

Implementation requirements: As noted in Section 2, you must submit three files, with all functions (except `main()`) and structures defined in the `prod_cons_MT.h/.c` files. At a minimum, the `prod_cons_MT.h/.c` files must contain:

- A structure definition for your monitor, including:
  - A shared buffer holding integer data
  - Any other variables required to manage the state of the buffer
  - Any locks and condition variables required to synchronize accesses to the buffer
- Prototypes (in the .h file) and definitions (in the .c file) for your producer and consumer thread functions:
  - The producer function should generate a given number of random values between 1 and 10 and add each of those values to the buffer.
    - If the buffer is full, the producer should wait until a buffer slot is available.
    - Section 4: Grading Rubric specifies how many values each producer thread should create, depending on the objective(s) being satisfied.
    - Section 5: Hints contains a note on random number generation.
  - The consumer function should read a given number of values from the buffer, printing each value as it is read.
    - If the buffer is empty, the consumer should wait until a value is available.
    - The number of values to be consumed depends on the number of values produced. In your solution, your consumer thread(s) should read all data written to the buffer.
      - A simple way to handle this task is to divide the total number of values evenly across all consumer threads, with one thread responsible for any extra values.
      - For example, if 10 values are written to the buffer and 3 consumer threads are created, two of the consumer threads will read 3 values and the third consumer thread will read 4 values.

Any additional structure and/or function definitions should be included in those two files as well.

3. Specification (continued)

Implementation requirements (cont.): The file `OS_program2.c` only contains the `main()` function and any global variable declarations, with your monitor declared as a global variable in that file. The monitor must be declared as `extern` in `prod_cons_MT.c` for the producer and consumer functions to access it. See Section 5: Hints for a description of `extern` declarations.

Input: Your executable should take three command line inputs:

- The size of the buffer
- The number of producer threads
- The number of consumer threads

For example, if your executable name is `prog2`, to run the program with a buffer size of 10, 5 producer processes, and 3 consumers, use the command: `./prog2 10 5 3`

Output: As shown in Section 6: Test Cases, your program prints several messages in each thread. Each message lists the thread type and "number" in addition to the information described below. (For example, all messages from the first producer start with "P0"). Thread numbers are generated in `main()` and passed to the thread with any other necessary information.

Your message formats should match what is shown in Section 6.

Your main function prints three types of messages:

- Each new thread (producer or consumer) created
- Each thread is joined at the end of the program
- Entire program is complete.

Each producer thread prints messages in four cases:

- Immediately after entering the thread, print the number of values to be produced
- As a value is added to the buffer, print the value and position in which it is added
- If the buffer is full, print one message when the thread blocks and another when it is woken up and allowed to continue executing.
- Just before exiting the thread, print a message indicating the thread is finished

Each consumer thread prints messages in four cases:

- Immediately after entering the thread, print the number of values to be consumed
- As a value is removed from the buffer, print the value and position from which it is removed
- If the buffer is empty, print one message when the thread blocks and another when it is woken up and allowed to continue executing.
- Just before exiting the thread, print a message indicating the thread is finished

4. Grading Rubric

Your assignment will be graded according to the rubric below; partial credit may be given if you successfully complete part of a given objective. **You should not submit separate files for each objective, nor should your program contain separate sections or functions for each objective**—your sole submission will be judged on how many of the tasks below it accomplishes successfully. The rubric may also be used as an outline for developing your program—for example, first write a program that accomplishes objective A, then modify it to accomplish objective B, and so on.

In the rubric, objective A is the base case—you cannot complete any of the other objectives without completing that one, and the number of points listed with that objective is essentially the minimum you can earn with a working program. For each additional objective, the number of points will be added to your base score.

Completing one objective gives you credit for all earlier ones, assuming you do not make fundamental errors that prevent the program from working in a given case. If you make an error that compromises the correctness of the program in some test cases, partial credit may be given for one or more objectives.

Objectives:

A. (20 points) Your program contains no synchronization and therefore supports only one producer thread and one consumer thread, which must be run in that order.
   - The producer should therefore produce just enough data to fill the buffer.
   - For this objective and all others that follow, the consumer thread(s) must consume all values written to the buffer. As described above, values should be divided evenly among the consumers, with one thread responsible for any extra values.

B. (+30 points) Your program supports two concurrent threads—one producer and one consumer—with appropriate synchronization to protect access to the buffer.
   - For this objective and all others that follow, each producer thread should produce twice as much data as the buffer holds. For example, given a buffer size of 5, each producer writes 10 values to the buffer.

C. (+25 points) Your program still uses only a single producer thread, but supports multiple consumer threads. There is no limit on the number of consumers.

D. (+25 points) Your program supports multiple producer threads as well as multiple consumer threads. The number of producers and consumers do not have to match one another. There is no limit on the number of producers or consumers.

5. Hints

Working with Pthreads: As with the examples covered in class, you will use the POSIX thread (Pthreads) library to implement this program. Compiling multithreaded programs with `gcc` on the machines in Ball 410 requires the `-pthread` flag. Compiling with `gcc` on other platforms may require this flag or the `-lpthread` flag.

The example programs posted on the course schedule page provide detailed examples of thread creation, argument passing, and joining. We did not cover Pthread synchronization in class, so a brief overview is below:

Pthread locks use the `pthread_mutex_t` data type, with all relevant functions beginning with `pthread_mutex_`. A basic discussion of these functions is below; in all cases, the argument "`lock`" is a pointer to the actual lock:

- Use `pthread_mutex_init(lock, attr)` to initialize a lock at the start of the program, and `pthread_mutex_destroy(lock)` to destroy the lock at the end of the program. Passing NULL to the initialization function as the second argument will use default settings, which are sufficient.
- The lock and unlock functions are `pthread_mutex_lock(lock)` and `pthread_mutex_unlock(lock)`.

Pthread condition variables use the `pthread_cond_t` data type, with all relevant functions beginning with `pthread_cond_`. A basic discussion of these functions is below; in all cases, the argument "`CV`" is a pointer to the actual condition variable:

- Use `pthread_cond_init(CV, attr)` to initialize a condition variable at the start of the program, and `pthread_cond_destroy(CV)` to destroy the condition variable at the end of the program. Passing NULL to the initialization function as the second argument will use default settings, which are sufficient.
- The wait, signal, and broadcast functions are `pthread_cond_wait(CV, lock)`, `pthread_cond_signal(CV)`, and `pthread_cond_broadcast(CV)`. Note that the wait function requires pointers to both the condition variable to wait on and the lock to be released while waiting.

More details about Pthread synchronization can be found on the following websites:

- https://computing.llnl.gov/tutorials/pthreads/
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

5. Hints (continued)

extern declarations: In this assignment, you must synchronize accesses to a monitor declared as a global variable. To make this variable accessible to functions in multiple files, it must be listed in all of those files but can only be actually declared once. The `extern` keyword indicates that a variable is declared in another file but should be accessible in the current file.

For example, say OS_program2.c contains the following global variable declaration:

```
int gVar; // Global variable
```

To allow functions in `prod_cons_MT.c` to access this variable, that file must contain the following external declaration outside of any function definitions:

```
extern int gVar;     // gVar defined in another location
```

Random number generation: The assignment requires that each producing thread generate random integer values between 1 and 10. The library function `rand()` generates pseudo-random numbers between 0 and RAND_MAX (a large, constant value defined in the header `<stdlib.h>`, which you must include to use this function). To get a random value between 0 and N, use the modulus operator (%) as follows:

```
X = rand() % (N+1);
```

For example, to generate a value between 0 and 5, you can use `rand() % 6`.

Note that `rand()` is not truly random—if you do not provide a different starting point, or "seed", each time you run your program, `rand()` produces the same values. To specify a seed, use the function `srand(unsigned int seed)`. While your test cases may not match mine, if you repeatedly use the same seed value, your program should produce the same set of random values each time.

A common way to get (close to) true randomness is to use the system time as the seed:

```
srand(time(0));
```

`srand()` should only be called once per program, before any calls to `rand()`. To use the `time()` function, you must include the `<time.h>` header.

Dynamic memory allocation: You may find it useful to dynamically allocate some of the storage used in this program, as static allocation of data requires a known maximum value. C functions for dynamic memory allocation have been covered in prerequisite courses and will not be discussed here.

For a refresher on the use of these functions, you can find my EECE.2160 slides on dynamic allocation at:

http://mjgeiger.github.io/eece2160/prev/f17/lectures/eece.2160f17_lec32_dyn_alloc.pptx

6. Test Cases

This section provides sample outputs only for a program meeting Objective D (multiple producers, multiple consumers) listed in Section 4. Your outputs should match these general forms and must provide all information outlined earlier in the program. Your outputs will likely include different values, and statements may be in a different order than these test cases (and from one run of your program to the next).

This program run assumes a buffer size of 3, 3 producer threads, and 4 consumer threads. It was started using the command: `./prog2 3 3 4`

```
Main: started producer 0
P0: Producing 6 values
Main: started producer 1
P0: Writing 10 to position 0
P1: Producing 6 values
Main: started producer 2
P0: Writing 6 to position 1
P2: Producing 6 values
Main: started consumer 0
C0: Consuming 4 values
P1: Writing 2 to position 2
Main: started consumer 1
C1: Consuming 4 values
P0: Blocked due to full buffer
Main: started consumer 2
C2: Consuming 4 values
P2: Blocked due to full buffer
Main: started consumer 3
C3: Consuming 6 values
C0: Reading 10 from position 0
P1: Writing 7 to position 0
C1: Reading 6 from position 1
C2: Reading 2 from position 2
C3: Reading 7 from position 0
C0: Blocked due to empty buffer
P0: Done waiting on full buffer
P0: Writing 8 to position 1
P1: Writing 8 to position 2
C1: Reading 8 from position 1
P2: Done waiting on full buffer
P2: Writing 10 to position 0
C2: Reading 8 from position 2
C3: Reading 10 from position 0
P0: Writing 5 to position 1
C0: Done waiting on empty buffer
C0: Reading 5 from position 1
P1: Writing 1 to position 2
C1: Reading 1 from position 2
P2: Writing 7 to position 0
C2: Reading 7 from position 0
```

6. Test Cases (continued)

```
C3: Blocked due to empty buffer
P0: Writing 8 to position 1
C0: Reading 8 from position 1
P1: Writing 4 to position 2
C1: Reading 4 from position 2
C1: Exiting
P2: Writing 9 to position 0
C2: Reading 9 from position 0
C2: Exiting
P0: Writing 8 to position 1
P0: Exiting
C3: Done waiting on empty buffer
C3: Reading 8 from position 1
C0: Blocked due to empty buffer
P1: Writing 10 to position 2
Main: producer 0 joined
P1: Exiting
P2: Writing 3 to position 0
C3: Reading 10 from position 2
Main: producer 1 joined
C0: Done waiting on empty buffer
C0: Reading 3 from position 0
C0: Exiting
P2: Writing 2 to position 1
C3: Reading 2 from position 1
P2: Writing 2 to position 2
P2: Exiting
C3: Reading 2 from position 2
C3: Exiting
Main: producer 2 joined
Main: consumer 0 joined
Main: consumer 1 joined
Main: consumer 2 joined
Main: consumer 3 joined
Main: program completed
```