Prof. Jingke Li (lij@pdx.edu); Classes/Labs: TR 1400-1550 @FAB 150, 88-10; Office Hours: TR 1300-1400 @FAB 120-06

Assignment 1: Shared-Memory Programming with C++

(Due Thursday, 4/25/24)

This assignment is to practice shared-memory programming using C++'s thread library. Specifically, you are to implement three versions of the producer-consumer problem. Read the program specifications below carefully, since the grading will be based not only on your programs' correctness, but also on their conformance to the specific requirements. For this assignment, there is no extra work for CS515 students.

This assignment carries a total of 20 points (6 points on each program, and 2 points on the report).

Producer-Consumer Problem

You've seen the producer-consumer problem in this week's lecture and lab. The base version of the problem describes two threads, the producer and the consumer, sharing a common fixed-size buffer. The producer repeatedly adds items to the buffer and the consumer repeatedly removes items from the buffer. The two threads run concurrently. The producer blocks and waits if the buffer is full and the consumer blocks and waits if the buffer is empty. The extended versions of the problem allow multiple producers and/or consumers.

Buffer Implementation For this assignment, the items are just integers, and the buffer is implemented as a circular queue. The implementation is given in the files `queue.h` and `queue.cpp`. It has the following interface functions:

```
Queue(int n)          // construct a queue for a given capacity n
void add(int i)       // add a new item i (to tail)
int  remove()         // remove and return an item (from head)
int  size()           // return current size of queue
bool isEmpty()        // return true if queue is empty
bool isFull()         // return true if queue is full
```

1. Base Version (One Producer and One Consumer)

Implement the base version in C++'s with its thread utilities (not with Pthread routines!). Name your program `prodcons1.cpp`.

Program Requirements

- Set two global parameters

```
int BUFSIZE = 20;        // queue capacity
int NUMITEMS = 100;      // total number of data items
```

- Define two routines, `producer()` and `consumer()`. Both routines print out a message at the beginning of their execution and another at the end, indicating their status. In the first message, include the CPU core ID that the thread is executing on:

```
Producer starting on core 1
...
Producer ending
```

- The `producer` routine adds integer values `1..NUMITEMS` to the queue, one at a time. After each addition, it prints out a message showing the value and the post-addition queue size:

```
Producer added 23 (qsz: 8)
```

- The `consumer` routine removes `NUMITEMS` integer values from the queue, one at a time. After each removal, it prints out a message showing the value and the post-removal queue size:

  ```
  Consumer rem'd 54 (qsz: 3)
  ```

- The program's `main` routine performs the following tasks in order:
  1. initialize a queue with the given capacity,
  2. create two threads, to run `producer()` and `consumer()`, concurrently,
  3. wait for the two threads to join back,
  4. print out a final message: `Main: all done!`

- Synchronization need:

  You need to figure out where and what kinds of synchronization are needed. Remember that the producer can't add any item to a full queue and the consumer can't remove any item from an empty queue. In either case, **"busy-waiting"** (e.g. simply using a while loop to check queue status) **is not acceptable**. (*Hint:* Use condition variables.)

2. Extended Version (One Producer and Multiple Consumers)

Implement the extended version in C++. Name your program `prodcons2.cpp`. The queue representation and the program parameters stay the same.

Program Requirements

- The program takes an *optional* command-line argument, `numCons` (number of consumers). If it is not provided, a default value of 1 is used.

  ```
  linux> ./prodcons2 10    // 10 consumers
  linux> ./prodcons2       // 1  consumer
  ```

- The `producer` routine behaves the same as in the base version, except that it also provides help for consumer threads to terminate properlty. (See below.)

- The `consumer` routine needs some change. It now has a parameter, an integer thread id:

  ```
  void consumer(int k) { ... }
  ```

  which should be included in all of its print-out messages:

  ```
  Consumer[3] starting on core 5
  ..C[3] rem'd 55 (qsz: 2)
  ...
  ```

  Since the consumer threads compete to remove items from the queue, it is unknown ahead of time how many items each consumer thread will remove. Therefore, we want each consumer to track how many items it has successfully removed from the queue. The workload distribution across all consumer threads should be reported at the end of the program:

  ```
  Consumer stats: [14,19,9,21,6,9,5,17] total = 100
  ```

  where the numbers inside the brackets are the items-removed counts from the individual consumer threads; `total` is the sum of these counts, which should equal to `NUMITEMS`. (*Hint:* You may consider using an array to save the counts.)

  A correct program implementation should produce a more-or-less balanced workload distribution.

- Consumer thread termination:

  Due to the dynamic work assigments, the consumer threads themselves don't automatically know when to terminate. Your program should implement the following termination scheme:

  > The producer adds `numCons` copies of a bogus "termination" item (say `-1`) to the queue after all the actual items are added. Upon receiving such an item, a consumer thread terminates itself.

- The `main` routine still performs the steps as in the base version, but with two changes:
  - It creates one producer thread and `numCons` consumer threads.
  - It prints out the consumer workload distribution statistics.
- The synchronization requirements are the same as in the base version, *i.e.*, no "busy-waiting" should be used.

3. Full Version (Multiple Producers and Multiple Consumers)

Implement this third version, again in C++ with its thread utilities. Name your program `prodcons3.cpp`.

Program Requirements

- The program takes two optional command-line arguments, `numProd` and `numCons`; both with a default value of 1:

  ```
  linux> ./prodcons3         // 1 producer, 1 consumer (equivalent to the base version)
  linux> ./prodcons3 2       // 2 producers, 1 consumer
  linux> ./prodcons3 2 10    // 2 producers, 10 consumers
  ```

- The `producer` routine now has an integer parameter, a thread id:

  ```
  void producer(int k) { ... }
  ```

  which should be included in all of its print-out messages:

  ```
  ..P[2] added 55 (qsz: 12)
  ```

- The producer threads coordinate the addition of the integer values `1..NUMITEMS` to the queue; each is responsible for an equal-sized segment. In the case `numProd` does not evenly divide `NUMITEMS`, the last producer will handle the extra values. For instance, if there are 3 producers and `NUMITEMS=100`, then the three segments should be `[1..33]`, `[34..66]`, and `[67..100]`. A producer's starting message should include the segment information:

  ```
  Producer[2] starting on core 3 [34..66]
  ```

  Each producer thread terminates itself after adding its share of items to the queue.

- Adding termination items:

  Since there are multiple producers, the task of adding termination items to the queue becomes an issue. Which thread should do it? A simple solution to have the `main` routine to do it. But you need to make sure that the producers have all complemented their work before these termination items are added to the queue.

- The consumer threads work the same as in the previous version.

Testing

Two shell scripts are provided for you to test your second and third programs with multiple parameter settings. You can run them and save their output in files for analysis:

```
linux> ./run2 > script2    // testing prodcons2
linux> ./run3 > script3    // testing ProdCons3
```

Things you can check by looking at the output include the workload distribution across the threads, and the interleaving activities among the threads.

Summary and Submission

Write a one- or two-page summary covering your experience with this assignment. Include the following:

1. Status of your programs. Do they successfully run on all tests you conducted? If not, describe the remaining issues as clearly as possible.
2. Parallelization behavior. Do you see interleaving execution among all the threads? For multi-consumer programs, are the workload evenly distributed among the threads?
3. Experience and lessons. What issues did you encounter? How did you resolve them? Do you have any advice for other people trying to learn multi-threaded programming?

You may add additional contents that you deem relevant.

Make a zip file containing your programs and your write-up. Submit it through the "Assignment 1" folder on Canvas.

Appendix. Sample Output

```
linux> ./prodcons3 3 4
prodcons3 with 3 producers, 4 consumers
Producer[1] starting on core 6 [34..66]
Producer[0] starting on core 5 [1..33]
..P[1] added 34 (qsz: 1)
..P[1] added 35 (qsz: 3)
...
Producer[2] starting on core 4 [67..100]
..P[1] added 39 (qsz: 8)
..P[2] added 67 (qsz: 9)
...
Consumer[0] starting on core 7
..C[0] rem'd 34 (qsz: 19)
..P[2] added 72 (qsz: 18)
...
Consumer[2] starting on core 4
Consumer[1] starting on core 5
Consumer[3] starting on core 1
...
```

```
...
..C[2] rem'd 26 (qsz: 1)
..P[1] added 28 (qsz: 1)
...
Producer[0] ending
Main added 4 ENDSIGNs (qsz: 4)
..C[2] rem'd an ENDSIGN (qsz: 3)
Consumer[2] ending
..C[0] rem'd an ENDSIGN (qsz: 2)
Consumer[0] ending
..C[3] rem'd an ENDSIGN (qsz: 0)
..C[1] rem'd an ENDSIGN (qsz: 0)
Producer[2] ending
Producer[1] ending
Consumer[3] ending
Consumer[1] ending
Producer stats: [33,33,34] total = 100
Consumer stats: [56,23,15,6] total = 100
Main: all done
```