Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Classes: M 16:40-18:30, W 16:40-17:20, Labs: W 17:25-18:55, all @ FAB 88-10.

Assignment 1: Shared-Memory Programming with C++/Java

(Due Wednesday, 1/29/20)

This assignment is to practice shared-memory programming using C++ and Java thread libraries. Specifically, you are to implement two versions of the producer-consumer problem in both C++ and Java. CS515 students are required to implement a further third version, in either C++ or Java (see details below).

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.

1. 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, who share 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 version of the problem allows multiple producers and/or consumers.

Buffer Implementation For this assignment, the buffer between the producer(s) and the consumer(s) is implemented as a circular queue.

For C++, the queue implementation is given in the files `queue.h` and `queue.cpp`. It has the following interface functions:

```
Queue(int)            // construct a queue for a given capacity
void add(int)         // add a new item (to tail)
int  remove()         // remove 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
```

For Java, there is a built-in `Queue` collection, with the following interface methods (`E` represent the type of the elements):

```
boolean add(E)        // add a new item (to tail)
E remove()            // remove an item (from head)
int size()            // return current size of queue
boolean isEmpty()     // return true if queue is empty
```

An integer queue can be created by:

```
Queue<Integer> queue = new LinkedList<>();
```

The capacity of a Java queue is unlimited, but we can use an external parameter (say `BUFSIZE`) to simulate a fixed-size queue, and to implement the method `isFull()`.

2. Base Version (One Producer and One Consumer)

Implement the base version in both C++ and Java. Name your programs `prodcons1.cpp` and `ProdCons1.java`. (As a Java convention, the class name inside a program file should match the file name.)

Program Requirements The following requirements are for both programs:

- 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 (including CPU core info):

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

  Note: For the Java program, the CPU core info in the beginning message is not required.

- 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()`, respectively,
  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, **a "busy-waiting" is not acceptable**. (*Hint:* Use condition variables in C++ and signal and wait in Java.)

3. Extended Version (One Producer and Multiple Consumers)

Implement the extended version in both C++ and Java. Name your programs `prodcons2.cpp` and `ProdCons2.java`. 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
Consumer[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.

- Also, due to the dynamic workload distribution, the consumer threads themselves don't automatically know when to terminate. See below for a proper termination scheme.

- 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.

- **Consumer Thread Termination**
  Your program should implement the following termination scheme:

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

- **Testing**
  A set of shell scripts are provided for you to test your programs. You can run them and save their output in files for analysis:

  ```
  linux> ./runc2 > script.c2    // testing prodcons2 (C++)
  linux> ./runj2 > script.j2    // testing ProdCons2 (Java)
  ```

  One thing you can check by looking at the output is that the workload are properly distributed across the threads.

Synchronization Requirements The same as in the base version, *i.e.*, **no "busy-waiting" is allowed**.

4. Full Version (Multiple Producers and Multiple Consumers)

Implement this version in either C++ or Java. Name your program `prodcons3.cpp` or `ProdCons3.java`, respectively. This part is required for CS515 students. For CS415 students, if you correctly implement this part, you will earn extra 20% points.

Program Requirements

- The program takes 0, 1 or 2 command-line arguments, resulting it behaving as the base version, the extended version, and the full version, respectively:

  ```
  linux> ./prodcons3         // base version, 1 consumer, 1 producer
  linux> ./prodcons3 10      // ext. version, 10 consumers, 1 producer
  linux> ./prodcons3 10 3    // full version, 10 consumers, 3 producer
  ```

- 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:

```
Producer[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] for segment [34..66] starting on core 3
  ```

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

Summary and Submission

Write a short (one-page) summary (in plain text or pdf) covering your experience with this assignment. What issues did you encounter? How did you resolve them? What can you say about the workload distribution of your programs? What is your impression of the two languages' thread suport? etc.

Make a zip file containing your programs and your write-up. Submit it through the "Assignment 1" dropbox on D2L.

Appendix. Sample Output from a Reference Implementation

```
linux> ./prodcons3 10 7
Prodcons3 with 10 consumers, 7 producers
Producer[0] for segment [1..14] starting
Producer[0] added 1 (qsz: 1)
Producer[0] added 2 (qsz: 2)
...
Producer[2] for segment [29..42] starting
Producer[0] added 14 (qsz: 14)
Producer[2] added 29 (qsz: 15)
...
Producer[6] for segment [85..100] starting
...
Consumer[0] starting
Consumer[0] rem'd 1 (qsz: 19)
Consumer[0] rem'd 2 (qsz: 18)
...
Producer[2] added 35 (qsz: 19)
Producer[1] added 15 (qsz: 19)
Producer[3] added 43 (qsz: 19)
Producer[4] added 57 (qsz: 19)
Consumer[0] rem'd 5 (qsz: 19)
Consumer[0] rem'd 6 (qsz: 18)
...
Consumer[0] rem'd an ENDSIGN (qsz: 1)
Consumer[8] rem'd an ENDSIGN (qsz: 1)
Consumer[5] rem'd an ENDSIGN (qsz: 1)
Main added 10 ENDSIGNs (qsz: 2)
Consumer[9] rem'd an ENDSIGN (qsz: 0)
Producer stats: [14,14,14,14,14,14,16,] total = 100
Consumer stats: [11,7,9,8,8,5,7,7,3,6,8,0,5,4,6,6,] total = 100
Main: all done
```