CS5300 - Parallel & Concurrent Programming
Fall 2023
Comparing Different Queue Implementations
Submission Date: 12th September 2023 9:00 pm

Goal: The goal of this assignment is to implement different queue implementations studied in the class and compare their performance. Implement these algorithms in C++.

Details. The textbook has described two methods for building a concurrent queue:

- Coarse Grained Locking Queue, *CLQ*: The coarse lock based queue is described in figure 3.1 on page 50 of the textbook (page 67 of the pdf).
- Non-Locking Queue, *NLQ*: The non-locking queue, Herlihy–Wing queue, described in figure 3.14 on page 73 of the textbook (page 90 of the pdf). As discussed in the class, this implementation is not lock-free.

You have to implement both the algorithms CLQ and NLQ **in C++**. You can respectively name them as CLQ.cpp and NLQ.cpp.

Experimental Setup: To test the queues, please develop a program which is outlined below.

Listing 1: Queue Tester

```
1  public class QTester<T> {
2
3      Queue<T> qObj;
4      long thrTimes[];
5
6      public static void main(String[] args)
7      {
8          // Initialize the appropriate queue - CLQ or NLQ
9          qObj = new Queue<T>();
10
11         // Initialize the thread times array
12         thrTimes = new long[n];
13
14         // Create n threads
15         for (int i=0; i<n; i++) {
16             // invoke testThread
17         }
18
19         // Wait for all the threads to complete
20         wait();
21
22         // Compute the various statistics on the
```

```
23              computeStats();
24          }
25
26          public void testThread() {
27
28              double unifVal, sleepTime, lambda;
29              Random unifRandObj, expRandObj;
30              long startTime, endTime;
31
32              T v;
33
34              // Perform the appropriae initialization
35              unifRandObj = new Random();
36              expRandObj = new Random(λ);
37
38              // Each thread executes 'numOps' operations
39              for (int i=0; i <numOps; i++) {
40
41                  p = unifRandObj.nextDouble();
42                  if (p < rndLt) {      // Enq case test
43
44                      v = new T(); // some random value
45
46                      startTime = getCurTime();
47                      qObj.enq(v);
48                      endTime = getCurTime();
49
50                  }
51                  else {
52
53                      startTime = getCurTime();
54                      v = qObj.deq();
55                      endTime = getCurTime();
56                  }
57
58                  // Store the execution time of the current operation
59                  thrTime[curThr] += (endTime − startTime);
60
61                  // Simulate perform some other tasks using sleep
62                  sleepTime = expRandObj.nextDouble();
63                  sleep(sleepTime);
64
65                  }
66              }
67          }
```

Here the function $computeStats()$ as the name suggests computes the various statistics on the times taken. It should compute the average timetaken for: (a) all the enq operations (b) all the deq operations (c) all the operations including both the enq and deq.

Input: The input to the program QTester will be a file, named inp-params.txt, consisting of the parameters described above: $n, numOps, rndLt, \lambda$. They are explained here:

- n is the number of threads;
- $numOps$ as the name suggests the number of operations to be performed by each thread;

- $rndLt$ is a value between 0 and 1 that decides whether the current operation is eqn or deq;
- λ is the average with which a thread sleeps exponentially between different invocations on the queues.

Output: The program QTester should output the following a file containing the average time taken to complete $numOps$ operations by all the threads for each implementation: CLQ and NLQ. You can name the output files as: CLQ-out.txt and NLQ-out.txt.

Report: You have to submit a report for this assignment. The report should first explain the design of your program while explaining any complications that arose in the course of programming.

The report should include a comparison of CLQ and NLQ implementations, assessing their performance through multiple runs. To achieve this, both implementations must be executed several (atleast 5) times and then averaged to measure the performance. The performance metric to compare is *throughput*, which is defined as the number of operations per second. The report should present the results as an average of five runs and display them in the form of graphs.

1. **Impact of the number of operations on throughput:** In this graph, the Y-axis will represent throughput, while the X-axis will depict the number of operations per thread ($numOps$). The $numOps$ will vary from 10 to 60 in increments of 10. To maintain consistency in the experiment, all other parameters will remain constant as follows: $n = 16, randLt = 0.5, \lambda = 5$.
2. **Impact of the number of threads on throughput:** In this graph, the Y-axis will indicate throughput, and the X-axis will illustrate the number of threads (n), which will vary from 2 to 32 in the powers of 2. Similar to the above experiment, all other parameters will remain constant as follows: $numOps = 30, randLt = 0.5, \lambda = 5$.

Both the graphs will have two curves CLQ and NLQ. Please detail any observations made from the experimental results in the report.

Deliverables: You have to submit the following:

- The source file containing the actual program to execute. Please name it as Src-⟨rollno⟩.cpp. Please follow this convention. Otherwise, your program will not be evaluated. If you are submitting more than one file name it as CLQ1-⟨rollno⟩.cpp, NLQ1-⟨rollno⟩.cpp and CLQ2-⟨rollno⟩.cpp, NLQ2-⟨rollno⟩.cpp.
- A readme.txt that explains how to execute the program.
- The report as explained above.

Zip all the three files and name it as ProgAssn2-⟨rollno⟩.zip. Then upload it on the google classroom page of this course. Submit it by above mentioned deadline. Please follow the naming convention. Otherwise, your assignment will not be evaluated by the TAs.

Evaluation: The break-up of evaluation of your program is as follows:

1. Program Design as explained in the report - 30%
2. The Graphs obtained and the corresponding analysis shown in the report - 30%
3. Program Execution - 30%
4. Code Documentation & Indentation - 10%.

Please make sure that you your report is well written since it accounts for 60% of the marks.

Extra Credit: For extra credit, one has to implement the following methods described below along with **enqueue** and **dequeue**:

1. **front():** Returns the front element/ reference of the queue if the queue is not empty without deleting them; otherwise, returns null.
2. **rear():** Similarly, this method returns the element/ reference at the rear position in the queue without deleting them; otherwise, it returns null.

Please explain the correctness of your implementation in the report. Explain the correctness by showing the Linearization Points (LPs) of the above methods.

Extra Credit Marks Distribution: Five points for the correct working of each of the methods for CLQ and 10 points each for each of the methods of NLQ. So, one can get 30 points of extra credit by implementing the above methods for both algorithms. Deliverables for the Extra Credit are the same as mentioned above with the source code file name as CLQ-EC-⟨rollno⟩.cpp and NLQ-EC-⟨rollno⟩.cpp.

Late Submission and Plagiarism Check: All assignments for this course has the late submission policy of a penalty of 10% each day after the deadline for 6 days Submission after 6 days will not be considered.

Kindly remember that all submissions are subjected to plagiarism checks.