Kyle Andrews

Professor Ming Li

CSCI 114

December 3, 2021

Bounded Buffer Queue: The thread safe way

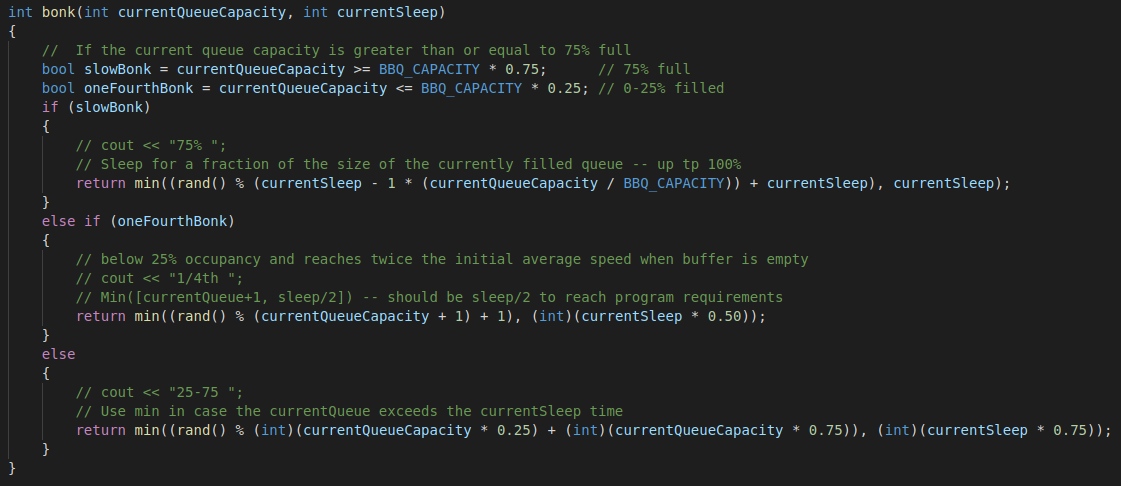
The bounded buffer queue is a queue that can be safely operate with threads. Using condition variables for insert and delete functions we can safely access the queue and take control of the queue from the thread level. No single thread can steal the queue from another thread using locks and condition variables.

This project contains a BBQ that inserts items and removes them via condition variables. Using a dynamic sleep for producers. The producers can speed up and slow down exponentially according to how full of empty the queue currently is. Consumers however are restricted to their range of sleep capabilities. Consumers are only allowed to sleep from a random sample from [0, T) where T is the user entered input.

To run the program, one needs to be on a UNIX machine. Testing the program worked in Mac but the program was extensively tested on a Linux operating system. When running the program, one may select a buffer size but when adjusting the buffer size, the program must be re-compiled as per program requirements. The user can specify TC and TP arguments which specify the users Thread Creation Sleep bounds and Thread Producer sleep bounds. Also, the program is using a queue built from a vector. We push items to the back of the vector and then pop them out as needed. It is still a FIFO operation, but the structure is using vector functions.

Observing the different inputs for accessing the queue we can form a hypothesis. If the thread producer sleep time is large and the thread consumer sleep time is near 0, the queue will never fill up and the consumers will be waiting more to consume. Flipping the situation to consumers having more sleep time and producers having no sleep time—the producers will be waiting for the consumers to empty the queue for more items to produce.

My sleep function I used for the project was supposed to accelerate the producer work and slowly ramp down the insertions as the queue fills up. My sleep function looks like this:

Taking the current queue capacity and using that as a determination to detect how fast or slow we should sleep. I’m still using rand() but the program description explicitly states to:

“… it (producer) should gradually accelerate the producing when the

buffer is below 25% occupancy and reaches twice the initial average speed when buffer is empty”

The sleep function does this but the calculation of the twice initial average speed when empty is hard to create with random values. I set a min, max using rand() and try to take 50% of the input sleep time and also rand%(currentQueueCapacity)+1. The rand() portion makes it so we are never sleeping for 0 ms and has a bound between [currentQueue + 1, 1). I potentially could remove this feature but it would make sense to produce more when the queue is empty. It was too late into the design to refer back to the details of sleeping for input/2 at 1/4th capacity.

Example output looks like this with a buffer size of 50 and sleep delay of 100 (ms):

Queue Size [0] sleeps for =>  1 ms   
Queue Size [1] sleeps for => 2 ms   
Queue Size [2] sleeps for => 2 ms   
Queue Size [3] sleeps for => 4 ms   
Queue Size [4] sleeps for => 5 ms   
Queue Size [5] sleeps for => 1 ms   
Queue Size [6] sleeps for => 1 ms   
Queue Size [7] sleeps for => 1 ms   
Queue Size [8] sleeps for => 1 ms   
Queue Size [9] sleeps for => 7 ms   
Queue Size [10] sleeps for => 6 ms   
Queue Size [11] sleeps for => 4 ms   
Queue Size [12] sleeps for => 1 ms   
Queue Size [13] sleeps for => 11 ms   
Queue Size [14] sleeps for => 11 ms   
Queue Size [15] sleeps for => 11 ms   
Queue Size [16] sleeps for => 15 ms   
Queue Size [17] sleeps for => 13 ms   
Queue Size [18] sleeps for => 15 ms   
Queue Size [19] sleeps for => 17 ms   
Queue Size [20] sleeps for => 15 ms   
Queue Size [21] sleeps for => 15 ms   
Queue Size [22] sleeps for => 17 ms   
Queue Size [23] sleeps for => 19 ms   
Queue Size [24] sleeps for => 21 ms   
Queue Size [25] sleeps for => 21 ms   
Queue Size [26] sleeps for => 21 ms   
Queue Size [27] sleeps for => 20 ms   
Queue Size [28] sleeps for => 23 ms   
Queue Size [29] sleeps for => 24 ms   
Queue Size [30] sleeps for => 25 ms   
Queue Size [31] sleeps for => 29 ms   
Queue Size [32] sleeps for => 25 ms   
Queue Size [33] sleeps for => 24 ms   
Queue Size [34] sleeps for => 28 ms   
Queue Size [35] sleeps for => 30 ms   
Queue Size [36] sleeps for => 32 ms   
Queue Size [37] sleeps for => 30 ms   
Queue Size [38] sleeps for => 100 ms   
Queue Size [39] sleeps for => 100 ms   
Queue Size [40] sleeps for => 100 ms   
Queue Size [41] sleeps for => 100 ms   
Queue Size [42] sleeps for => 100 ms   
Queue Size [43] sleeps for => 100 ms   
Queue Size [44] sleeps for => 100 ms   
Queue Size [45] sleeps for => 100 ms   
Queue Size [46] sleeps for => 100 ms   
Queue Size [47] sleeps for => 100 ms   
Queue Size [48] sleeps for => 100 ms   
Queue Size [49] sleeps for => 100 ms

The testing procedure consisted of setting up the function with the buffer sizes of 25, 50, and 100. Three separate tests were conducted and thread producer values were always set to 200 (ms). The thread consumer values tested were 50, 100, 150, 200, 250, 300, 400, and 500. The data plotted is the Producer Wait and Consumer Wait times emulating queue full/empty.

Notice how on Buffer 25 the chart was highly active and populated. There were several times when the queue was full and empty. As the buffer size increases the full and empty started to decrease. The inconsistency in all three graphs is to blame for the dynamic sleep function for the producing threads. If the sleep function did not include the current queue size and instead did a [rand() sleep/2] sleep delay I think the 3 graphs would be more consistent. Due to the size of the queue being included as a min(currentQueueCapacity, sleep/2) it creates variance between the 3 buffer sizes. During testing this was not apparent because the program was first developed using a buffer of 25. By developed I mean the sleep function itself. To meet the requirements of inserting for 2x the average speed of the sleep function. At the time the outputs look like such for a buffer of 25:

Queue Size [0] sleeps for => 1 ms

Queue Size [1] sleeps for => 1 ms

Queue Size [2] sleeps for => 2 ms

Queue Size [3] sleeps for => 1 ms

Queue Size [4] sleeps for => 4 ms

Queue Size [5] sleeps for => 1 ms

Queue Size [6] sleeps for => 4 ms

Queue Size [7] sleeps for => 5 ms

Queue Size [8] sleeps for => 6 ms

Queue Size [9] sleeps for => 6 ms

Queue Size [10] sleeps for => 7 ms

Queue Size [11] sleeps for => 7 ms

Queue Size [12] sleeps for => 7 ms

Queue Size [13] sleeps for => 7 ms

Queue Size [14] sleeps for => 7 ms

Queue Size [15] sleeps for => 7 ms

Queue Size [16] sleeps for => 7 ms

Queue Size [17] sleeps for => 7 ms

Queue Size [18] sleeps for => 7 ms

Queue Size [19] sleeps for => 10 ms

Queue Size [20] sleeps for => 10 ms

Queue Size [21] sleeps for => 10 ms

Queue Size [22] sleeps for => 10 ms

Queue Size [23] sleeps for => 10 ms

Queue Size [24] sleeps for => 10 ms

Notice how at lower values it is 2x average speed of a sleep of 10 ms. Developing this function was mostly tested on 10 and a buffer of 25. Later the program switched from sleep() because that uses seconds. I switched over to

this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_value));

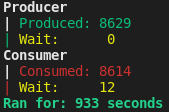
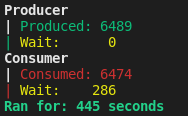
which performs much better yet exposes many flaws with the current sleep function. Either way the program works as expected by inserting and removing items and waiting for the queue to be available for insert/delete functions.

The BBQ itself uses condition variables and mutex to acquire locks on the insert and delete functions. Since the queue is only accessed by these functions it is safe to say this is how the threads acquire locks on the queue itself. I did learn however using the unique\_lock<mutex> lock function—you do not need to explicitly define unlocking. The unique lock handles this inside of its destructor. My program however has it defined for my own documentation on lock/unlock.

Lastly some miscellaneous testing of slower sleep times resulted in interesting results. While recording video and using a CPU intensive compression via x264 video encoder—two different tests produced drastically different results:

./project 2 10

\* Using CPU encoding (background task) \* Using GPU encoding (background task)



*Even after GPU test ran for 900 seconds (about 15 minutes) --the CPU test waited longer for producers in less time.*