**7376 HW5**

**Name: Xuebao Zhao NUID: 002108354**

**Problem1:**

1. **Nolock**

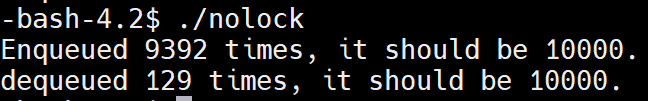
In this problem, we use four threads to enqueue and dequeue 100000 elements respectively without any locks. First, we create all the enqueue threads and join threads. After that we create all the dequeue threads and join them. Every time one element is enqueued/dequeued, we add one on the corresponding count(line 58 and line 42). Finally, we check the number count.

The implementation is not thread-safe, and there is a data race problem that can occur when two threads access the queue concurrently. For example, if two enqueue threads run concurrently, they can both update the tail pointer of the queue, causing one of the enqueued values to be lost.

For example, consider the case where two enqueue threads execute simultaneously. Thread 1 *“executes q->tail->next = tmp”*, setting the next pointer of the current tail to the new node. Then, before Thread 1 can execute *“q->tail = tmp”*, Thread 2 executes the same line, also setting the next pointer of the current tail to the new node. Now both threads have updated the tail pointer to point to the new node, and one of the enqueued values has been lost.

Also, there will be problem when we free a memory address. Consider the case when two dequeue threads execute simultaneously. Thread 1 execute *q->head = new\_head*, setting a new dummy head. Then before Thread 1 can execute it, Thread 2 execute node\_t \*tmp = q->head. When they execute *“free(tmp);”*, the two *“tmp”* point to the same address. It means it will be freed twice which will cause an error.

The count will smaller than 100000 which we want to enqueue and dequeue.



Attempted to free a memory address that has already been freed or is invalid.

Text

Description automatically generated

1. **One lock VS. Two locks**

In this problem, we also use four threads to enqueue and dequeue 100000 elements respectively. First, we create all the enqueue threads and join threads. After that we create all the dequeue threads and join them. Every time one element is enqueued/dequeued, we add one on the corresponding count(line 58 and line 42). “clock\_t” is used to obtain the running time. Finally, we check the number count and compare the running time. The only difference between these two codes is one uses lock while the other uses two.

**Result:**

**Text

Description automatically generated**

Based on the results of the performance comparison, using a single lock for both enqueue and dequeue operations appears to be better in terms of overall performance. This is because using a single lock reduces contention and overhead associated with managing multiple locks, leading to less waiting time and higher throughput.

1. **No dummy**

In this problem, we only use one enqueue thread and one dequeue to operate one element. We check if the element we enqueue is the same with the element we dequeue.

**Result:**

**Text

Description automatically generated**

It can be seen that the dequeued element is a wrong value. When using two locks for a concurrent queue, one lock protects the head pointer while the other lock protects the tail pointer. In this case, a dummy node is needed so that the update of the tail pointer is separated from the update of the head pointer. This is important to ensure that a dequeue operation does not encounter a null pointer exception when it tries to access the next field of a node whose next field is currently being updated by an enqueue operation.

Without the dummy node, it would be impossible to differentiate between an empty queue (where head and tail both point to NULL) and a queue with one element (where head and tail both point to the same node). By inserting a dummy node at the front of the queue, we can ensure that head always points to a valid node, even when the queue is empty. This allows us to distinguish between an empty queue (where head points to the dummy node and tail points to NULL) and a non-empty queue (where head points to the first real node in the queue and tail points to the last real node in the queue).

**Problem2:**

This program takes two arguments *“num\_elements”* and *“num\_threads”*, where *“num\_elements”* is the number of elements in the array to be summed and *“num\_threads”* is the number of threads to be used for the summation.

The program first initializes a vector v of size *“num\_elements”* and populates it with integers from 0 to *“num\_elements-1”*. It then initializes a mutex *“sum\_mutex”* for thread synchronization.

The program creates *“num\_threads”* threads, where each thread sums a subset of the array v. The runner function is the function that each thread runs. The *“runner\_args\_t”* struct holds the arguments for each thread, which includes a pointer to the array v, an index indicating the starting position of the subset of the array to be summed, and a size indicating the number of elements in the subset.

Each thread computes the partial sum of its subset of the array and then acquires the mutex lock to update the global variable sum by adding its partial sum to the global sum. Once the update is complete, the thread releases the mutex lock and frees its argument struct.

The main thread waits for all the threads to complete using pthread\_join() and then frees the resources and prints the final sum.

**Result:**

Graphical user interface

Description automatically generated with medium confidence

**Problem3:**

Table

Description automatically generated with medium confidence