The Design and Implementation of a Multi-access Threaded Queue

Xihuai Wang 16337236

Contents

1 Analytics and Design

To make sure that the queue works well when multiple threads inserting and multiple threads extracting from the queue, the queue should guarantee that all the threads see the same contents at the same time. That is, we should guarantee that access to the last element and pushing new element into the queue will not happen simultaneously, and that access to the first element and poping the head of the queue will not happen simultaneously, as well as that requireing the size or checking whether the queue is empty will not happen when threads are trying to modify the queue.

To analyse the performance of the queue, we need a comparsion, the same operations in serial case.

My design is that I encapsulates the queue template class from the stardand library and makes it support multi-access. To avoid unnecessarily blocking threads, we need 2 mutexlock, one is used to control the head of the queue, the other is used to control the tail of the queue. Both of the two is used to pretend modification of the queue when requireing the size or checking the status of the queue. Besides, the two methods the template class designed provided, front() and back() will return a copy of the element in the corresponding place rather return a reference to the element, because the threads may make a mess if that.

To analyse the performance, I calculate the time the multi-threads version used to insert and extract to and from the queue a certain times, and the time the serial version used.

2 Implementation Code

The multi-access queue is implemented using pthread library.This section shows the main components of the code.

The declaration of the MultiAccessQueue template class is shown in code block below.

```
template<typename T>
class MultiAccessQueue
{
public:
  MultiAccessQueue();
  ~MultiAccessQueue();
  bool empty();
  size_t size();
  T front();
  T back();
  void push(T ele);
  void pop();

private:
  std::queue<T> oriQue;
  pthread_mutex_t headLock;
  pthread_mutex_t tailLock;
};
```

The queue instantiation and the 2 locks are the most important designs. Most methods of queue are overloaped.

As an example, the implementation of size() and push() methods are shown in the following code block.

```
template<typename T>
size_t MultiAccessQueue<T>::size()
{
  pthread_mutex_lock(&headLock);
  pthread_mutex_lock(&tailLock);
  size_t ret = oriQue.size();
  pthread_mutex_unlock(&headLock);
  pthread_mutex_unlock(&tailLock);
  return ret;
}

template<typename T>
void MultiAccessQueue<T>::push(T ele)
{
  pthread_mutex_lock(&tailLock);
  oriQue.push(ele);
  pthread_mutex_unlock(&tailLock);
}
```

The main work is to lock and unlock the mutex lock.
And the flow chart of the program to use this queue is shown is figure 1.

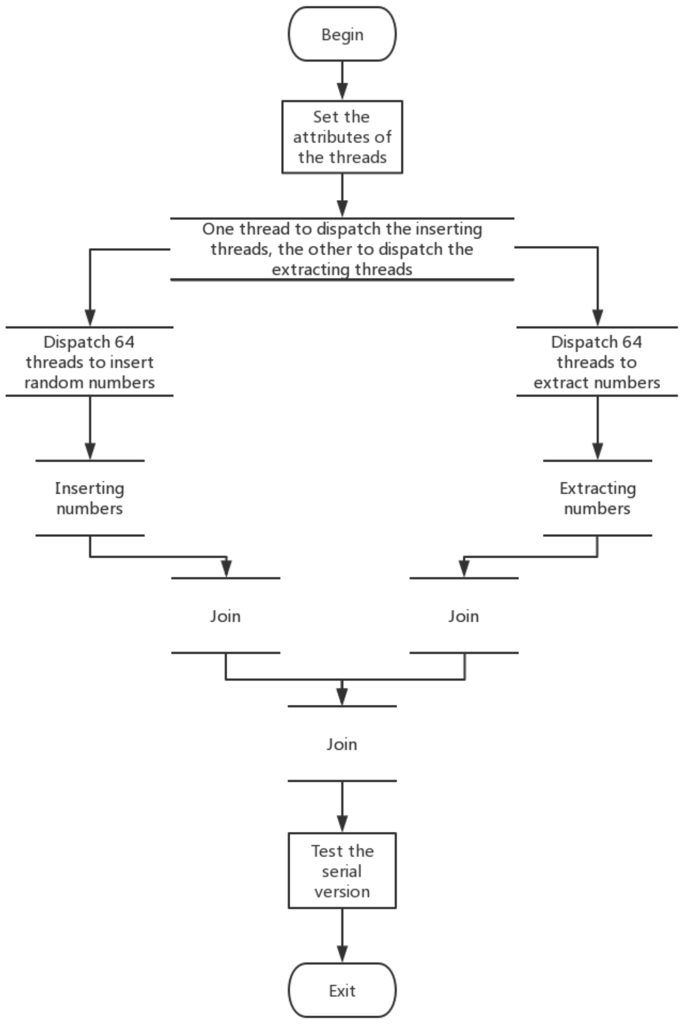
Begin
Set the attributes of the threads
One thread to dispatch the inserting threads, the other to dispatch the extracting threads
Dispatch 64 threads to insert random numbers
Dispatch 64 threads to extract numbers
Inserting numbers
Extracting numbers
Join
Join
Join
Test the serial version
Exit

Figure 1: Flow Chart

The program here used the fork-join mode, and the tricknesses here are:

1. The argument pointers passed to the new threads should point to independent memory, otherwise there will be a mess. So I choose the way that

"new" a temporary variable $\rightarrow$ copy the variable expected to pass $\rightarrow$ pass the temporary variable $\rightarrow$ delete it in new thread.

2. In order to divide a loop to multiple threads, the loop control variables should be initialized differently and cover all the loop range. The method I used here is dividing the loop range into 64 equal parts, except for the last one.

3. If a thread want to extract a element from the queue, it must wait until the queue is not empty. I use a spin wait here.

3 Result

The results that the programs run in my 4-core, 8-thread computer, are shown in table 1 and 2, the times of insertions and extractions vary from 1000 to 100000000.

Table 1: Results of Parallel Version

Number of operation	Time(s)
1000	0.020
10000	0.024
100000	0.037
1000000	0.447
10000000	5.350
100000000	41.715

Table 2: Results of Serial Version

Number of operation	Time(s)
1000	0.000
10000	0.002
100000	0.022
1000000	0.272
10000000	1.453
100000000	25.035

4 Conclusion

Obviously and surprisingly that the parallel version of program is much slower. But the reason is also obvious:

1. The parallel spends too much time scheduling the processes, that is because the number of threads is 130 but my computer has only 8 threads at all.

2. The task per thread does is easy and low load in comparison.

3. The CPU resource is wasted in spin wait, and thus the efficiency is lower.