Multithreaded data structures for parallel computing

Part 1, Designing concurrent data structures

Arpan Sen (arpansen@gmail.com)
Independent author

17 May 2011

Everyone is talking about *parallel computing*: It's all the rage. In this first article of a two-part series on multithreaded structures, learn how to design concurrent data structures in a multithreaded environment using the POSIX library.

Introduction

So, your computer now has four CPU cores; *parallel computing* is the latest buzzword, and you are keen to get into the game. But parallel computing is more than just using mutexes and condition variables in random functions and methods. One of the key tools that a `C++` developer must have in his or her arsenal is the ability to design concurrent data structures. This article, the first in a two-part series, discusses the design of concurrent data structures in a multithreaded environment. For this article, you'll use the POSIX Threads library (also known as Pthreads; see Resources for a link), but implementations such as Boost Threads (see Resources for a link) can also be used.

This article assumes that you have a basic knowledge of fundamental data structures and some familiarity with the POSIX Threads library. You should have a basic understanding of thread creation, mutexes, and condition variables, as well. From the Pthreads stable, you'll be using `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_wait`, `pthread_cond_signal`, and `pthread_cond_broadcast` rather heavily throughout the examples presented.

Designing a concurrent queue

Begin by extending one of the most basic data structures: the queue. Your queue is based on a linked list; the interface of the underlying list is based on the Standard Template Library (STL; see Resources). Multiple threads of control can simultaneously try to push data to the queue or remove data, so you need a mutex object to manage the synchronization. The constructor and destructor of the queue class are responsible for the creation and destruction of the mutex, as shown in Listing 1.

Listing 1. Linked list and mutex-based concurrent queue

```
#include <pthread.h>
```

```
#include <list.h> // you could use std::list or your implementation

namespace concurrent {
template <typename T>
class Queue {
public:
   Queue( ) {
       pthread_mutex_init(&_lock, NULL);
    }
    ~Queue( ) {
       pthread_mutex_destroy(&_lock);
    }
    void push(const T& data);
    T pop( );
private:
    list<T> _list;
    pthread_mutex_t _lock;
}

};
```

Inserting data into and deleting data from a concurrent queue

Clearly, pushing data into the queue is akin to appending data to the list, and this operation must be guarded by mutex locks. But what happens if multiple threads intend to append data to the queue? The first thread locks the mutex and appends data to the queue, while the other threads wait for their turn. The operating system decides which thread adds the data next in the queue, once the first thread unlocks/releases the mutex. Usually, in a Linux® system with no real time priority threads, the thread waiting the longest is the next to wake up, acquire the lock, and append the data to the queue. Listing 2 shows the first working version of this code.

Listing 2. Pushing data to the queue

```
void Queue<T>::push(const T& value ) {
       pthread_mutex_lock(&_lock);
       _list.push_back(value);
       pthread_mutex_unlock(&_lock);
}
```

The code for popping data out is similar, as Listing 3 shows.

Listing 3. Popping data from the queue

```
T Queue<T>::pop( ) {
       if (_list.empty( )) {
           throw ”element not found”;
       }
       pthread_mutex_lock(&_lock);
       T _temp = _list.front( );
       _list.pop_front( );
       pthread_mutex_unlock(&_lock);
       return _temp;
}
```

To be fair, the code in Listing 2 and Listing 3 works fine. But consider this situation: You have a long queue (maybe in excess of 100,000 elements), and there are significantly more threads reading data out of the queue than those appending data at some point during code execution. Because you're sharing the same mutex for push and pop operations, the data-read speed is somewhat compromised as writer threads access the lock. What about using two locks? One for the read operation and another for the write should do the trick. Listing 4 shows the modified `Queue` class.

Listing 4. Concurrent queue with separate mutexes for read and write operations

```
template <typename T>
class Queue {
public:
   Queue( ) {
       pthread_mutex_init(&_rlock, NULL);
       pthread_mutex_init(&_wlock, NULL);
    }
    ~Queue( ) {
       pthread_mutex_destroy(&_rlock);
       pthread_mutex_destroy(&_wlock);
    }
    void push(const T& data);
    T pop( );
private:
    list<T> _list;
    pthread_mutex_t _rlock, _wlock;
}
```

Listing 5 shows the push/pop method definitions.

Listing 5. Concurrent queue Push/Pop operations with separate mutexes

```
void Queue<T>::push(const T& value ) {
       pthread_mutex_lock(&_wlock);
       _list.push_back(value);
       pthread_mutex_unlock(&_wlock);
}

T Queue<T>::pop( ) {
       if (_list.empty( )) {
           throw ”element not found”;
       }
       pthread_mutex_lock(&_rlock);
       T _temp = _list.front( );
       _list.pop_front( );
       pthread_mutex_unlock(&_rlock);
       return _temp;
}
```

Designing a concurrent blocking queue

So far, if a reader thread wanted to read data from a queue that had no data, you just threw an exception and moved on. This may not always be the desired approach, however, and it is likely that the reader thread might want to wait or block itself until the time data becomes available. This kind of queue is called a *blocking queue*. How does the reader keep waiting once it realizes the queue is empty? One option is to poll the queue at regular intervals. But because that approach does not guarantee the availability of data in the queue, it results in potentially wasting a lot of CPU cycles. The recommended method is to use condition variables—that is, variables of type `pthread_cond_t`. Before delving more deeply into the semantics, let's look into the modified queue definition, shown in Listing 6.

Listing 6. Concurrent blocking queue using condition variables

```
template <typename T>
class BlockingQueue {
public:
   BlockingQueue ( ) {
       pthread_mutex_init(&_lock, NULL);
       pthread_cond_init(&_cond, NULL);
    }
    ~BlockingQueue ( ) {
       pthread_mutex_destroy(&_lock);
       pthread_cond_destroy(&_cond);
    }
    void push(const T& data);
    T pop( );
private:
    list<T> _list;
    pthread_mutex_t _lock;
    pthread_cond_t _cond;
}
```

Listing 7 shows the modified version of the pop operation for the blocking queue.

Listing 7. Popping data from the queue

```
T BlockingQueue<T>::pop( ) {
       pthread_mutex_lock(&_lock);
       if (_list.empty( )) {
           pthread_cond_wait(&_cond, &_lock) ;
       }
       T _temp = _list.front( );
       _list.pop_front( );
       pthread_mutex_unlock(&_lock);
       return _temp;
}
```

Instead of throwing an exception when the queue is empty, the reader thread now blocks itself on the condition variable. Implicitly, `pthread_cond_wait` will also release the `mutex _lock`. Now, consider this situation: There are two reader threads and an empty queue. The first reader thread locked the mutex, realized that the queue is empty, and blocked itself on `_cond`, which implicitly released the mutex. The second reader thread did an encore. Therefore, at the end of it all, you now have two reader threads, both waiting on the condition variable, and the mutex is unlocked.

Now, look into the definition of the `push()` method, shown in Listing 8.

Listing 8. Pushing data in the blocking queue

```
void BlockingQueue <T>::push(const T& value ) {
       pthread_mutex_lock(&_lock);
       const bool was_empty = _list.empty( );
       _list.push_back(value);
       pthread_mutex_unlock(&_lock);
       if (was_empty)
           pthread_cond_broadcast(&_cond);
}
```

If the list were originally empty, you call `pthread_cond_broadcast` to post push data into the list. Doing so awakens all the reader threads that were waiting on the condition variable `_cond`; the reader threads now implicitly compete for the mutex lock as and when it is released. The operating system scheduler determines which thread gets control of the mutex next—typically, the reader thread that has waited the longest gets to read the data first.

Here are a couple of the finer aspects of the concurrent blocking queue design:

- Instead of `pthread_cond_broadcast`, you could have used `pthread_cond_signal`. However, `pthread_cond_signal` unblocks at least one of the threads waiting on the condition variable, not necessarily the reader thread with the longest waiting time. Although the functionality of the blocking queue is not compromised with this choice, use of `pthread_cond_signal` could potentially lead to unacceptable wait times for some reader threads.
- Spurious waking of threads is possible. Hence, after waking up the reader thread, verify that the list is not empty, and then proceed. Listing 9 shows the slightly modified version of the `pop()` method, and it is strongly recommended that you use the `while` loop-based version of `pop()`.

Listing 9. Popping data from the queue with tolerance for spurious wake-ups

```
T BlockingQueue<T>::pop( ) {
pthread_cond_wait(&_lock) ; //need writer(s) to acquire and pend on the condition
while(_list.empty( )) {
pthread_cond_wait(&_cond,&_lock) ;
}
T _temp = _list.front( );
_list.pop_front( );
pthread_mutex_unlock(&_lock);
return _temp;
}
```

Designing a concurrent blocking queue with timeouts

There are plenty of systems that, if they can't process new data within a certain period of time, do not process the data at all. A good example is a news channel ticker displaying live stock prices from a financial exchange with new data arriving every *n* seconds. If some previous data could not be processed within *n* seconds, it makes good sense to discard that data and display the latest information. Based on this idea, let's look at the concept of a concurrent queue where push and pop operations come with timeouts. This means that if the system could not perform the push or pop operation within the time limit you specify, the operation should not execute at all. Listing 10 shows the interface.

Listing 10. Concurrent queue with time-bound push and pop operations

```
template <typename T>
class TimedBlockingQueue {
public:
   TimedBlockingQueue ( );
    ~TimedBlockingQueue ( );
    bool push(const T& data, const int seconds);
    T pop(const int seconds);
private:
    list<T> _list;
    pthread_mutex_t _lock;
    pthread_cond_t _cond;
}
```

Let's begin with the timed `push()` method. Now, the `push()` method doesn't depend on any condition variables, so no extra waiting there. The only reason for the delay could be that there are too many writer threads, and sufficient time has elapsed before a lock could be acquired. So, why don't you increase the writer thread priority? The reason is that increasing writer thread priority does not solve the problem if all writer threads have their priorities increased. Instead, consider creating a few writer threads with higher scheduling priorities, and hand over data to those threads that should always be pushed into the queue. Listing 11 shows the code.

Listing 11. Pushing data in the blocking queue with timeouts

```
bool TimedBlockingQueue <T>::push(const T& data, const int seconds) {
       struct timespec ts1, ts2;
       const bool was_empty = _list.empty( );
       clock_gettime(CLOCK_REALTIME, &ts1);
       pthread_mutex_lock(&_lock);
       clock_gettime(CLOCK_REALTIME, &ts2);
       if ((ts2.tv_sec – ts1.tv_sec) <seconds) {
       was_empty = _list.empty( );
       _list.push_back(value);
       {
       pthread_mutex_unlock(&_lock);
       if (was_empty)
           pthread_cond_broadcast(&_cond);
}
```

The `clock_gettime` routine returns in a structure `timespec` the amount of time passed since epoch (for more on this, see Resources). This routine is called twice—before and after mutex acquisition—to determine whether further processing is required based on the time elapsed.

Popping data with timeouts is more involved than pushing; note that the reader thread is waiting on the condition variable. The first check is similar to `push()`. If the timeout has occurred before the reader thread could acquire the mutex, then no processing need be done. Next, the reader thread needs to ensure (and this is the second check you perform) that it does not wait on the condition variable any more than the specified timeout period. If not awake otherwise, at the end of the timeout, the reader needs to wake itself up and release the mutex.

With this background, let's look at the function `pthread_cond_timedwait`, which you use for the second check. This function is similar to `pthread_cond_wait`, except that the third argument is the absolute time value until which the reader thread is willing to wait before it gives up. If the reader thread is awakened before the timeout, the return value from `pthread_cond_timedwait` will be `0`. Listing 12 shows the code.

Listing 12. Popping data from the blocking queue with timeouts

```
T TimedBlockingQueue <T>::pop(const int seconds) {
       struct timespec ts1, ts2;
       clock_gettime(CLOCK_REALTIME, &ts1);
       pthread_mutex_lock(&_lock);
       clock_gettime(CLOCK_REALTIME, &ts2);

       // First Check
       if ((ts1.tv_sec – ts2.tv_sec) < seconds) {
           ts2.tv_sec += seconds; // specify wake up time
           while(_list.empty( ) && (result == 0)) {
               result = pthread_cond_timedwait(&_cond, &_lock, &ts2) ;
           }
           if (result == 0) { // Second Check
               T _temp = _list.front( );
              _list.pop_front( );
              pthread_mutex_unlock(&_lock);
              return _temp;
          }
      }
      pthread_mutex_unlock(&lock);
      throw “timeout happened”;
}
```

The `while` loop in Listing 12 ensures that spurious wake-ups are handled properly. Finally, on some Linux systems, `clock_gettime` may be a part of librt.so, and you may need to append the `–lrt` switch to the compiler command line.

Using the pthread_mutex_timedlock API

One of the sore points in Listing 11 and Listing 12 is that when the thread finally does manage to get access to the lock, there may already be a timeout. So, all it will do is release the lock. You can further optimize this situation by using `pthread_mutex_timedlock` API, if your system supports it (see Resources). This routine takes in two arguments, the second being the absolute value of time by which, if the lock could not be acquired, the routine returns with a non-zero status. Using this routine can therefore reduce the number of waiting threads in the system. Here's the routine declaration:

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
       const struct timespec *abs_timeout);
```

Designing a concurrent blocking bounded queue

Let's end with a discussion on concurrent blocking bounded queues. This queue type is similar to a concurrent blocking queue except that the size of the queue is bounded. There are many embedded systems in which memory is limited, and there's a real need for queues with bounded sizes.

In a blocking queue, only the reader thread needs to wait when there is no data in the queue. In a bounded blocking queue, the writer thread also needs to wait if the queue is full. The external interface resembles that of the blocking queue, as the code in Listing 13 shows. (Note the choice of a vector instead of a list. You could use a basic `C/C++` array and initialize it with size, as appropriate.)

Listing 13. Concurrent bounded blocking queue

```
template <typename T>
class BoundedBlockingQueue {
public:
   BoundedBlockingQueue (int size) : maxSize(size) {
       pthread_mutex_init(&_lock, NULL);
       pthread_cond_init(&_rcond, NULL);
       pthread_cond_init(&_wcond, NULL);
       _array.reserve(maxSize);
    }
    ~BoundedBlockingQueue ( ) {
       pthread_mutex_destroy(&_lock);
       pthread_cond_destroy(&_rcond);
       pthread_cond_destroy(&_wcond);
    }
    void push(const T& data);
    T pop( );
private:
    vector<T> _array; // or T* _array if you so prefer
    int maxSize;
    pthread_mutex_t _lock;
    pthread_cond_t _rcond, _wcond;
}
```

Before explaining the push operation, however, take a look at the code in Listing 14.

Listing 14. Pushing data to the bounded blocking queue

```
void BoundedBlockingQueue <T>::push(const T& value ) {
       pthread_mutex_lock(&_lock);
       const bool was_empty = _array.empty( );
       while (_array.size( ) == maxSize) {
           pthread_cond_wait(&_wcond, &_lock);
       }
       _ array.push_back(value);
      pthread_mutex_unlock(&_lock);
      if (was_empty)
          pthread_cond_broadcast(&_rcond);
}
```

Is the general idea of locking extensible to other data structures?

Sure. But is that optimal? No, it isn't. Consider a linked list, which should be amenable to use by multiple threads. Unlike a queue, a list does not have a single insertion or deletion point, and using a single mutex to control access to the list results in a functional but rather slow system. Yet another implementation choice is to use locks on a per-node basis, but this would definitely increase the memory footprint of the system. The second part of this series will discuss some of these issues.

The first thing of note in Listing 13 and Listing 14 is that there are two condition variables instead of the one that the blocking queue had. If the queue is full, the writer thread waits on the `_wcond` condition variable; the reader thread will need a notification to all threads after consuming data from the queue. Likewise, if the queue is empty, the reader thread would wait on the `_rcond` variable, and a writer thread sends a broadcast to all threads waiting on `_rcond` after inserting data into the queue. What happens when there are no threads waiting on `_wcond` or `_rcond` but broadcast notifications? The good news is that nothing happens; the system just ignores these messages. Also note that both condition variables use the same mutex. Listing 15 shows the code for the `pop()` method in a bounded blocking queue.

Listing 15. Popping data from the bounded blocking queue

```
T BoundedBlockingQueue<T>::pop( ) {
       pthread_mutex_lock(&_lock);
       const bool was_full = (_array.size( ) == maxSize);
       while(_array.empty( )) {
           pthread_cond_wait(&_rcond, &_lock) ;
       }
       T _temp = _array.front( );
       _array.erase( _array.begin( ));
       pthread_mutex_unlock(&_lock);
       if (was_full)
           pthread_cond_broadcast(&_wcond);
       return _temp;
}
```

Note that you've invoked `pthread_cond_broadcast` after releasing the mutex. This is good practice, because the waiting time of the reader thread is reduced after wake-up.

Conclusion

This installment discussed quite a few types of concurrent queues and their implementations. Indeed, further variations are possible. For example, consider a queue in which reader threads are allowed to consume data only after a certain time delay from insertion. Be sure to check the Resources section for details on POSIX threads and concurrent queue algorithms.

Resources

Learn

- Read a good introduction to Pthreads.
- Learn more about the POSIX Thread library.
- Check out Avoiding memory leaks with POSIX threads (Wei Dong Xie, developerWorks, August 2010) to learn more about Pthread programming.
- Learn more about concurrent queue algorithms.
- Find more information on clock time routines.
- Learn more about time locking with mutexes.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- Learn more about and download the Boost Thread library.
- Learn more about and download the Standard Template Library.

Discuss

- Follow developerWorks on Twitter.
- developerWorks blogs: Check out our blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - AIX 5L—technical forum
  - AIX for Developers Forum
  - Cluster Systems Management
  - IBM Support Assistant
  - Performance Tools—technical
  - More AIX and UNIX forums

About the author

Arpan Sen

Arpan Sen is a lead engineer working on the development of software in the electronic design automation industry. He has worked on several flavors of UNIX, including Solaris, SunOS, HP-UX, and IRIX as well as Linux and Microsoft Windows for several years. He takes a keen interest in software performance-optimization techniques, graph theory, and parallel computing. Arpan holds a post-graduate degree in software systems. You can reach him at arpansen@gmail.com.