Multithreaded data structures for parallel computing: Part 2, Designing concurrent data structures without mutexes

Arpan Sen (arpansen@gmail.com)
Independent author

24 May 2011

In this second article in a two-part series on multithreaded structures, learn about design choices for implementing a mutex based concurrent list, and discover how to design concurrent data structures without mutexes.

View more content in this series

Introduction

This article—the concluding part in this series—discusses two things: Design choices for implementing a mutex based concurrent list and designing concurrent data structures without mutexes. For the latter topic, I have chosen to implement a concurrent stack and highlight some of the issues in designing such a data structure. Designing a mutex-free data structure in `C++` that is platform independent is not a reality yet, so I chose GCC version 4.3.4 as the compiler and used GCC-specific `__sync_*` functions in the code. If you're a WIndows®`C++` developer, consider the Interlocked* group of functions for similar work.

Design choices in a concurrent, singly linked list

Listing 1 shows the most basic concurrent, singly linked list interface. Is anything obviously missing?

Listing 1. A concurrent, singly linked list interface

```
template <typename T>
class SList {
   private:
       typedef struct Node {
            T data;
            Node *next;
            Node(T& data) : value(data), next(NULL) { }
       } Node;
       pthread_mutex_t _lock;
       Node *head, *tail;
   public:
       void push_back(T& value);
       void insert_after(T& previous, T& value); // insert data after previous
       void remove(const T& value);
       bool find(const T& value);  // return true on success
       SList(  );
       ~SList(  );
};
```

For the expected lines, Listing 2 shows the `push_back` method definition.

Listing 2. Pushing data into the concurrent linked list

```
void SList<T>::push_back(T& data)
{
    pthread_mutex_lock(&_lock);
    if (head == NULL) {
        head = new Node(data);
        tail = head;
    } else {
        tail->next = new Node(data);
        tail = tail->next;
    }
    pthread_mutex_unlock(&_lock);
}
```

Now, consider a thread trying to push *n* integers into this list in quick succession by calling `push_back`. The interface itself mandates that you acquire and release the mutex *n* times, even if all data to be inserted is known before acquiring the lock for the first time. A far better approach would be to define another method that accepts a list of integers and acquire and release the mutex only once. Listing 3 shows the method definition.

Listing 3. Appending to the linked list intelligently

```
void SList<T>::push_back(T* data, int count) // or use C++ iterators
{
    Node *begin = new Node(data[0]);
    Node *temp = begin;
    for (int i=1; i<count; ++i) {
        temp->next = new Node(data[i]);
        temp = temp->next;
    }

    pthread_mutex_lock(&_lock);
    if (head == NULL) {
        head = begin;
        tail = head;
    } else {
        tail->next = begin;
        tail = temp;
    }
    pthread_mutex_unlock(&_lock);
}
```

Optimizing search elements

Now, let's move on to optimizing search elements in the list—that is, the `find` method. Here are a few potential situations that may occur:

- Insertion or deletion requests come in while some threads are iterating over the list.
- Iteration requests come in while some threads are iterating the list.
- Iteration requests come in while some threads are inserting data into or deleting data from the list.

Clearly, you should be able to service multiple iteration requests concurrently. For a system where the insertion/deletion rate is minimal and the primary activity consists of searching, having a single lock-based approach is way below par. In this context, get to know read-write locks or `pthread_rwlock_t`. In the examples in this article, you'll use `pthread_rwlock_t` in `SList` instead of `pthread_mutex_t`. Doing so allows for multiple threads to search the list concurrently. Insertion and deletion would still lock the whole list, which is fine anyway. Listing 4 shows some of the list implementation with `pthread_rwlock_t` followed by the code for `find`.

Listing 4. A concurrent, singly linked list using the read-write lock

```
template <typename T>
class SList {
   private:
       typedef struct Node {
           // … same as before
       } Node;
       pthread_rwlock_t _rwlock; // Not pthread_mutex_t any more!
       Node *head, *tail;
   public:
       // … other API remain as-is
       SList(  ) : head(NULL), tail(NULL) {
           pthread_rwlock_init(&_rwlock, NULL);
       }
       ~SList(  ) {
           pthread_rwlock_destroy(&_rwlock);
           // … now cleanup nodes
       }
};
```

Listing 5 shows the code for the list search.

Listing 5. Searching the linked list using read-write lock

```
bool SList<T>::find(const T& value)
{
    pthread_rwlock_rdlock (&_rwlock);
    Node* temp = head;
    while (temp) {
         if (temp->value == data) {
              status = true;
              break;
         }
         temp = temp->next;
    }
    pthread_rwlock_unlock(&_rwlock);
    return status;
}
```

While Listing 6 shows `push_back` using the read-write lock.

Listing 6. Pushing data into the concurrent, linked list using read-write lock

```
void SList<T>::push_back(T& data)
{
    pthread_setschedprio(pthread_self( ), SCHED_FIFO);
    pthread_rwlock_wrlock(&_rwlock);
    // … All the code here is same as Listing 2
    pthread_rwlock_unlock(&_rwlock);
}
```

Let's take stock of things. You have used two locking function calls—`pthread_rwlock_rdlock` and `pthread_rwlock_wrlock`—for synchronization and a call to `pthread_setschedprio` to set the priority of writer threads. If no writer threads are blocked on this lock (in other words, no insertion/deletion requests), then multiple reader threads requesting list search can concurrently operate, because one reader thread would not block another reader thread in such a situation. If writer threads are waiting on this lock, then of course no new reader thread is allowed to acquire the lock, and the threads wait until the existing reader threads have finished, followed by the writer threads. If you don't adhere to this approach in prioritizing writer threads using `pthread_setschedprio`, then given the nature of read-write lock, it is easy to see how writer threads could starve.

Here are a few things to remember with this approach:

- `pthread_rwlock_rdlock` may fail if the maximum number of read locks (implementation defined) for the lock has been exceeded.
- Take care to invoke `pthread_rwlock_unlock`*n* times if there are *n* concurrent read locks for the lock.

Allowing concurrent insertions

The last method you should learn is `insert_after`. Once again, the expected usage pattern governs your decision to tweak the data structure. If the application begins with a pre-provided linked list that has an almost equal number of insertions and searches but minimal deletions, then it's not prudent to lock the entire list during insertion. Allowing for concurrent insertions at disjoint

points in the list is a good idea in such a case, and you use the read-write-lock based approach again. Here's how you structure the list:

- Locking occurs on two levels (see Listing 7): The list has a read-write lock, while individual nodes contain a mutex. If space-saving is what you're looking for, then consider a plan to share mutexes—maybe maintain a map of nodes versus mutexes.
- During insertion, the writer thread makes a read lock on the list and proceeds. The individual node, after which the new data is to be added, is locked before the insertion and released after insertion followed by releasing the read-write lock.
- Deleting creates a write lock on the list. No node-specific lock needs to be acquired.
- Searching can be done concurrently, as earlier.

Listing 7. Concurrent, singly linked list with two-level locking

```
template <typename T>
class SList {
   private:
       typedef struct Node {
            pthread_mutex_lock lock;
            T data;
            Node *next;
            Node(T& data) : value(data), next(NULL) {
                pthread_mutex_init(&lock, NULL);
            }
            ~Node( ) {
                pthread_mutex_destroy(&lock);
            }
       } Node;
       pthread_rwlock_t _rwlock; // 2 level locking
       Node *head, *tail;
   public:
       // … all external API remain as-is
       }
};
```

Listing 8 shows the code for inserting data into the list.

Listing 8. Inserting data into the list with double-locking

```
void SList<T>:: insert_after(T& previous, T& value)
{
    pthread_rwlock_rdlock (&_rwlock);
    Node* temp = head;
    while (temp) {
         if (temp->value == previous) {
              break;
         }
         temp = temp->next;
    }
    Node* newNode = new Node(value);

    pthread_mutex_lock(&temp->lock);
    newNode->next = temp->next;
    temp->next = newNode;
    pthread_mutex_unlock(&temp->lock);

    pthread_rwlock_unlock(&_rwlock);
    return status;
}
```

The problem with a mutex-based approach

So far, you have used a mutex or multiple mutexes included as part of the data structure for synchronization. This approach is not without its problems, however. Consider the following situations:

- Waiting on mutexes consumes precious time—sometimes, a lot of time. This delay has negative effects on system scalability.
- Lower-priority threads could acquire a mutex, thus halting higher-priority threads that require the same mutex to proceed. This problem is known as *priority inversion* (see Resources for links to more information).
- A thread holding a mutex can be de-scheduled, perhaps because it was the end of its time-slice. For other threads waiting on the same mutex, this has a negative effect, because the wait time is now even longer. This problem is known as *lock convoying* (see Resources for links to more information).

The issues with mutexes don't end here. In recent times, solutions that don't use mutexes have been coming up. That said, although mutexes are tricky to use, they are definitely worth your attention if you're looking for better performance.

The compare and swap instruction

Before we move on to solutions that don't involve mutexes, let's pause for a moment and look into the CMPXCHG assembly instruction available on all Intel® processors starting with 80486. From a conceptual standpoint, Listing 9 shows what the instruction does.

Listing 9. Compare and swap instruction behavior

```
int compare_and_swap ( int *memory_location, int expected_value, int new_value)
{
  int old_value = *memory_location;
  if (old_value == expected_value)
     *memory_location = new_value;
  return old_value;
}
```

What's happening here is that the instruction is checking whether a memory location has an expected value; if it does, then the new value is copied into the location. From an assembly language perspective, Listing 10 provides the pseudo-code.

Listing 10. The compare and swap instruction assembly pseudo-code

```
CMPXCHG OP1, OP2
if ({AL or AX or EAX} = OP1)
       zero = 1               ;Set the zero flag in the flag register
       OP1 = OP2
else
       zero := 0              ;Clear the zero flag in the flag register
       {AL or AX or EAX}= OP1
```

The CPU chooses the AL, AX, or EAX register depending on the width of the operand (8, 16, or 32 bits). If the contents of the AL/AX/EAX register matches that of operand 1, then the contents of operand 2 are copied to the first; otherwise, the AL/AX/EAX register is updated with the value

of operand 2. The Intel Pentium® 64-bit processor has a similar instruction named *CMPXCHG8B* that supports 64-bit compare and exchange. Note that the CMPXCHG instruction is *atomic*, which means that there is no intermediate visible state of the system before this instruction finishes. It's either completely executed or not yet started. Equivalent instructions are available on other platforms—for example, the Motorola MC68030 processor has an instruction named *compare and swap* (CAS) that has similar semantics.

Why are we interested in CMPXCHG? Does this mean I would code in assembly?

You need to understand CMPXCHG and related instructions like CMPXCHG8B well, because they form the crux of lock-free solutions. However, you could do without coding in assembly. Thankfully, GCC (GNU Compiler Collection, from version 4.1 onwards) provides atomic built-ins (see Resources) that you can use to implement CAS operations for both x86 and x86-64 platforms. No header file need be included for this support. In this article, you use the GCC built-ins in your implementation of lock-free data structures. Here's a look at the built-ins:

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)
```

The `__sync_bool_compare_and_swap` built-in compares `oldval` with `*ptr`. If they match, it copies `newval` to `*ptr,`. The return value is True if `oldval` and `*ptr` match and False otherwise. The `__sync_val_compare_and_swap` built-in's behavior is similar, except that it always returns the old value. Listing 11 provides a sample usage.

Listing 11. Sample usage of GCC CAS built-ins

```
#include <iostream>
using namespace std;

int main()
{
   bool lock(false);
   bool old_value = __sync_val_compare_and_swap( &lock, false, true);
   cout >> lock >> endl; // prints 0x1
   cout >> old_value >> endl; // prints 0x0
}
```

Designing a lock-free concurrent stack

Now that you have some understanding of CAS, let's design a concurrent stack. No locks will be included; this kind of lock-free, concurrent data structure is also referred to as a *non-blocking data structure*. Listing 12 provides the code interface.

Listing 12. Linked list-based implementation of a non-blocking stack

```
template <typename T>
class Stack {
    typedef struct Node {
                          T data;
                          Node* next;
                          Node(const T& d) : data(d), next(0) { }
                        } Node;
    Node *top;
    public:
       Stack( ) : top(0) { }
       void push(const T& data);
       T pop( ) throw (…);
};
```

Listing 13 shows the Push operation.

Listing 13. Pushing data in a non-blocking stack

```
void Stack<T>::push(const T& data)
{
    Node *n = new Node(data);
    while (1) {
        n->next = top;
        if (__sync_bool_compare_and_swap(&top, n->next, n)) { // CAS
            break;
        }
    }
}
```

What's going on with the Push operation? From the standpoint of a single thread, a new node is created whose next pointer points to the top of the stack. Next, you invoke CAS and copy the new node to the top location.

From the standpoint of multiple threads, it is entirely possible that two or more threads were simultaneously trying to push data into the stack. Say you have Thread A trying to push 20 and Thread B trying to push 30 into the stack, and Thread A got the time slice first. Thread A also got de-scheduled after the instruction `n->next = top` finished. Now, Thread B (and a lucky thread this is) got into action, was able to complete CAS, and finished by pushing 30 into the stack. Next, Thread A resumes, and clearly `*top` and `n->next` do not match for this thread, because Thread B modified the contents of the top location. So, the code loops back, points to the proper top pointer (which was changed because of Thread B), invokes CAS, and is done with pushing 20 into the stack. All of this was done without any locks.

Now for the Pop operation

Listing 14 shows the code for popping elements off the stack.

Listing 14. Popping data from a non-blocking stack

```
T Stack<T>::pop( )
{
    if (top == NULL)
       throw std::string(“Cannot pop from empty stack”);
    while (1) {
        Node* next = top->next;
        if (__sync_bool_compare_and_swap(&top, top, next)) { // CAS
            return top->data;
        }
    }
}
```

You define Pop operation semantics along similar lines to `push`. The top of the stack is stored in `result`, and you use CAS to update the top location with `top-<next` and return the appropriate data. If there was thread preemption just before CAS, after resumption of the thread, CAS would fail, and the looping would continue until valid data were available.

All's well that ends well

Unfortunately, there are problems with the pop implementation of the stack—both of the obvious and the non-obvious variety. The obvious issue is that the NULL check must be part of the `while` loop. If Thread P and Thread Q are both trying to pop data from a stack that has only one element left and Thread P is de-scheduled just before CAS, by the time it regains control, there isn't anything left to pop. Because `top` would be NULL, accessing `&top` is a sure-fire way to crash—clearly an avoidable bug. This problem also highlights one of the fundamental design principles when it comes to working with parallel data structures: Do not assume sequential execution of any code, ever.

Listing 15 shows the code with the obvious bug fix.

Listing 15. Popping data from a non-blocking stack

```
T Stack<T>::pop( )
{
    while (1) {
        if (top == NULL)
           throw std::string(“Cannot pop from empty stack”);
        Node* next = top->next;
        if (top && __sync_bool_compare_and_swap(&top, top, next)) { // CAS
            return top->data;
        }
    }
}
```

The next problem is somewhat more complicated, but if you understand how memory managers work (see Resources for links to more information), this shouldn't be too difficult. Listing 16 shows the problem.

Listing 16. Recycling of memory can cause serious issues with CAS

```
T* ptr1 = new T(8, 18);
T* old = ptr1;
// .. do stuff with ptr1
delete ptr1;
T* ptr2 = new T(0, 1);

// We can't guarantee that the operating system will not recycle memory
// Custom memory managers recycle memory often
if (old1 == ptr2) {
   …
}
```

In this code, you can't guarantee that `old` and `ptr2` will have different values. Depending on the operating system and the custom application memory management system, it is entirely likely that the deleted memory is recycled—that is, the deleted memory is stored in specialized pools for the application to reuse if needed and not returned to the system. This obviously improves performance, because you don't need to go through system calls to request additional memory. Now, although this is generally a good thing to have, let's see why it isn't such good news to the non-blocking stack.

Suppose you have two threads—A and B. A called `pop` and was de-scheduled just before CAS. B then called `pop` and pushed in data, one part of which was from recycled memory from the earlier Pop operation. Listing 17 shows the pseudo-code.

Listing 17. A sequence diagram

```
Thread A tries to pop
Stack Contents: 5 10 14 9 100 2
result = pointer to node containing 5
Thread A now de-scheduled

Thread B gains control
Stack Contents: 5 10 14 9 100 2
Thread B pops 5
Thread B pushes 8 16 24 of which 8 was from the same memory that earlier stored 5
Stack Contents: 8 16 24 10 14 9 100 2

Thread A gains control
At this time, result is still a valid pointer and *result = 8
But next points to 10, skipping 16 and 24!!!
```

The fix is reasonably simple: Don't store the next node. Listing 18 shows the code.

Listing 18. Popping data from a non-blocking stack

```
T Stack<T>::pop( )
{
    while (1) {
        Node* result = top;
        if (result == NULL)
           throw std::string(“Cannot pop from empty stack”);
        if (top && __sync_bool_compare_and_swap(&top, result, result->next)) { // CAS
            return top->data;
        }
    }
}
```

With this arrangement, even if Thread B has modified the top while Thread A tries to pop, you're sure that no elements in the stack are skipped.

Summary

This series provided insights to the world of designing data structures that are amenable to concurrent access. You have seen that the design choices could be mutex based or lock-free. Either way, both require ways of thinking that go beyond the traditional functionality of these data structures—in particular, you always need to keep in mind preemption and how the thread resumes when it is rescheduled. The solutions—particularly on the lock-free side of things—are rather platform/compiler specific at this point. Consider looking into the Boost library for an implementation of threads and locks and John Valois's paper on lock-free linked lists (see Resources for links). The `C++0x` standard provides for an `std::thread` class, but support for it has been fairly limited to downright nonexistent in most compilers to date.

Resources

Learn

- MSDN provides good information on priority inversion of threads.
- Learn more about lock convoying.
- Check out Cambridge University's page on practical lock-free data structures.
- Check out John Valois's paper on lock-free linked lists.
- Learn more about memory manager for `C++` (Arpan Sen and Rahul Kumar Kardam, developerWorks, February 2008).
- 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

- Check out GCC atomic built-ins.
- Learn more about and download the Boost Thread 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.