March 28, 2014

`shared_state<T>`, a smart pointer for thread-safe access to `T`

Johan Gustafsson,
johan.b.gustafsson@gmail.com

Abstract

The `shared_state` class is a smart pointer that guarantees thread-safe access to shared states in a multithreaded environment. `shared_state` renders compile-time errors on missing or improper locks and run-time exceptions on lock timeouts, while being generic and straightforward to use with reasonably good performance (include `shared_state.h`, `shared_state_mutex.h` and `shared_timed_mutex_polyfill.h` from the appendix and compile under C++11). It is also possible to extend or customize the behaviour using type traits. Type traits can be used to select different mutex types, to get backtraces on deadlocks, or warnings on locks that are held too long.

1 Motivation

In multi-threaded environments, access to shared resources must be synchronized. A common way to deal with this is by using a *blocking lock* [1]. With a blocking lock a thread locks a lockable object while using a shared resource and unlocks it when finished. Concurrent threads who need to use the same resource will wait (i.e block) until the lock becomes available. This method has quite a few drawbacks, of which most are addressed using a *non-blocking algorithm* in which the program is guaranteed to progress in finite time with little overhead [2]. However, non-blocking algorithms typically rely on hardware support and are limited to synchronize simple constructs whereas a blocking lock can synchronize larger structures and complex resources. Then again, several examples have been made of how larger structures or complex resources can be split into simpler components where a non-blocking algorithm suffices for achieving synchronization [3].

While synchronization introduces both overhead and complexity it should be noted here that programming models without shared resources requires none of these explicit synchronization mechanisms. This applies for instance to functional programming and programming models based on message passing. A computer program may have some parts without shared resources and other parts that use shared resources. For instance, a programming model based on message passing will use an underlying message passing system with shared resources and will thus require synchronization mechanisms for concurrent computing.

A blocking lock is typically implemented as a mutually exclusive lock, or in short a *mutex*. There are alternatives to standard mutexes [4], and variations such as *funnels* and *serializing tokens* are used in some kernels. Only mutexes from the C++ standard library and the Boost library will be considered here.

A *critical section* is a piece of code in which the mutex is locked so that a thread can access a shared resource without running into conflicts with concurrently executing threads. It is in general safe for multiple threads to read from a shared resource as long as no one changes the state of the shared resource. Such read access may instead use a critical section that allow simultaneous access for multiple *readers* while preventing access for any concurrent *writer*.

The mutex is in general is a separate component from the shared resources so it is possible to omit the mutex lock, either deliberately or as a bug, and make unprotected use of the shared resource. The program might then seem to work but fail occasionally when another thread use of the same shared resource simultaneously and end up corrupting the state of the shared resource. The programmer thus need to remember to explicitly enter such a critical section before accessing the resource.

2 The method

This study presents a smart pointer called `shared_state<T>` for ensuring adequate critical sections in C++. Within a critical section for write access the object is accessible through a `T*` pointer, and within a critical section for read-only access the object is accessible through a `const T*` pointer.

Let type `T` represent some state or a handle to some resource. Then type `shared_state<T>` can be said to represent a thread-safe shared mutable state or a shared resource with synchronized access. Listing 1 illustrates such a type:

```
class A {
public:
  void foo();
  void bar() const;
};
```

Listing 1: Example class A

Create an instance of A that is safe to use in concurrent execution like this:

```
shared_state<A> a{new A};
```

and then enter a critical section like this:

```
// Read and write access
{
  auto w = a.write();
  w->foo();
  w->bar();
}

// Read-only access
{
  auto r = a.read();
  r->bar();
}
```

`auto` will resolve to `shared_state<A>::write_ptr` or `shared_state<A>::read_ptr`. These types act as lock guards that block other threads from entering a critical section for the same object. They are also exception safe through RAII, so regardless how you leave the critical section (through `return`, `throw` or normal execution) they will release their lock when they go out of scope.

It is also possible to check whether the lock is available and only enter the critical section if its possible without waiting.

```
// Conditional critical section,
// don't wait for lock
if (auto w = a.try_write())
{
  w->foo();
}
```

One can also enter and leave a critical section in the same statement:

```
a.write()->foo();
a.read()->bar();
```

The critical sections created by `.read()` and `.write()` are built using a *mutex*. The situation where an algorithm spends time on waiting for locks to become available is referred to as *lock contention*. A crucial part of any such algorithm is to keep lock contention low as it will prevent the algorithm from scaling otherwise. The idea is to lock as seldom as possible and not keeping the lock longer than absolutely needed. In the most extreme case this strive to minimize lock contention ends up in using non-blocking algorithm, i.e an algorithm without locks has zero lock contention.

Timeout

By default, `shared_state` will timeout an attempt to enter a critical section after 0,1 seconds and throw a `lock_failed` exception. The timeout happens if the lock is held by another thread and the lock isn't released fast enough. But that other thread may also be waiting for a lock that the first thread is keeping. This is referred to as a *deadlock*, i.e when two threads end up waiting for each other on different locks. A deadlock can only happen if at least two threads attempt to enter at least two critical sections simultaneously, and it can be considered a logical error in the program design if the program design even makes it possible that a deadlock may happen.

```
class A {
   ...
};

template<>
struct shared_state_traits<A>
  : shared_state_traits_default
{
  double timeout() {
    return 0.010;
  }
};
```

Listing 2: Example of non-invasive type traits

```
class A {
public:
  struct shared_state_traits
    : shared_state_traits_default
  {
    double timeout() {
      return 0.010;
    }
  };

  ...
};
```

Listing 3: Example of invasive type traits

The behavior of timeouts can be parameterized using type traits, Listing 2 and Listing 3 show two equivalent ways of doing this. This list outline the properties defined by creating a custom `shared_state_traits`:

1. `double timeout()` specify the time in seconds until an attempt to enter a critical section will fail with a `lock_failed` exception, a negative value disables the timeout and lock attempts will wait indefinitely.

2. `template<class C> void timeout_failed ()` is called whenever a lock attempt fails. The client can use this to deal with failed locks in a custom way instead of throwing a `lock_failed` exception per default.

3. `void was_locked ()` and `void was_unlocked ()` are called whenever a lock is obtained and released, respectively.

4. `typename shared_state_mutex` defines which mutex type to use when protecting instances of the client type.

Both methods inherit from `shared_state_traits_default` to get the standard behavior for most settings. Please refer to `shared_state.h` for more complete documentation, see Appendix.

Thread-safe methods

While the methods presented above may suffice to design the bulk of a thread-safe program one might want to pull a few extra tricks to avoid redundant locks. It is possible to discard the requirement of entering a critical section by using `shared_mutable_state`. Listing 4 show an example that does not obtain any lock, and allows unprotected concurrent access.

It is in general advisable to completely separate instances representing shared states from instances that do not represent shared states, in which case `.read()` and `.write()` should suffice as distinct synchronization points and `shared_mutable_state` will not be needed.

```
shared_state<A> a{new A};

shared_state<A>::shared_mutable_state
     m{a};

m->foo();
m->bar();
```

Listing 4: `shared_mutable_state`

3 Results

The implementation of `shared_state` was tested on a MacBook Pro with Intel Core i7, 2.7 GHz, 4 cores (or 8 threads with hyper-threading) in OS X 10.9.2. The test program was built with clang-503.0.38, libc++ and boost 1.55.0. See Appendix for source code.

A C++14 `std::shared_timed_mutex` provides all the functionality needed by `shared_state`, but it is possible to trade performance for functionality using other mutex types. A mutex can be implemented in a number of ways so it is possible to choose another mutex that is performing well under the given circumstances.

Measurements

Table 1 lists some performance characteristics of using a few different mutex types. This study includes C++11 mutexes from libc++, boost mutexes based on pthreads, and a C++14-compatible shared timed mutex built using C++11 mutexes.

When using mutexes without timeout functionality `shared_state` uses the lock functions without timeout arguments and thus blocks indefinitely until a lock can be acquired. When using mutexes without shared functionality `shared_state` makes all read attempts mutually exclusive, just like write attempts.

The *overhead* measure denote the extra time it takes to call a method through `shared_state` if a lock was not needed. That is, a thread-safe program running on just 1 thread should not take much longer to execute than the same program with all thread-safety precautions removed.

The *try_write* measure denote the time it takes to check if an unavailable lock is available. That is, how much time it takes to call *try_write* if the lock attempt fails.

The *verify* measure uses the extensability of `shared_state` to measure for how long a lock is held. Ideally locks should only be kept for the time it takes to transition the state of the shared resource from one valid state to the next. As this should typically only take a fraction of a millisecond this method can be used to issue warnings if a lock is kept longer than some threshold value. Such functionality could then be disabled in a production build. Listing 5 show such a `shared_state_traits`.

When multiple readers and writers race for the same lock there is *lock contention*, which means that the threads has to wait for each other which in turn decreases efficiency. In these tests 8 threads where running a loop to lock an object and execute some work function (calling `rand()` 100 times or 1000 times respectively). The lock was mostly for shared read access but 1% or 10% of all locks where write locks instead. The lock was kept until the work function returned. Depending on how long time the work function took to complete different mutexes had the best performance. Table 1 list how many calls to the work function that could be completed per 1 ms.

std, timed, shared and *std, shared* denotes a clone of `boost::shared_mutex` built using `std::mutex` as a drop-in replacement of `boost::mutex`. This mutex effectively implements `std::shared_timed_mutex` from C++14 but only requires C++11 which is more widely adopted. *boost, timed, shared* and *boost, shared* both denote `boost::shared_mutex` which support timeouts on lock attempts. Boost mutexes are implemented through pthread mutexes on the test platform.

				1000 `rand()` [calls / ms]		100 `rand()` [calls / ms]	
	overhead	try_write	verify	1% write	10% write	1% write	10% write
std, shared, timed	126 ns	67 ns	358 ns	151	122	163	127
boost, shared, timed	202 ns	70 ns	445 ns	150	123	152	117
std::timed_mutex	112 ns	59 ns	351 ns	130	**130**	126	131
boost::timed_mutex	107 ns	65 ns	333 ns	128	129	93	94
std, shared	127 ns	67 ns	358 ns	**153**	128	168	147
boost, shared	215 ns	69 ns	445 ns	149	120	166	133
std::mutex	**60 ns**	**41 ns**	294 ns	105	105	275	272
boost::mutex	65 ns	48 ns	**286 ns**	104	106	**277**	**273**
without lock				240	240	2169	2169

Table 1: Performance measures of using different mutex types with `shared_state`*, with the best value per column in* ***bold text****.*

4 Discussion

As can be seen in Table 1 it matters which kind of mutex you use. `std::mutex` might seem like the best choice. If the work performed while keeping the lock only takes a few microseconds to complete a `std::mutex` is the fastest solution regardless of the amount of shared read access. But if the work takes more time and there is a high risk for lock contention it might be worth the overhead to instead use a shared mutex.

A program that needs to read from a shared state several times in a function will in general require that state to stay consistent for the duration of the function. Typically the function should first copy the shared state, release its lock on the shared state, and then use its local copy.

If it takes a long time to make that local copy it might makes sense to use a shared mutex. For instance, replacing 1 MB of data in some shared structure should take about 100 000 ns (assuming a RAM speed of 10 GB/s), i.e 10 calls/ms without lock. To be compared with time estimates in the results table. On the other hand, in that case updates should be infrequent, and thus there will be no lock contention which makes the overhead of a shared mutex unnecessary.

Other reasons may also apply as to why the shared state cannot be copied in an instant or why it is important that the shared state is not modified during the course of a function call. But keeping a lock for an extended period of time is a design smell. Even the developer of `boost::shared_mutex` didn't want shared mutexes to be included in C++11 [5], partly due to this reason. However, the developer who wrote `shared_timed_mutex` for C++14 could show that there are viable cases for using shared locks [6][7].

By default `shared_state` uses a shared mutex with support for timeouts and shared read access to make it easy to use for generic states and arbitrary algortihms. It is still possible to use a different mutex type if the usage pattern suggests that it would be more efficient. But an optimization for those cases might benefit more from using a lockfree structure such as those provided by `std::atomic` or Boost.Lockfree. Listing 6 illustrate an example of a custom mutex without support for shared access and without support for timeouts by simply wrapping `std::mutex`.

Another useful customization is to embed a backtrace onto the exception object thrown when a timeout happens. Such a backtrace can for instance help to locate the design bug causing a dead-lock. Listing 7 show a customization using the Backtrace library [8]. When a lock fails `shared_state_traits<MyType>` will keep waiting for a second timeout in an attempt to find all threads participaiting in a deadlock.

To conclude, the `shared_state` class is a smart pointer that guarantees thread-safe access to shared states in a multithreaded environment. `shared_state` renders compile-time errors on missing locks and it is possible to extend the behaviour using type traits to get backtraces on deadlocks, get run-time warnings on locks that are kept too long, and to use custom mutex types. While certainly not ideal for wrapping every and all cases of shared resources `shared_state` is a generic class that is straightforward to use with reasonably good performance.

References

[1] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[2] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. *23rd International Conference on Distributed Computing Systems*, page 522, 2003.

[3] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *Proceedings of the 17ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2012)*, pages 141–150.

[4] Leslie Lamport. A fast mutual exclusion algorithm. *Digital Equipment Corporation.*

[5] Anthony Williams. Boost mailing lists, gmane.comp.lib.boost.devel, re: [thread] on shared_mutex. `http://permalink.gmane.org/gmane.comp.lib.boost.devel/211180`, 29 Nov 11:05 2010.

[6] Howard Hinnant. Performance of std::shared_mutex in C++14. `http://home.roadrunner.com/~hinnant/mutexes/performance.html`.

[7] Howard Hinnant. Using std::shared_mutex in C++14. `http://home.roadrunner.com/~hinnant/mutexes/locking.html`.

[8] Johan Gustafsson. Cross-platform run-time backtraces. `https://github.com/gustafsson/backtrace`, March 28, 2014.

Appendix

The latest version of `shared_state` with the complete source code can be found at:

`https://github.com/gustafsson/backtrace`

```
template<>
struct shared_state_traits<MyType>
: shared_state_traits_default
{
  double timeout() { return 0.001; }

  void was_locked() {
    start = chrono::high_resolution_clock::now ();
  }

  void was_unlocked() {
    chrono::duration<double> diff = chrono::high_resolution_clock::now () - start;
    if (diff.count() > verify_execution_time)
      exceeded_execution_time (diff.count());
  }

  float verify_execution_time = 0.001;
  function<void(double)> exceeded_execution_time = [](double T) {
    cout << "Warning:␣Lock␣of␣MyType␣was␣held␣for␣" << T << "seconds" << endl;
  };
private:
  chrono::high_resolution_clock::time_point start;
};

... {
  shared_state<MyType> m(new MyType);
  ...
  {
    auto w = m.write();
    // work for more than 1 ms
    ...
  } // prints "Warning: Lock of MyType was held for ... seconds"
  ...
... }
```

Listing 5: Custom lock and unlock handler with type traits

```
class mutex_simple
  : public std::mutex
{
public:
  void lock_shared() { lock(); }
  bool try_lock_shared() { return try_lock(); }
  void unlock_shared() { unlock(); }

  bool try_lock_for(...) { lock(); return true; }
  bool try_lock_shared_for(...) { lock_shared(); return true; }
};

template<>
struct shared_state_traits<MyType>
  : shared_state_traits_default
{
  typedef mutex_simple shared_state_mutex;
};

... {
  shared_state<MyType> m(new MyType);

  // This will use mutex_simple
  m.write()->...
... }
```

Listing 6: Custom mutex with type traits

```
template<class T>
class lock_failed_boost
  : public shared_state<T>::lock_failed
  , public virtual boost::exception
{};


template<>
struct shared_state_traits<MyType>
        : shared_state_traits_default
{
  template<class T>
  void timeout_failed () {
    this_thread::sleep_for (chrono::duration<double>{timeout()});

    BOOST_THROW_EXCEPTION(lock_failed_boost<MyType>{} << Backtrace::make ());
  }
};

... {
  shared_state<MyType> m(new MyType);
  ...
  try {
    auto w = m.write();
    ...
  } catch (lock_failed& x) {
    const Backtrace* backtrace = boost::get_error_info<Backtrace::info>(x);
    ...
  }
... }
```

Listing 7: Custom timeout handler with type traits