Design Patterns in C++

Concurrency

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

May 2, 2011

Outline

Threads in boost

- A thread in boost is just an object of class `boost::thread`
- Threads objects are “movable”
  - In other words, threads are not “copied” (with the result of having two threads), but actually “moved”
  - This means that a copy constructor does not “move”, but actually “moves” the ownership from source to destination
  - this allows thread objects to be returned from functions without caring about ownership

```
boost::thread f();
void g()
{
    boost::thread some_thread = f();
    some_thread = join();
}
```

What is a thread

- To create a thread, you have to pass a “callable”:
  - a function
  - a function object
- the callable is copied into the thread object, so it is safe to destroy it afterwards

```
struct callable {
    void operator()() { ... }
};

boost::thread ok_function()
{
    callable x;
    return boost::thread(x);
} // x is destroyed, but it has been copied, so this is ok

boost::thread bad_function()
{
    callable x;
    return boost::thread(boost::ref(x));
}
// passing a reference to x, the reference is copied,
// but since x is destroyed, the behavior is undefined
```

Arguments

- Arguments are copied too, so unless they are references, it is ok to destroy them afterwards

```
class A {...};

void myfun(A a) { /* thread body */ }

boost::thread f()
{
    A a;
    return boost::thread(myfunction, a);
}
```

- if instead you need to pass a reference, use `boost::ref()`

Exceptions in threads

- If a function or callable object passed to `boost::thread` throws an exception that is not `boost::thread_interrupted`, the program is terminated

```
void f() {
    throws "This is an exception";
}

void g() {
    boost::thread th(f);
    cout << "This is never printed" << endl;
    th.join();
}
```

Destroying the object

- What happens if the object is destroyed before the thread has terminated?

```
x = 0;
void long_thread()
{
    for (long long i=0; i<100000000; i++);
    x = 1;
}

void make_thread()
{
    boost::thread th(long_thread);
}

int main()
{
    make_thread();
    while (!x); // waits for the thread to finish
}
```

Detaching and joining

- In the previous example, the thread becomes *detached*
- this means that the thread continues execution, but it is not possible to *join* it anymore
- it is possible to explicitly detach a thread by calling `detach` on the object.
  - in that case, the object becomes "*not a thread*"
- to wait for the thread termination, you can call the
  `void join()`
  member function
  - it will block the invoking thread waiting for the termination of the other thread
  - you can also invoke the
    `bool timed_join(const system_time &timeout)`
    specifying a time-out, after which it returns anyway
  - if you do a join on a "not a thread" object, the join returns immediately, but not error is reported

Interrupting a thread

- An “interrupt” to a thread corresponds to a “cancel” in the POSIX thread terminology
- to sent an “interruption”, you call
  `void interrupt();`
  member function.
- the thread continues to execute until one of the pre-defined *interruption points*
  - `join` and `timed_join`
  - `wait()` and `timed_wait()` on a condition variable
  - `sleep()`
  - a special `interruption_point()`
- when a thread is interrupted while blocked on one of the above functions, an exception `thread_interrupted` is thrown
  - if the exception is not interrupted, the **thread** will terminate

Disabling interrupts

- It is possible to “disable” interrupts like follows

```
void g()
{
    // interruption enabled here
    {
        boost::this_thread::disable_interruption di;
        // interruption disabled
        {
            boost::this_thread::restore_interruption ri(di);
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

- The `disable_interruption` object implements the RAII technique

Thread exit

- It is possible to install “exit handlers”
  - these are functions called upon thread exist, even when the thread is interrupted
- it is possible to install such handlers by calling
  `void boost::this_thread::at_thread_exit(Callable func);`
  where Callable is a function or a function object.
- the Callable is copied in a thread internal storage, and it is called also when the thread has been detached
- the function is not called if `exit()` is called and the process is terminated

Groups

- It is possible to create thread groups by using class `thread_group`

```
using namespace boost;
thread_group grp;
grp.create_thread(fun_body); // creates a thread in the group
grp.add_thread(new thread(fun_body)); // equivalent
thread *pt = new thread(fun_body);
grp.add_thread(pt);             // equivalent
...
grp.remove_thread(pt);          // remove it from the group
grp.join_all();                 // wait for all of them
```

- the only motivation for a group is to be able to join all of the threads in a group, or to interrupt all of them with
  `void join_all();`
  `void interrupt_all();`

Outline

Mutexes

- Boost.Thread provides different kinds of mutex:
  - non-recursive, or recursive
  - exclusive ownership or shared ownership (multiple readers / single writer)
- To implement all mutexes in a generic way, the Boost.Thread library supports four basic concepts of *lockable objects*:
  - Lockable, TimedLockable, SharedLockabel, UpgradeLockable

Concepts

- A *Lockable* class must provide three functions:

```
void lock();
void try_lock();
void unlock();
```

- A *TimedLockable* class must additionally provide:

```
bool timed_lock(boost::system_time const& abs_time);
template<typename DurationType>
bool timed_lock(DurationType const& rel_time)
```

Concepts - II

- The *SharedLockable* concept must additionally provide:

```
void lock_shared();
bool try_lock_shared();
bool timed_lock_shared(boost::system_time const& abs_time);
void unlock_shared();
```

- Finally, the *UpgradeLockable* concept must additionally provide:

```
void lock_upgrade();
void unlock_upgrade();
void unlock_upgrade_and_lock();
void unlock_upgrade_and_lock_shared();
void unlock_and_lock_upgrade();
```

- this allows to upgrade the ownership from shared to exclusive, and vice-versa

Classes implementing concepts

- The simplest mutex implements Lockable:

```
class mutex:
    boost::noncopyable
{
public:
    mutex();
    ~mutex();

    void lock();
    bool try_lock();
    void unlock();

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

TimedLockable

- The following one implements a timed mutex:

```
class timed_mutex:
    boost::noncopyable
{
public:
    timed_mutex();
    ~timed_mutex();

    void lock();
    void unlock();
    bool try_lock();
    bool timed_lock(system_time const & abs_time);

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<timed_mutex> scoped_timed_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_timed_lock scoped_lock;
};
```

Recursion

- A recursive mutex is used to avoid self-locking when we implement complex classes

```
class recursive_mutex:
    boost::noncopyable
{
public:
    recursive_mutex();
    ~recursive_mutex();

    void lock();
    bool try_lock();
    void unlock();

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<recursive_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

- A similar one is `recursive_timed_mutex`

Shared mutex

- A shared mutex is used to implement multiple readers / single writer

```
class shared_mutex
{
public:
    shared_mutex();
    ~shared_mutex();

    void lock_shared();
    bool try_lock_shared();
    bool timed_lock_shared(system_time const& timeout);
    void unlock_shared();

    void lock();
    bool try_lock();
    bool timed_lock(system_time const& timeout);
    void unlock();

    void lock_upgrade();
    void unlock_upgrade();

    void unlock_upgrade_and_lock();
    void unlock_and_lock_upgrade();
    void unlock_and_lock_shared();
    void unlock_upgrade_and_lock_shared();
};
```

Guards

- Instead of accessing mutexes directly, it is useful to use guards
- the simplest one is `lock_guard`

```
template<typename Lockable>
class lock_guard
{
public:
    explicit lock_guard(Lockable& m_);
    lock_guard(Lockable& m_,boost::adopt_lock_t);

    ~lock_guard();
};
```

Unique lock

- A more complex implementation allows to release and acquire the ownership directly:

```
template<typename Lockable>
class unique_lock
{
public:
    unique_lock();
    explicit unique_lock(Lockable& m_);
    unique_lock(Lockable& m_,adopt_lock_t);
    unique_lock(Lockable& m_,defer_lock_t);
    unique_lock(Lockable& m_,try_to_lock_t);
    unique_lock(Lockable& m_,system_time const& target_time);

    ~unique_lock();

    ...

    void lock();
    bool try_lock();

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const& relative_time);
    bool timed_lock(::boost::system_time const& absolute_time);

    void unlock();
    bool owns_lock() const;
    Lockable* mutex() const;
    Lockable* release();
};
```

Shared lock

- The guard for readers (writers will use a `unique_lock`)

```
template<typename Lockable>
class shared_lock
{
public:
    shared_lock();
    explicit shared_lock(Lockable& m_);
    shared_lock(Lockable& m_,adopt_lock_t);
    shared_lock(Lockable& m_,defer_lock_t);
    shared_lock(Lockable& m_,try_to_lock_t);
    shared_lock(Lockable& m_,system_time const& target_time);

    ~shared_lock();

    ...

    void lock();
    bool try_lock();
    bool timed_lock(boost::system_time const& target_time);
    void unlock();

    ...
};
```

Non-member functions lock

- If we want to lock multiple mutexes at once, we can use one of the following global (non-member) functions

```
template<typename Lockable1,typename Lockable2>
void lock(Lockable1& l1,Lockable2& l2);

template<typename Lockable1,typename Lockable2,typename Lockable3>
void lock(Lockable1& l1,Lockable2& l2,Lockable3& l3);

template<typename Lockable1,typename Lockable2,typename Lockable3,typename Lockable4>
void lock(Lockable1& l1,Lockable2& l2,Lockable3& l3,Lockable4& l4);

...
```

- The order in which they are acquired is unspecified, but it guarantees deadlock avoidance; in other words, even swapping the order in two different tasks, avoids deadlock
- (Question: you know how?)
- other similar functions

```
template<typename ForwardIterator>
void lock(ForwardIterator begin,ForwardIterator end);
```

Outline

Conditions

- Two implementations: `condition_variable` and `condition_variable_any`
- the first one requires an instance of `unique_lock`, and it uses this information to perform some internal optimization
- the second one can work with every mutex, therefore has a more complex and less optimized implementation

```
class condition_variable
{
public:
    condition_variable();
    ~condition_variable();

    void notify_one();
    void notify_all();

    void wait(boost::unique_lock<boost::mutex>& lock);

    template<typename predicate_type>
    void wait(boost::unique_lock<boost::mutex>& lock,predicate_type predicate);

    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    boost::system_time const& abs_time);
    ...
};
```

Barrier

- A Barrier is a very simple object:
  - it contains a counter that it is initialized to some value *n*
  - when a thread calls is wait function, it blocks
  - the *n*-th thread will unblock all previous ones

```
class barrier
{
public:
    barrier(unsigned int count);
    ~barrier();

    bool wait();
};
```

Outline

Asynchronous values

- Sometimes we want to launch a thread to compute a value, and then perform some other action
- at some point, we need to get the result of the computation
- this can be done by using an appropriate protocol, mutex and conditions
- however, this is a very common operation, so it has been packaged in the library
- the basic concept is the *future*
  - a future is an object that will contain the result when it is ready
  - if we want to get the result, but it has not been produced yet by the thread, we block, and we will be unblocked when the result is finally produced
  - this is similar to a one place-producer/consumer buffer that is used only once

Futures in Boost.Thread

- The simplest way to produce the behavior described in the previous slide is to use the concept of “packaged task”

```
int calculate_the_answer()
{
    return 42;
}

boost::packaged_task<int> pt(calculate_the_answer);
boost::unique_future<int> fi=pt.get_future();

boost::thread task(boost::move(pt)); // launch task on a thread

fi.wait(); // wait for it to finish

assert(fi.is_ready());
assert(fi.has_value());
assert(!fi.has_exception());
assert(fi.get_state()==boost::future_state::ready);
assert(fi.get()==42);
```

Promises

- Promises are more low-level, in the sense that we can avoid using the packaged task structure

```
boost::promise<int> pi;
boost::unique_future<int> fi;
fi=pi.get_future();

pi.set_value(42);

assert(fi.is_ready());
assert(fi.has_value());
assert(!fi.has_exception());
assert(fi.get_state()==boost::future_state::ready);
assert(fi.get()==42);
```

Lazy futures

- It is also possible to compute the result only when strictly necessary, by using wait callbacks
- a wait callback is a callable invoked when a thread performs a wait on the future

```
int fun()
{
    return 42;
}

void invoke_lazy_task(boost::packaged_task<int>& task)
{
    try {
        task();
    }
    catch(boost::task_already_started&) {}
}

int main()
{
    boost::packaged_task<int> task(fun);
    task.set_wait_callback(invoke_lazy_task);
    boost::unique_future<int> f(task.get_future());

    assert(f.get()==42);
}
```

Shared future

- `unique_future` implements exclusive ownership: copying a `unique_future` passes the ownership
- if you need shared ownership, use `shared_future` instead

```
template <typename R>
class shared_future {
public:
    typedef future_state::state state;

    shared_future();
    ~shared_future();

    // copy support
    shared_future(shared_future const& other);
    shared_future& operator=(shared_future const& other);

    // move support
    shared_future(shared_future && other);
    shared_future(unique_future<R> && other);
    ...
    // retrieving the value
    R get();
    ...
    // waiting for the result to be ready
    void wait() const;
    template<typename Duration>
    bool timed_wait(Duration const& rel_time) const;
    bool timed_wait_until(boost::system_time const& abs_time) const;
};
```

Multiple wait

- It is possible to wait for more than one future by using the following global (non-member) functions:

```
template<typename Iterator>
Iterator wait_for_any(Iterator begin,Iterator end);

template<typename F1,typename F2>
unsigned wait_for_any(F1& f1,F2& f2);

template<typename F1,typename F2,typename F3>
unsigned wait_for_any(F1& f1,F2& f2,F3& f3);

...
```

- Waits for any of the futures specified in the list

```
template<typename Iterator>
void wait_for_all(Iterator begin,Iterator end);

template<typename F1,typename F2>
void wait_for_all(F1& f1,F2& f2);

template<typename F1,typename F2,typename F3>
void wait_for_all(F1& f1,F2& f2,F3& f3);
```

- Waits for all the futures specified in the list