Multithreading with C++ and the BOOST library

J. Pela

Imperial College London

2014-07-22

Disclaimer

About me...

- I am not an expert in multi-threading...
- I am a user with some experience and understanding of this subject and I will share that with you.

About this presentation...

- This is not a complete guide on the subject of multiple threads... it offers one possible approach.
- Many things I will show are reflection my understanding of this subject and should not be understood as such :)

About the usage

- Multithreading is very useful but may be restricted in some environments, like the IC batch system, so always think before you code and don't annoy system admins. ;)

What is multithreading?

Multithreading is a widespread programming and execution model that:

Definition:

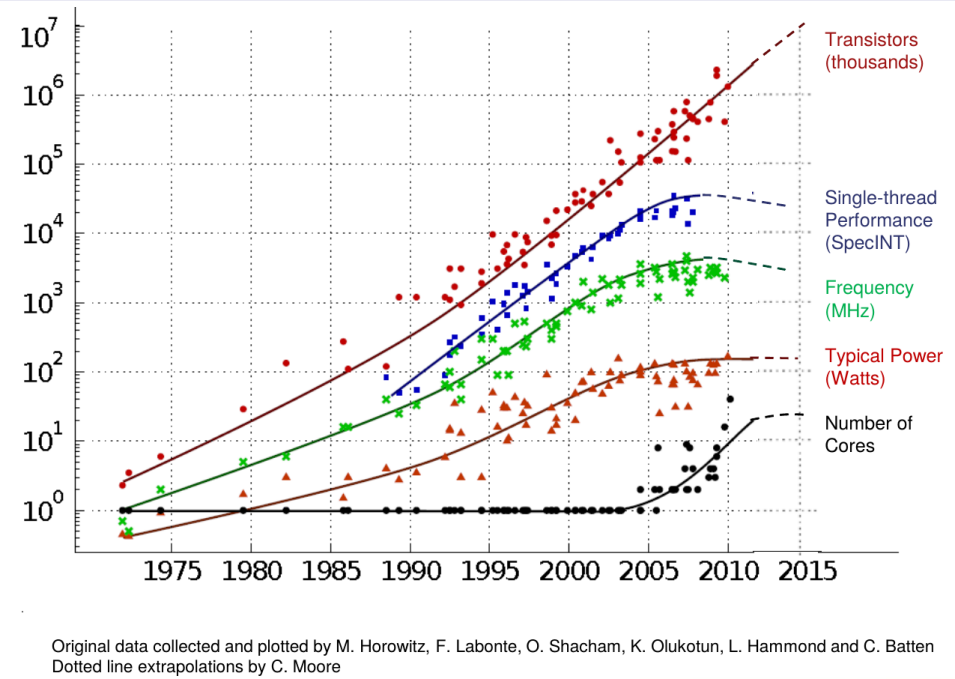
- Allows multiple threads to exist within the context of a single process.
- Threads share the process's resources, but are able to execute independently.
- Provides developers with a useful abstraction of concurrent execution.
- Can also be applied to a single process to enable parallel execution on a multiprocessing system.

Imperial College London

Evoluting of computing

In the last year we have seen a tendency for single core performance to stabilise and the number of cores to increase in the CPU market.

Evolution of CPU performance



Reasons

- Reaching the minimum size limit for a single transistor
- Difficulty in increasing clock speed
- Increasing complexity of each CPU core: optimisation, additional instructions, etc.

Solution

Make CPUs with lots cores working in parallel. But this implies code that can run in multiple execution threads!

Obviously there are several advantages and disadvantages in this approach! Lets look at some:

Multithreading: Advantages and disadvantages.

Advantages:

- Theoretically optimal multithreaded code can be the number of available cores times faster that any single core code.
- Allows better resource usage since several processors will use the same resources thus less resources will be idle.
- Can split low speed operations (example: file access from disk) from high speed calculations thus removing idle unnecessary idle times.
- In low power applications allow to switch off completely some of the cores allowing to only have minimal necessary resources powered.
- GPUs take this approach even further and some boards out there have +1000 cores.

Disadvantages.

- Dealing with multiple threads opens several new classes of problems in coding which can be very hard to debug.
- Sharing resources may also create overheads at hardware and software level.
- Greater exposure of the hardware to OS and user programs thus increasing complexity of code.

This are just some of the issues to consider.

Imperial College London

Why BOOST?

BOOST Library

BOOST is an extensive C++ library that:

- Widely used available over multiple platforms in a portable way
- Covers multiple topics: date/time manipulation, filesystem interfaces, networking, numerical programming, interprocess communication, etc.
- Good documentation with lots of examples.
- Well maintained with frequent updates and bug fixes.

BOOST Multithreading

- Provides complete support to thread handling.
- Package classes are portable thus allowing OS independent code.
- Many examples and good community support.

The class boost::thread

- Is the representation on the boost library of a single thread of execution.
- This interface is implemented in the platforms where boost is available allowing making portable multithreaded code.
- Normally boost::thread object is constructed by passing the function or method it is to be run.

NOTE: A thread object can be set to a special state of not-a-thread, in which case it is inactive

Compiling examples on this tutorial

For your system you need to supply boost headers and libraries location. For my system (opensuse linux) would be:

```
g++ <input_file>.cxx -o <output_file>.exe -I/usr/include/boost -L/usr/lib64 -lboost_thread-mt -lboost_system
```

Simulating work

To simulate real work we are going to simple use boost "wait" function :D

```
boost::posix_time::seconds workTime(3);
boost::this_thread::sleep(workTime);
```

Note that boost::this_thread gives us a convenient refer to the currently running thread.

Imperial College London

Type A: A Thread Function

Example A

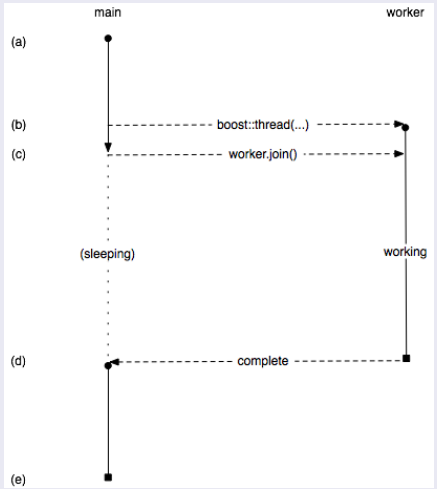
```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>

using namespace std;

void workerFunc(){

    boost::posix_time::seconds workTime(3);
    cout << "Worker: running" << endl;

    // Pretend to do something useful...
    boost::this_thread::sleep(workTime);
    cout << "Worker: finished" << endl;
}

int main(int argc, char* argv[]){

    cout << "main: startup" << endl;
    boost::thread workerThread(workerFunc);

    cout << "main: waiting for thread" << endl;
    workerThread.join();
    cout << "main: done" << endl;

    return 0;
}
```

What is happening?

Here we just pass a basic C-style function to boost::thread constructor.

- The function will start running immediately.
- We can wait (in the main thread) for the threaded function to complete by calling join().

Diagram:



Type A: Passing Arguments

This last example not not very useful, we probably would like to pass some parameters to out function for processing. Let;s do that:

- simply add parameters to the thread object's constructor
- arguments will automatically be and passed in to the thread function

Contructor

```
void workerFunc(const char* msg, unsigned delaySecs) //...
```

Now pass the arguments to the thread constructor after the name of the thread function:

Creating the thread

```
boost::thread workerThread(workerFunc, "Hello, boost!", 3);
```

Exercise:

It's your turn now! Let's try to implement, compile and run this example! (5 min)

Imperial College London

Type B: Functor

Functor

```
class Worker{
public:
  Worker(unsigned delaySecs) : m_delay(delaySecs){}

  void operator()(){
    boost::posix_time::seconds workTime(m_delay);
    std::cout << "Worker: running" << std::endl;

    boost::this_thread::sleep(workTime);
    std::cout << "Worker: finished" << std::endl;
  }

private:
  unsigned m_delay;
};
```

Functor

Functor is a name of an object that can be called like a function.

- The class defines a special method by overloading the operator()
- This operator will be invoked when the functor is called
- This allows the functor to encapsulate the thread's context and still behave like a thread function.

Main program:

```
int main(int argc, char* argv[]){

    std::cout << "main: startup" << std::endl;
    Worker w(3);
    boost::thread workerThread(w);

    std::cout << "main: waiting for thread" << std::endl;
    workerThread.join();
    std::cout << "main: done" << std::endl;

    return 0;
}
```

Main code

- We create our callable object with all necessary parameters.
- Pass the instance to the boost::thread constructor, which will invoke the operator()()
- This becomes a new thread and as access to all objects context and methods.

Type C: Method in a thread

Sometimes its convenient to define an object with an instance method that runs on its own thread. This can be done easily with BOOST.

What to do?

- We will be making a regular function into a thread.
- We have to pass the method address to boost thread via the class qualifier.
- On C++ all methods receive a implicit "this" as first parameter, we need to replicate that behavior.
- then we will pass the remaining parameters.

Code:

```
Worker w(3);
boost::thread workerThread(&Worker::processQueue, &w, 2);
```

NOTE: be careful to not destroy the object while the thread is running since this will create a, difficult to debug, mess in memory.

Type D: Object encapsulated thread

You may want to make classes that manage their own threads. Thus whole process of creation, handling and destruction of new threads is encapsulated.

Class code

```
//...
void start(int N){
  m_Thread = boost::thread(&Worker::doRun,this,N);
}

void join(){
  m_Thread.join();
}

private:
  boost::thread m_Thread;
//...
```

Main code

```
Worker worker;
worker.start(3);
worker.join();
```

Structure

- The thread object will exist as an instance member (as opposed to a pointer).
- In this case the default constructor for the thread creates it in an invalid state called "not-a-thread".
- Thread instance will wait until we assing a real one to start.
- Method Worker::start() spawns a thread that will run method doRun.
- Method Worker::join() will call the tread join().

So on the main code we do not need to have any thread action only direct interaction with the object.

Exercise:

It's your turn now! Let's try to implement, compile and run this example! (5 min)

Imperial College London

Topic conclusions

We have seen several techniques to create and handle threads with BOOST

Types of implementation:

- Type A: C-Style function
- Type B: Functor
- Type C: Method in a thread
- Type D: Object encapsulated thread

BOOST allows great flexibility and ease in implementing multithread code. But there are many possible problems to consider. Let's look at some:

Problems: shared state

Up to now all examples that we looked had only 2 threads and had no shared state. What is meant by shared state is any shared data or resources between threads:

Shared state

- such as file handle
- socket
- graphics context
- queue
- buffer
- ...

If two threads are truly independent

- They can safely run concurrently without care or consideration.
- We wouldn't need sophisticated mechanisms for synchronisation.

- As soon as you introduce shared state, you have to worry about atomicity, consistency, race conditions, and all sorts of issues.
- So one of the first design considerations for concurrent systems is to try to minimise the amount of shared state between threads.

Problems: atomicity

An operation is atomic if the operation completes without interruption. It is never partially complete, which may leave the system or data in an inconsistent or invalid state.

Classic example: Bank transfer (pay landlord)

Operations:

- Ensure you have sufficient funds
- Ensure the receiving account number is valid
- Withdraw the £1200 in funds from your account
- Deposit the £1200 in the landlord's account

What can go wrong?

- Steps 1 and 2 are precondition, if operation fails there no problem.
- But after step 3 you "absolutely" want step 4 to complete!
- Steps 3 and 4 must be performed atomically

The same concept applies to in-memory operations that must be atomic. A for this we can use mutex (for "mutual exclusion"):

- Used to serialise access to resources within a region of code that must run as an atomic operation.
- We would lock the mutex before the transaction, and unlock it afterwards.

Problems: race conditions

A race condition is a general term for a class of problems, whereby the result of an operation is at the mercy of the timing of concurrent events:

- Program becomes non-deterministic, it may not always produce the correct result!
- One of the most subtle and difficult to debug problems in software development.

Example: Counter - 1 thread incremented

One of the most basic operations in C++ is to increment (example: $m_SequenceNumber++$;) a counter and most would assume this is an atomic action, but in fact (in x86 architecture) would be:

```
movl    -12(%ebp), %eax
incl    %eax
movl    %eax, -12(%ebp)
```

So even a simple operation can consist of several operation in memory (3 operations in this case load: increment, then store).

There are two potential points that a race condition could occur: between the load and the increment, and between the increment and the save.

Example: Counter - 2 thread incremented

Thread	Instruction	m_SequenceNumber	Register
A	LOAD m_SequenceNumber,R0	234	234
A	INCR R0	234	235
B	LOAD m_SequenceNumber,R1	234	234
B	INCR R1	234	235
B	STORE R1,m_SequenceNumber	235	235
A	STORE R0,m_SequenceNumber	235	235

So, instead of the sequence number being incremented twice, and being 236 as we would expect, the sequence number is only 235.

This problem can be avoided by using mutex and locking usage of our variable by other threads. But Mutexs can generate problems too...

Imperial College London

Problems: deadlocks

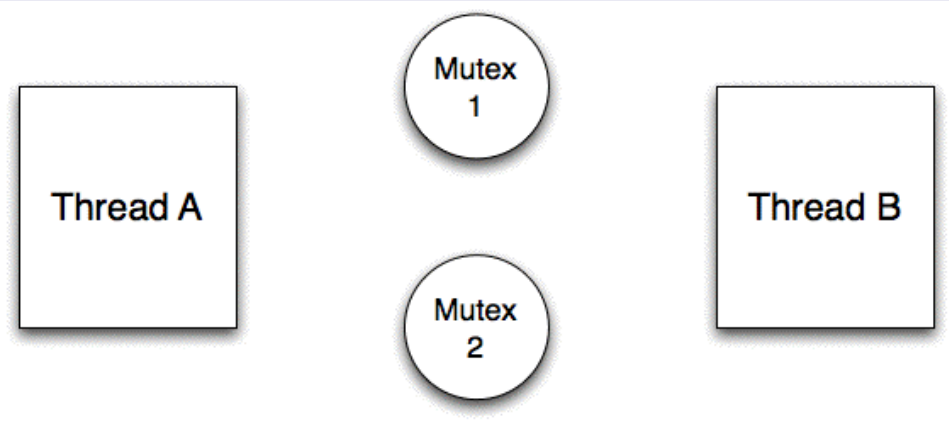
When more than one thread locks more than one mutex, there arises the potential for a condition known as a deadlock.

- One thread is holding a lock while waiting for another to become available
- Second thread is holding the second lock and waiting for the first lock to become available
- More complicated situations may arrives where several threads form a ring of holding and waiting for locks.

Problematic code:

```
void threadA(){
    while (running){
        mutexOne.lock();
        mutexTwo.lock();
        processStuff();
        mutexOne.unlock();
        mutexTwo.unlock();
    }
}
void threadB(){
    while (running){
        mutexTwo.lock();
        mutexOne.lock();
        processStuff();
        mutexTwo.unlock();
        mutexOne.unlock();
    }
}
```

Diagram:



The problem of deadlocks can (largely) be avoided by consistently locking mutexes in the same order, and unlocking them in reverse order.

Problems: exceptions

Exceptions in $C++$ can be a very effective mechanism for error handling. However, like many $C++$ features, care must be taken, especially in threads.

- A thread is destroyed when the thread function returns.
- Uncaught exceptions can cause the entire thread to exit, without warning or notice.
- So if you have a thread that mysteriously disappears, an uncaught exception is a likely culprit.

If you wish to avoid your thread being killed, you should add a top-level try-catch block.

- You may simply log the uncaught exception then exit.
- You may resume thread processing if it is safe to do so.

Code:

```
void processThread(){

  while(keepProcessing){
    try{// Do actual work}

     // Catch more specific exceptions first if you can
     catch (std::exception& exc){
       log("Uncaught exception: " + exc.what());
       // Maybe return here?
     }
  }
}
```

Just as an errant exception can cause havoc with threads running, they can also cause problems with mutexes. We can use lock guard to ensure mutexes are unlocked, even if an exception is thrown.

Problems: other problems

There are more categories of problems you may encounter but they fall out of the scope of today's presentation. But just to reinforce the idea that multithreaded programming is not a trivial undertaking, I leaving you a list of some other problem categories for further studies:

- starvation
- priority inversion
- high scheduling latency
- ...

Mutithreading is currently an active research and development area and issues such as the ones presented today should be always considered for concurrent programming.

Topic conclusions

Most typical problems with multi-threading are well known and there are clear solutions for them and is possible to write robust code in this paradigm but one must be careful to avoid the many pitfalls.

Somethings to take into account in your coding:

- Minimise shared state
- Program defensively
- Assume your threading code can be preempted at any time
- Identify shared resources and protect them with mutexes
- Identify algorithms that need to be atomic
- Identify code with dependent results that needs to be atomic
- Minimise the number of resources shared between threads
- Minimise the duration of locks
- Ensure exceptions cannot disrupt synchronisation flow
- Always acquire and release mutexes in the same order

Now lets look at the details of how to use mutexs.

Mutual exclusion: Mutex

What is a mutex?

- A mutex is special kind of lock, which is used to protect shared resources.
- It guaranteed to be held by at most one thread at a time.
- Can be used as a protection mechanism for resources and data
- Can be used to avoid race conditions, and implement atomic operations.

What happens if multiple threads try to lock a mutex?

- If a mutex is not locked by a thread, then any thread can potentially lock it.
- If any other thread attempts to lock it, it will typically block and wait until the mutex is unlocked by the owning thread.
- It is also possible to return if the lock fails, or wait for a certain period of time before giving up.

A mutex is typically declared alongside the resource it protects. The mutex should be in the same scope as the resource, or an enclosing scope. BOOST provides 4 flavours of mutex.

Type A: Regular Mutex

The simplest form of mutex is a regular boost::mutex. You lock and unlock it, and only one thread can lock the mutex at a time.

- Any thread that calls lock() on a mutex held by another thread will block indefinitely (important for factor in design).
- This must be carefully managed to avoid program hangs.

Code:

```
boost::mutex work_queue_mutex;
queue<item> work_queue;

// ...

work_queue_mutex.lock();

auto work_item = work_queue.pop();

work_queue_mutex.unlock();
```

Regular mutexes also have a try_lock() method, which will return immediately with a failure status if the mutex cannot locked.

- Most threads run in a loop, so this can be useful to preform something else
- In the case nothing else can be done maybe putting the thread to sleep for some time may be useful

Type B: Timed Mutexes

The boost::timed_mutex class is a subtype of boost::mutex, which adds the ability to specify a timeout.

- Attempts to lock the mutex for some time and if it cannot returns false.
- Takes either an absolute time, or a relative time.

Code:

```
boost::mutex work_queue_mutex;
queue<item> work_queue;
TimeDuration mutex_timeout;

// ...

if (work_queue_mutex.timed_lock(mutex_timeout)){

    auto work_item = work_queue.pop();
    work_queue_mutex.unlock();
}
```

With some study of your application you can tune the timeout to maximise the performance of your algorithm.

Type C: Recursive Mutexes

Normally a mutex is locked only once, then unlocked. But on some applications may be useful to allow the same thread to lock the same mutex multiple times.

- Useful for nested method calls.
- Situations where the thread may call multiple methods that require a given mutex lock.
- Still the mutex must be unlocked the same number of times that it got locked on the specific thread.
- This behaviour assumes that allow method the lock/unlock mutex will do so in a compartmentalised and nested way.

Type D: Shared Mutexes

Some concurrency scenarios involve having one writer and many readers.

Example

- One thread downloading data from the network, another displaying the data on the screen and another saving the data to database.
- Downloading thread needs locking for writing
- Other two threads for reading
- There is no reason for the reading threads to exclude each other (concurrent reading is safe)
- Only when the downloading thread is writing its necessary to lock the other out.

Code:

```
boost::mutex work_queue_mutex;
queue<item> work_queue;

// Reader thread
work_queue_mutex.read_lock();
auto work_item = work_queue.head();
work_queue_mutex.unlock();

// Writer thread
work_queue_mutex.write_lock();
auto work_item = work_queue.push(new_item);
work_queue_mutex.unlock();
```

Summary

- Any time the reading threads need to read the resource, they obtain a read-lock.
- Allows other read-locks to successfully access the resource, preventing a write-lock.
- When the writer obtains lock, readers will need to wait until update is complete.

Lock/Unlock Pairing:

If a mutex is not released due to a logical error (such as an uncaught exception), this is probably not recoverable. Thus it is vitally important that all lock/unlock operations appear in pairs.
To protect against this class of problem, the lock_guard object was introduced.

- Locks the mutex for you in its constructor, and unlocks it in the destructor.
- If thread is destroyed for some reason the mutex is automatically released.

Code:

```
unsigned applyTotals(unsigned count){

  boost::lock_guard totalsLock(mTotalsMutex);
  mTotals.globalCount += count;
  writeTotalsToDatabase();
}
```

The totalsLock will acquire (or wait to acquire) the mutex when created, and when it is destroyed at the end of the method, it will unlock the mutex. If an exception is thrown, the lock's destructor will still be called, and the mutex will be safely released. This ensures the operation is atomic, and can safely handle error conditions.

Conditional variables

While a mutex allows you to lock a resource for usage it tells you nothing about the content of the resource itself (example: stack of events to process). Cconditional variables can help:

Code:

```
unsigned applyTotals(unsigned count){
boost::condition_variable cond;
boost::mutex mut;
bool data_ready;

void process_data();

void wait_for_data_to_process(){

  boost::unique_lock<boost::mutex> lock(mut);
  while(!data_ready){
    cond.wait(lock);
  }
  process_data();
}
```

- Lock is passed to wait: wait will atomically add the thread to the set of threads waiting on the condition variable, and unlock the mutex.
- When the thread is woken, the mutex will be locked again before the call to wait returns.
- This allows other threads to acquire the mutex in order to update the shared data.
- Wnsures that the data associated with the condition is correctly synchronised.

Summary and next steps

Today we covered

- Multithreading techniques using BOOST
- Common problems and solutions
- Mutex and synchronisation
- Conditional variables

Thank you:

- Thank you for listening and I hope today's presentation helps you with your code.
- A good part of this presentation was based on Mr. Gavin Baker blog on C++ coding where you can find more examples.