Concurrency in C++ 11

thread

- C++11 has native support for threads.
- You can create a thread and give it a task (along with the arguments for the task).
- Threads of a process share the same address space. This of course means that you have to be careful with global resources.
- Threads have their own stack space.

thread

- thread t1;
  - Create a thread object with no task associated with it. This will not spawn a thread.

- thread t1 (DoSomething);
  - Create a thread which will call function DoSomething

- thread t1 (DoSomething, arg1, arg2);
  - Create a thread which will call function DoSomething and pass it parameters arg1 and arg2

- thread t2(t1);
  - t2 is created from t1 using move constructor

thread

- Once you start thread(s) from your main thread, you can at some point wait for the threads to finish (synchronization barrier).

- This can be done by calling join on the thread(s).

```
// This is the main thread, and it will start a thread t1 below:

thread t1 ( DoSomething, arg1 );    // Spawn a thread that will call DoSomething


// Now main thread can do some other things here, if needed
// Some code here that executes in main thread.

// Now, main thread wants to wait for thread t1 to finish running, so it waits for
// that by calling join() on t1:

t1.join();
```

thread

- Calling detach on a thread object separates (detaches) that thread object from the executing thread.

```
thread t1 ( DoSomething, arg1 );
…
t1.detach();
```

- Once detach is called, the thread is no longer joinable.
  - This means you now cannot call join on t1.

- You should call detach only on a thread that is joinable.

thread

- If a thread object is not joinable, it means the following is true:
  - t1.get_id() == thread::id();
  - i.e., t1's thread id is same as the default id
  - i.e., its not associated with any thread of execution.
- And, you should not call join() on this thread object that is not joinable.

thread

- thread objects cannot be copied, but can be moved.
- If a thread is still joinable when a thread object destructor is called, it will terminate the program (by calling std::terminate).
- This means that you have to ensure that the thread is not joinable on any path through the program.
- Thread's destructor could have called detach or join, in order to not terminate.
- But they did not do this as calling detach or join as a general solution may not be appropriate for everyone.

Unjoinable

- A thread becomes unjoinable if any of the following happens:
  - join is called on it.
  - detach is called on it.
  - It is moved to another thread object.

- To make a thread unjoinable, you can have an RAII class to do the desired action (detach or join).

this_thread

- this_thread namespace has methods for the current thread.

- this_thread::get_id()
- this_thread::sleep_for (duration)
- this_thread::yield ()

thread_local

- If you declare a variable thread_local, then it is accessible by only its owning thread (unless you specifically share that address with other threads).
- Each thread has its own copy of thread_local objects/variables.
- Threads have their own stack space, so each thread has its own copy of thread_local objects, but these objects have a lifetime of the scope in which they are defined (like inside a method or function).
- The lifetime of a thread_local is as long as the thread is running.

thread_local

- thread_local objects are initialized before their first use.
- They are initialized to zero (if value not specified).

- Order of construction of thread_local objects is undefined.

```
class MyClass
{
  private:
          static thread_local  char *   mData;
};
```

thread

- int n = thread::hardware_concurrency();
- Returns the number of concurrent threads the underlying hardware can support.
- This is not meant to be an exact number representing the number of processors or cores, but more of a hint.

Mutex

- When you want shared data (among threads) to be updated by a thread, you can use mutex to grant exclusive access to it by a thread.
- A mutex can only be owned by one thread at any given point in time.
- If a thread makes a call to acquire a mutex, and that mutex is already owned by some other thread, then this can block the thread attempting to acquire, until the owner thread releases the mutex.
- Operations on a mutex:
  - lock()
  - unlock()
- You should minimize the parts of code that are protected by a mutex.
  - i.e., don't use mutex around code that does not really need to be locked (accessed exclusively by one thread at a time).

deadlock

- A deadlock is a situation where one or more threads (or processes) are waiting to lock a resource that is already locked.

```
mutex m;
void Foo()
{
    m.lock();              // Acquire the lock
    …
    Foo2 ();
    …
    m.unlock();
}
void Foo2()                // Foo2 also plans to acquire a lock on the same mutex
{
    m.lock();   ← deadlock, since this thread already locked m. So it hangs here
    ..
    m.unlock();
}
```

recursive mutex

- recursive_mutex is like a mutex, but a given thread can acquire it repeatedly.

- What we saw earlier was that if a thread acquires a mutex a second time (while still holding the same mutex), it will enter into a deadlock.

- recursive_mutex is a way to avoid the deadlock… in other words, lock on a recursive_mutex can be acuquired multiple times.

recursive mutex

```
recursive_mutex m;

void Foo()
{
    m.lock();              // Acquire the lock
    …
    Foo2 ();
    …
    m.unlock();
}
void Foo2()                // Foo2 also plans to acquire a lock on the same mutex
{
     m.lock();             ← lock succeeds since m is a recursive_mutex
     …
     m.unlock();
}
```

lock_guard

- The concept here is similar to smart pointers, i.e., you don't want the code to leave the mutex in a locked state because the code had an exception or an early return.

```
void Foo()
{
        lock_guard <mutex> lg1 ( m );
        …
}
OR                      // if you can reduce the amount of code to be locked, you should
void Foo()
{
        if ( someCondition )
        {
                lock_guard <mutex> lg1 ( m );
                …
        }
}
Destructor of lock_guard <mutex> will call unlock  on m
```

lock_guard

- If there is no condition to check, you can/should create a scope to lock only the needed part of the code:

```
void Foo()
{
        …
        …
        // Need to lock only the code below, adding a begin/end scope:
        {
                lock_guard <mutex> lg1 ( m );
                …
        }   ← Here lg1 will be destructed, which will release lock on m
        …
        …
}
```

multiple locks

- Lets say your code needs to acquire more than one lock.
- Look at the code below:

```
mutex m1, m2;                    // this code has potential problems.
m1.lock();
m2.lock();
```

Above code can lead to a deadlock if the thread locks m1, but then cannot lock m2 since another thread has m2 locked. And if that other thread is now trying to lock m1, both threads are deadlocked.

multiple locks

- One way to resolve this is to use the **lock** function to lock multiple mutex objects.

```
void Foo()
{
        mutex m1, m2;

        lock( m1, m2 );        // will lock m1 and m2, if it cannot, it will throw

        // now we can add a lock_guard for each mutex for safety.

        lock_guard< mutex > lg1( m1,  adopt_lock );
        lock_guard< mutex > lg2( m2, adopt_lock );

        // …
}
```

- **NOTE**: The adopt_lock parameter indicates that the thread already has a lock on the object (and so lock_guard should not attempt to lock it).

unique_lock

- This is similar to lock_guard, except it provides some extra features for lock attempts. Its use is a little more expensive than lock_guard.

- Multiple locks code can also be written using unique_lock

```
void Foo()
{
        mutex m1, m2;

        unique_lock  lockA ( m1, defer_lock );        // unique_lock for safety
        unique_lock  lockB ( m2, defer_lock );        // unique_lock for safety

        lock ( lockA, lockB );
        // …
}
```

async

- If you want to write a multi threaded program, you need :
  - A way to let your function run on a separate thread
  - A way to then get the results of that code you ran on a separate thread
    - The result could be what is returned by your function, or could be an exception thrown by your function.

- *async* lets you specify a function that you would want to potentially run in parallel (asynchronously from the thread that creates the async).
- *future* is what would let you get the results of that execution into the calling thread.

async

- async is easier to use than threads.
  - Threads are a lower level construct, and you are responsible to take care of a few things (more on this soon)
- async provides a way to access the result (via the future that it returns… we will see an example soon)
- Delegates thread management to the underlying system

- auto future1 = std::async(DoSomething);

- Now, when I need to look at the result:

```
try
{
        auto result = future1.get();   // Should expect an exception if DoSomething() can throw
}
catch ( MyException & e )          //  Have a try catch
{
        …
}
```

async

- async optionally takes a thread launch policy as a parameter.
  - auto future1 = std::async(DoSomething, std::launch::async);
  - This means you want DoSomething to run on a different thread.
- auto future2 = std::async(DoSomething, std::launch::deferred);
  - This means you want DoSomething to run only when get (or wait) is called. And when they are called, DoSomething will run on the same thread that calls get or wait, which means it will be a blocking call (i.e., the caller will block until DoSomething returns, either via completed execution or by throwing an exception).
  - So here you are explicitly saying that you want the execution of DoSomething to be deferred.
  - Note that if your call to get or wait is conditional, and that condition does not happen (evaluates to false), then DoSomething will not run.

async

- Default value of launch policy
- If I say:
  - auto future3 = std::async(DoSomething);
- That is same as saying:
  - auto future4 = std::async(DoSomething, std::launch::async | std::launch::deferred);
- This default policy then allows you to not worry about the decision of running DoSomething on a different thread or same thread.
- The underlying system (Standard library implementation) will then handle the decision, keeping oversubscription and load balancing in view (the system has knowledge of other threads/processes running, but you/your program only know about threads running in your program.
- So you can assume that the underlying system can make better overall decisions.

Default launch policy

- One thing to keep in mind is that if you use default launch policy and the code in question is using thread local variables, then you will not know which thread's TLS will get used.
  - This is because you don't know if the task will execute in the calling thread (deferred) or in a different thread (async)

Oversubscription

- Oversubscription is the situation where you have :
  - N threads that are ready to run
    - i.e., they have all the data needed to continue execution and are not blocked for anything such as I/O, etc.
  - M cores total in the hardware (computer)
    - Assume no hyper-threading, i.e., # hardware threads == number of cores
  - N > M
  - In other words, the number of threads ready to run is greater than number of cores. Now, that in itself is not an issue really, but it does mean that the threads will be time-sliced onto the cores, i.e., they will run for "some time" and then the core will context switch to another ready to run thread. This context switching does have a performance cost.

Load balancing

- Load balancing is simply balancing the execution of threads across the available cores on the processor(s).

Async vs. thread

- Reasons to use thread over async:
  - Thread pools
  - Thread priority
  - If u need access to some lower level thread APIs

- Reasons to use async over thread:
  - Let underlying system decide thread creation/load balancing/over subscription.
    - Note: This is if use default launch policy discussed earlier.
  - Easier to access return value information.
    - This is via the future returned by async.