Concurrency in C++11

In this lab you will learn the basics of running concurrent threads with shared memory. You will also get some general exposure to C++11, which is the latest incarnation of the C++ programming language.

- C++11
- Shared memory parallelization
    - Threads
    - Race conditions
    - Mutexes
    - Atomicity
    - Asynchronous tasks
    - Condition variables
- Producer-consumer problem

Resources

- Concurrency examples - Examples of concurrency in C++11 and other languages.
- C++ reference - Standard library reference that clearly marks what is new in C++11

C++11

C++11 is the latest C++ standard that was published in 2011. Before it was published, it was known as C++0x. There are still plenty of old references using this name and you should consider them synonymous. The standard is only a long document with specifications and it is up to various groups to implement compilers that conform to this standard.

Currently, GCC 4.8.1 supports almost all core features of C++11 (GCC C++11 status). C++11 is tightly coupled with its standard library, where many features are still missing (libstdc++ status). As of very recently, another popular open-source compiler, Clang, now has full C++11 support both in core functionality (Clang C++11 status) and its own version of the standard library (libc++ status).

The lab computers only have GCC 4.6, but we have made sure you can run this lab. However, if you read tutorials on C++11 and can't seem to get things to compile, it might be because your compiler lacks the necessary support. That being said, GCC doesn't even fully support the C++98 or the C++03 standards, so let's not worry too much about that.

One thing that C++11 and its standard library vastly improves is concurrent programming, so let's dive in.

Threads

Creating threads is easy:

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

void func(int x) {
    cout << "Inside thread " << x << endl;
}

int main() {
    thread th(&func, 100);
    th.join();
```

```
    cout << "Outside thread" << endl;
    return 0;
}
```

Copy this to `main.cpp` and compile it by running:

```
$ g++ -std=c++0x main.cpp -pthread
```

Note that we have to specify that we want to use C++11, but we're using its old name since we have an old version of GCC. We also have to specify `-pthread`, since the implementation for our GCC uses something called *pthreads* as its backend. Make sure it compiles and runs.

Race conditions

Let us imagine that `x * x` is a very costly operation (it's not, but use your imagination) and we want to calculate the sum of squares up to a certain number. It would make sense to parallelize the calulation of each square across threads. We can do something like this:

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

int accum = 0;

void square(int x) {
    accum += x * x;
}

int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }

    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

This should sum all squares up to and including 20. We iterate up to 20, and launch a new thread in each iteration that we give the assignment to. After this, we call `join()` on all our threads, which is a blocking operation that waits for the thread to finish, before continuing the execution. This is important to do before we print `accum`, since otherwise our threads might not be done yet. You should always join your threads before leaving `main`, if you haven't already.

Before moving on, also note that C++11 offers more terse iteration syntax of the `vector` class, very close in syntax to Java. We are also using the keyword `auto` instead of specifying the data type `thread`, which we can do whenever the compiler can unambiguously guess what the correct type should be. We added an `&` to retrieve a reference and not a copy of the object, since `join` changes the nature of the object.

Now, run this. Chances are it spits out 2870, which is the correct answer.

Let's use our bash shell to run it a few more times (assuming you compiled it to `a.out`):

```
$ for i in {1..40}; do ./a.out; done
```

See any inconsistencies yet? Better yet, let's list all distinct outputs from 1000 separate runs, including the count for each output:

```
$ for i in {1..1000}; do ./a.out; done | sort | uniq -c
```

You should definitely see plenty of incorrect answers, even though most of the time it gets it right. This is because of something called a *race condition*. When the compiler processes `accum += x * x;`, reading the current value of `accum` and setting the updated value is not an atomic (meaning indivisible) event. Let's re-write `square` to capture this:

```
int temp = accum;
temp += x * x;
accum = temp;
```

Now, let's say the first two threads are interleaved over time, giving us something like this:

```
// Thread 1             // Thread 2
int temp1 = accum;      int temp2 = accum;          // temp1 = temp2 = 0
                        temp2 += 2 * 2;             // temp2 = 4
temp1 += 1 * 1;                                     // temp1 = 1
                        accum = temp1;              // accum = 1
accum = temp2;                                      // accum = 4
```

We end up with `accum` as 4, instead of the correct 5.

Exercise:

- Before we fix the race condition, since keeping `accum` as a global variable is poor style, we would rather pass it into the thread. You already know how to pass variables, so add a parameter `int& accum` to `square`. It is important that it's a reference, since we want to be able to change the accumulator. However, we can't simply call `thread(&square, accum, i)`, since it will make a copy of `accum` and then call `square` with that copy. To fix this, we wrap `accum` in `ref()`, making it `thread(&square, ref(accum), i)`.

Mutex

A *mutex* (**mut**ual **ex**lusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the `main` function the same:

```
int accum = 0;
mutex accum_mutex;

void square(int x) {
    int temp = x * x;
    accum_mutex.lock();
    accum += temp;
    accum_mutex.unlock();
}
```

Try running the program repeatedly again and the problem should now be fixed. The first thread that calls `lock()` gets the lock. During this time, all other threads that call `lock()`, will simply halt, waiting at that line for the mutex to be unlocked. It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we're running our calculations.

Atomic

C++11 offers even nicer abstractions to solve this problem. For instance, the `atomic` container:

```
#include <atomic>

atomic<int> accum(0);

void square(int x) {
    accum += x * x;
}
```

We don't need to introduce `temp` here, since `x * x` will be evaluated before handed off to `accum`, so it will be outside the atomic event.

Tasks

An even higher level of abstraction avoids the concept of threads altogether and talks in terms of *tasks* instead. Consider the following example:

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

int square(int x) {
    return x * x;
}

int main() {
    auto a = async(&square, 10);
    int v = a.get();

    cout << "The thread returned " << v << endl;
    return 0;
}
```

The `async` construct uses an object pair called a *promise* and a *future*. The former has made a promise to eventually provide a value. The future is linked to the promise and can at any time try to retrieve the value by `get()`. If the promise hasn't been fulfilled yet, it will simply wait until the value is ready. The `async` hides most of this for us, except that it returns in this case a `future<int>` object. Again, since the compiler knows what this call to `async` returns, we can use `auto` to declare the future.

Exercise:

- Use `async` to solve the sum of squares problem. Iterate up to 20 and add your `future<int>` objects to a `vector<future<int>>`. Then, finally iterate all your futures and retrieve the value and add it to your accumulator. This should be only a few modifications from the code above.

`this_thread`

Let's make sure this really runs in parallel. Using the code from the last exercise, now add a `cout` that prints `x` inside `square`. Run your program again. Every time you run it, it should be listing the values of `x` in order. This seems awfully deterministic and is not characteristic of running things in parallel. We did start them in that order, so maybe the threads aren't overtaking each other. We can check this by adding a sleep inside `square`, which we can pretend is the heavy computation of `x * x`:

```
this_thread::sleep_for(chrono::milliseconds(100));
```

Note that all these seemingly global objects are in the `std` namespace, but since we issued `using namespace std`, we made them visible globally. Okay, run this and time the execution. They are clearly taking turns and they are not running in parallel. Use `cout` to print `this_thread::get_id()`. Since the main execution is also considered a thread, try printing the thread ID inside `main` using this same function. What does this tell you?

The function `async` by default gives the program the option of running it *asynchronously* or *deferred*. The latter means `square` will be called first when we call `get()`, and it will be executed in the same thread. Ideally, the program should make an intelligent decision, optimized for performance, but for some reason GCC always defers, so let's not give it a choice about it. Change the call to `async` as follows:

```
async(launch::async, &square, ...)
```

Run it again, timing the execution.

> **Note**
>
> We got a 20 time speed up only because our threads aren't actually heavy on the CPU while sleeping, so it's not an ideal surrogate for imagining that `x * x` actually takes 100 ms of CPU time. In the real case, the speed up would be largely determined by how many cores we have on our computer.
>
> Ideally, you should avoid starting more computationally intensive threads than your computer can truly run in parallel, since otherwise your CPU cores will start switching its attention between different threads. Each switch is called a *context switch* and comes with an overhead that will hurt performance.

Condition variables

If we return to threads, it would be useful to be able to have one thread wait for another thread to finish processing something, essentially sending a signal between the threads. This can be done with mutexes, but it would be awkward. It can also be done using a global boolean variable called `notified` that is set to `true` when we want to send the signal. The other thread would then run a for loop that checks if `notified` is `true` and stops looping when that happens. Since setting `notified` to `true` is atomic and in this example we're only setting it once, we don't even need a mutex. However, on the receiving thread we are running a for loop at full speed, wasting a lot of CPU time. We could add a short `sleep_for` inside the for loop, making the CPU idle most of the time.

A more principled way however is to add a call to `wait` for a *condition variable* inside the for loop. The instructor will cover this in Thursday's class, so this will be sneak preview:

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
#include <chrono>
#include <queue>
using namespace std;

condition_variable cond_var;
mutex m;

int main() {
    int value = 100;
    bool notified = false;
```

```
    thread reporter([&]() {
        /*
        unique_lock<mutex> lock(m);
        while (!notified) {
            cond_var.wait(lock);
        }
        */
        cout << "The value is " << value << endl;
    });

    thread assigner([&]() {
        value = 20;
        /*
        notified = true;
        cond_var.notify_one();
        */
    });

    reporter.join();
    assigner.join();
    return 0;
}
```

First of all, we're using some new syntax from C++11, that enables us to define the thread functions in-place as anynomous functions. They are implicitly passed the local scope, so they can read and write `value` and `notified`. If you compile it as it is, it will output 100 most of the time. However, we want the reporter thread to wait for the assigner thread to give it the value 20, before outputting it. You can do this by uncommenting the two `/* ... */` blocks in either thread function. In the assigner thread, it will set `notified` to `true` and send a signal through the condition variable `cond_var`. In the reporter thread, we're looping as long as `notified` is `false`, and in each iteration we wait for a signal. Try running it, it should work.

But wait, if `cond_var` can send a signal that will make the call `cond_var.wait(lock)` blocking until it receives it, why are we still using `notified` and a for loop? Well, that's because the condition variable can be spuriously awaken even if we didn't call `notify_one`, and in those cases we need to fall back to checking `notified`. This for loop will iterate that many times.

This is a simplified description since we are also giving `wait` the object `lock`, which is associated with a mutex `m`. What happens is that when `wait` is called, it not only waits for a notification, but also for the mutex `m` to be unlocked. When this happens, it will acquire the lock itself. If `cond_var` has acquired a lock and `wait` is called again, it will be unlocked as long as it's waiting to acquire it again. This gives us some structure of mutual exclusion between the two threads, as we will see in the following example.

Producer-consumer problem

Time permitting

You should now have all the tools needed to fix an instance of the producer-consumer problem. Simply put, one thread is producing goods and another thread is consuming goods. We want the consumer thread to wait using a condition variable, and we want `goods.push(i)` to be mutually exclusive to `goods.pop()`, so are data doesn't get corrupted. We are letting `c++` and `c--` be surrogates for this mutual exclusion, since we can easily check if we correctly end up with 0 in the end. Run the code as it is, and you will see that the net value is way off:

```
#include <iostream>
#include <thread>
#include <condition_variable>
```

```
#include <mutex>
#include <chrono>
#include <queue>
using namespace std;

int main() {
    int c = 0;
    bool done = false;
    queue<int> goods;

    thread producer([&]() {
        for (int i = 0; i < 500; ++i) {
            goods.push(i);
            c++;
        }

        done = true;
    });

    thread consumer([&]() {
        while (!done) {
            while (!goods.empty()) {
                goods.pop();
                c--;
            }
        }
    });

    producer.join();
    consumer.join();
    cout << "Net: " << c << endl;
}
```

Don't expect the fix to be that trivial, and feel free to ask the instructor for help. Just wrapping all access to the shared memory in lock-unlock blocks can fix this problem, but remember that we don't want the consumer loop to run amok, taking up resources, so a condition variable is ideal. Use the commands for executing the problem 1000 times to test the integrity of your solution.