Homework 1

COP 5618/CIS4930 Concurrent Programming
Due: 10/5/16 at 11:59pm

Thread local variables are provided in Java via the java.lang.ThreadLocal<T> class and in C++11 via the thread_local storage class specifier. They provide a convenient way for each thread to have its own independently initialized and maintained instance of an object.

See https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html and http://en.cppreference.com/w/cpp/language/storage_duration for documentation.

As an example in c++, consider the following

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

thread_local  int counter = 1;  //each thread gets and instance initialized to 1
std::mutex cout_mutex;  //used to serialize output

void increment_counter(const std::string& thread_name)
{
    ++counter; ;//threadlocal, so no lock required
    std::lock_guard<std::mutex> lock(cout_mutex); //guard the output
    std::cout << "counter for " << thread_name << " in increment_counter: "
         << counter << std::endl;
}

void scale_counter(const std::string& thread_name, int scale)
{
    counter = counter * scale;//threadlocal, so no lock required
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << "counter for " << thread_name << "in scale_counter: "
        << counter << std::endl;
}


int main()
{
    increment_counter("main");
    std::thread a(increment_counter, "a"), b(scale_counter, "b", 5);

    a.join();
    b.join();

    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << "counter for main after join: " << counter << '\n';
}
```

At the beginning, the main thread obtains an instance of counter initialized to the value on the right side of the assignment (i.e. 1), then it increments it. Thread a is started and obtains its own instance of counter initialized to 1 which it then increments. Thread b also obtains its own instance of counter initialized to 1 and multiplies it by 5. After joining threads a and b, print the value of counter in the main thread.

In Java, ThreadLocal variables are provided via a class. The class has get and set methods. The initial value of a ThreadLocal variable is obtained by calling the initialValue method, which is executed the first time a get method on the variable is called by a thread, unless a set method has already been called. The method returns null, but can be overridden to behave as desired. Because ThreadLocals are classes in Java, primitive types cannot be made thread_local. There is also a remove method which removes the current thread's value for this thread-local variable.

Thread local variables in Java and C++ are implemented by maintaining a data structure within a thread that maps threadlocal variables to the value. Whenever a thread is terminated, the thread's instances are handled according to the way things are done in that language.

For this assignment, we are going to implement our own class to implement thread local variables in C++. There is no reason to do this rather than use the ones provided by C++, except this is an excellent exercise that also makes us aware of the underlying implementation so that we can make better cost benefit tradeoffs in using thread local variables.

We will need a slightly different scheme than the compiler uses. Each variable will encapsulate a map from thread to value, and the interface and behavior will be similar to Java's, except that we will not have an initial_value method—each thread will be responsible for initializing their instance by calling set. Calling get prior to calling set will result in an exception being thrown.

What you need to do:

1. Start with template below, which MUST be named threadLocal.h and fill in what is missing.

2. Create a file threadLocal_test.cpp which contains a parameterless function int test() that runs your test suite and returns the number of errors. This file should NOT contain a main function. Use the main function in the provided file HW1.cpp. Note that you should include tests that do not initialize the variable on some thread—correct behavior is throwing an exception that will be caught by your test. Also, make sure to test a variety of types.

We will run your threadLocal.h with your tests, and your threadLocal.h with our tests, and several of our buggy versions of threadLocal.h with your tests.

3. This is a pencil and paper question. You may find the following information useful if you used an STL container in your implementation: http://en.cppreference.com/w/cpp/container .

1. Which STL container, if any, did you use for you thread→value map?
2. Which pattern did you use (fully synchronized object, etc.)?
3. Consider the following pairs of operations:
    a. Set value where the thread does not have a preexisting value.
    b. Set value where the thread has a preexisting value
    c. Attempt to read a non-existing value
    d. Read an existing value
    e. Remove a value
    f. Attempt to remove a non-existing value

    Consider all possible pairs of operations by different threads, and indicate which ones **conflict**. For convenience in grading, please use the following list.

    a,a
    a,b
    a,c
    a,d
    a,e
    a,f
    b,b
    b,c
    b,c
    b,d
    b,e
    b,f
    c,c
    c,d
    c,e
    c,f
    d,d
    d,e
    d,f
    e,e
    e,f
    f,f
4. For each of the conflicting operations, in part 2, indicate whether it involves a data race.
5. (**COP5618 students only**) Consider whether applying some constraints on the way a threadLocal variable is used (for example: not allowing a set after a remove, or knowing the number of threads in advance, or other possibilities) would allow you to offer a correct implementation with less synchronization than the one you provided. You may change the public interface. If you think so, describe your solution and justify its correctness. If you think that it is not possible

to do this justify your answer. No implementation is required, just thinking. Hint: Make sure your answer is succinct and clear and use appropriate vocabulary.

```
#ifndef THREADLOCAL_H_
#define THREADLOCAL_H_

#include <iostream>

namespace cop5618 {

template <typename T>
class threadLocal {
public:
	threadLocal();
	~threadLocal();

	//disable copy, assign, move, and move assign constructors
	 threadLocal(const threadLocal&)=delete;
	 threadLocal& operator=(const threadLocal&)=delete;
	 threadLocal(threadLocal&&)=delete;
	 threadLocal& operator=(const threadLocal&&)=delete;
	/**
	 * Returns a reference to the current thread's value.
	 * If no value has been previously set by this
	 * thread, an out_of_range exception is thrown.
	 *
	 */
	const T& get() const;


	/**
	 * Sets the value of the threadLocal for the current thread
	 * to val.
```

```
         */
        void set(T val);

        /**
         * Removes the current thread's value for the threadLocal
         */
        void remove();

        /**
         * Friend function.  Useful for debugging only, shows values for all threads.
         */
        template <typename U>
        friend std::ostream& operator<< (std::ostream& os, const threadLocal<U>& obj);

private:

        ADD PRIVATE MEMBERS HERE
};

ADD IMPLEMENTATIONS OF THE PUBLIC METHODS AND THE << OVERLOAD HERE


} /* namespace cop5618 */

#endif /* THREADLOCAL_H_ */
```