CS 4414 Final – Solution Set

1. [20 pts, 2 pts per T/F question] **Locking.** Suppose that you are developing a multithreaded C++ program that will run on a NUMA computer. Here is a list of "assertions". Mark the ones that are true by writing "true" next to them, and the ones that are false by writing "false". If a question says something that is partly true but partly false, you would write false.

	True/False	
a	True	A data object that is read and updated by two or more threads will need to be declared as std::atomic or protected by a std::mutex object.
b	True	It is *not* necessary to use a std::mutex to protect shared objects marked as "const".
c	False	When using the std::scoped_lock type to create a lock, you do need to specify a mutex object but do not need to give the std::scoped_lock a variable name, because you would never perform any operations on the object.
d	False	C++ monitors are created using a special template called std::monitor that you instantiate using lambdas for the reader and writer logic, and for deadlock detection.
e	False.	With monitors, deadlock cannot occur.
f	False	If a program has lots of objects that need to protected, you would need one mutex *per object* (for example if x and y are distinct objects, and both are accessed by a mix of reader and writer threads, you would need at least two mutex objects, one for x and one for y).
g	False	We only consider a process to be deadlocked if *all* the threads in it are involved in cyclic waits. If some threads are in a cyclic wait but others are running, this is a livelock.
h	True	In contrast to C++ programs, transactional systems that have deadlocks will often detect the cycle, abort the transaction that caused it (this will roll back the changes it made to variables while it was running), and then automatically restart that transaction.
i	True	The readers and writers pattern can be used to safely protect an STL object like a std::list or std::map that will be read by some threads and written by other threads.
j	True	With a C++ bounded buffer we can support a mix of producer and consumer threads. The solution also limits how far producers can get ahead of consumers.

2. [10 pts, 2pts per T/F question] **MapReduce Pattern.** These questions relate to the MapReduce pattern. Again, write true if the statement is correct and false if it is not correct.

	True/False	
a	True	MapReduce is valuable if a data set is so large that it can't fit on any one computer and must be split into pieces ("sharded") and spread over many computers.
b	False	Unlike parallel computing, a MapReduce computation is sequential. Every shard will be processed but the computations occur one by one, with each worker running in turn in an order decided by the leader.
c	True	In MapReduce, the term *shuffle* is used for the step in which each worker sends subresults to each of the other workers.
d	True	Before applying the *reduce* function, each worker sorts its received set of subresults, then groups them by key, and then the reduce function's role is to collapse each vector of values to some single result.
e	False	After MapReduce finishes, every worker has a complete and identical copy of the output of the entire computation

3. [10 pts, 2pts per T/F question] Debugging with constexpr and templates. True or false…

	True/False	
a	True	Template code is expanded (as much as possible) at compile time. As a result, gdb and the profiler won't necessarily be able to associate bugs that cause a crash to the proper line within the template, or give proper runtime cost-accounting for templated methods.
b	True	A constexpr expression cannot include variables that hold values the program reads from a user or from some other kind of input.
c	True	In gdb or gprof, a variable with a templated type will often have more type-signature content than you used to define that variable, because of expansion of default template type parameters and argument.
d	False	If a constexpr performs a zero divide, then when you run C++ to compile the code, the compiler will exit with a zero-divide exception.
e	True	gprof won't count the time C++ spends evaluating a constexpr when it prints a formatted profile report for the program.

4. [5 points] **DLLs.**

a. [1 pt] Give a brief explanation in your own words of the "meaning" of the term DLL. We know that the letters stand for dynamically linked library. But what do DLLs do?

A DLL is a library of methods that can be called from a program that references those methods. DLLs are dynamically linked to the program, meaning they are memory mapped at runtime, which allows a single DLL to be shared by more than one process. Each instance would have its own private copy of data and heap objects, but would share the same code.

b. [1 pt] Why is it important to compile a DLL with the "position-independent code" (PIC) g++ compiler flag?

Linux can't guarantee that the DLL will be mapped to the same address range in different processes, which means that the code needs to work properly even if it is at base address 10000 in process A but at 25000 in process B. Position independent code is a compilation option which causes the g++ compiler to only generate machine instructions that will execute correctly no matter where the instruction resides. The DLL has an associated base address in each process, and this will be in a register, allowing a base-relative addressing style that the g++ compiler can exploit.

c. [3 pts] Suppose that A and B are two programs that share a DLL, and that it contains code but also some global variables. These objects are modified by the code as it runs. Do all the programs have private copies (so that A would see its own version, but not updates made by B), or shared copies (A sees B's updates)?

Each process has a private copy of any data needed by the DLL, so A can make changes and B's data will be untouched.

5. [10 points total] **Lambda Functions** For this problem, we will be using the following overload

of std::accumulate defined in C++'s *numeric* library:

```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );
```

This is the C++ equivalent of fold left that you might have seen in other contexts. Conceptually, it starts with an initial value of the "accumulator", and successively updates it by processing elements of a given sequence.

The first two arguments represent the range of the elements. For us, the range will be the entire sequence. For example, if our sequence is an std::vector<int> named v, we will pass in v.begin() and v.end().

The third argument is an initial value of the accumulator and the fourth argument is a function that takes the current accumulator and an element of the sequence, and returns the new value of the accumulator. For instance, we can sum up all elements of an std::vector<int> by passing 0 as the initial value and a lambda function that takes two integers (the first integer is the accumulator type, the second is the element type) and returns their sum.

For each of the following tasks, write a one line call to std::accumulate to accomplish them. You are welcome to test your code in a C++ program to make sure that the syntax compiles and that your solution works. You can also make the code boxes bigger if your solution needs a little more space.

a. **Modulo 7** Given an std::vector<int> object named *num*, where each element is a digit between 0 and 9 (inclusive), find the remainder mod 7 of the integer that has these same digits in sequence. For example, the number 635 can be represented as a vector of its digits 6, 3, and 5 (in that order). The result should be 5 (= 635 % 7).

```
accumulate(num.begin(), num.end(), 0, [](int a, int b){ return (a*10+b)%7; });
```

b. **Coversion to std::string** Given a std::vector<char> object named *str*, convert it to an object of std::string. For example, if the vector is ['a', 'b', 'c'] the result should be the string "abc".

```
accumulate(str.begin(), str.end(), std::string(),
             [](string a, char b){ return a+b; }
```

c. **Distance from origin** Given a std::vector<double> named *point*, which represents a point in the n-dimensional space, R^n, compute the square of its distance from the origin. For example, if the vector is [1.6, 1.2], the result should be 4.0 (= $1.6^2 + 1.2^2$) (or an equivalent double value).

```
accumulate(point.begin(), point.end(), 0.0,
              [](double a, double b){ return a + b*b; });
```

6. [10 points] Concurrency. Consider the following snippet of code:

```
std::atomic<int>     counter(0);
std::function<void()> increment = [&counter]() {
    for(int i = 0; i < 100000; ++i) { counter = counter + 1; }
};
std::vector<std::thread> workers(4);
for(int i = 0; i < 4; ++i) {
   workers[i] = std::thread(increment);
}
for(int i = 0; i < 4; ++i) {
   workers[i].join();
}
std::cout << "counter = " << counter << std::endl;
```

As you can see, 4 threads concurrently execute the increment function in which they update an atomic int, *counter* 100K times each. Will the output of the program be 400K? Explain. If you think the answer is no, suggest a one line fix that does not involve using any other forms of locking or synchronization (that is, just use the given atomic integer).

The program is incorrect because it breaks the operation on the atomic counter into an assignment: the expression on the right would normally be computed in a register, and then stored. This sequence could be interrupted by a context switch or by concurrent execution of some other thread, causing some increments to be lost. If the compiler realizes that it should compile this as a single instruction, the code would work, but if the compiler generates multiple instructions, a bug results.

A simple fix is to just recode the body of the lambda to say "counter++;" With a std::atomic integer, this should generate a single atomic machine instruction to increment the counter.

7. [10 pts, 2 pts per question] C++ true/false questions

	True/False	
a	False	Execution of ptr && *ptr will segfault for an integer pointer, *ptr*.
b	True	A functor in C++ is a class that defines the function call operator.
c	True	In order to use std::map<K,V>::operator [], for a user-defined class V, a default constructor for V must be implemented.
d	True	If a thread waits on a condition variable object, it may wake up even if it was not notified. For this reason it is important to recheck the wait condition.
e	False	Calling notify_one on a condition variable object will wake up the first thread that called wait on it (that is, notify_one guarantees a "FIFO" wake-up ordering).

8. [10 pts] C++ RAII (Resource Acquisition Is Initialization)

Consider a class *meeting* defined as followed:

```
class meeting {
// room where the meeting will be held std::shared_ptr<room> r;

public:
meeting(std::shared_ptr<room> r) : r(r) {}

void start() {
    r->turn_lights_on();
}

void end() {
    r->turn_lights_off();
}
};
```

Now suppose that you have been asked to modify this into an RAII implementation, in which allocating the object always turns the lights on, and they automatically go off when the object goes out of scope, Show us the revised code:

```
class meeting {
     public:

           meeting(std::shared_ptr<room> r): r(r) { start(); }
           ~meeting() { end(); }


           void start() {
               r->turn_lights_on();
           }

           void end() {
               r->turn_lights_off();
           }
 };
```