ECE498SL: Engineering Software Systems

Final Examination

14 May 2009

- You may use your brain and assorted writing implements.
- You may not use books, notes, calculators, other people's brains, etc.
- You may have one 8.5×11-inch page (double-sided) of handwritten notes.

tremorous: a person who insists that his/her delta is going
to change the world every time you meet them
—Websta's New Virtual World Dictionary, 2020

Most of the problems require a short response. If you choose to write a long response, say more than about 25 words, I'll probably stop reading after the short part at the front. Avoid pronouns: it makes it hard to know what it means.

Name: _____

A. The template mechanism in C++ performs all type checking and code instantiation at compile-time. Consider an alternate approach in which some of this work is performed at run-time.

A.1 (5 points): Explain one benefit of the approach taken by C++ relative to the alternative mentioned.

A.2 (5 points): Explain one drawback of the approach taken by C++ relative to the alternative mentioned.

B (5 points): Given the template function definition below, explain why the compiler reports an error for the **`twitter`** call within **`main`**, then explain how to fix the call.

```
template<class RVAL, class ARG>
RVAL twitter (const ARG& arg, const char* msg) {
    if (arg.exists () && arg.online ()) {
        return arg.send_msg (msg);
    }
    return RVAL (0);
}
int main ()
{
    UNIX_FRIEND joe_programmer;
    int rval;
    rval = twitter (joe_programmer,
                    "Try templates--they're cool!");
    std::cout << rval << std::endl;
}
```

C (5 points): In Lab #2, you developed a template to automatically count the number of references to a dynamically-allocated object. The code below creates a new ALPHA instance and then uses the resulting pointer to create two garbage-collected pointers to the new ALPHA. Explain why the code below is error-prone, and rewrite the code to avoid the pitfall.

```
ptr = new ALPHA (42);
gcp<ALPHA> gcp_ptr (ptr);
gcp<ALPHA> gcp_two (ptr);
// long procedure using these variables
```

D (5 points): ISAs such as x86 have instructions that are not guaranteed to execute atomically with respect to other instructions. Give an example and explain how this lack of atomicity might cause problems.

E (5 points): A single-producer, single-consumer FIFO queue provides a mechanism to enforce dependence between threads by requiring the consumer to wait for the producer to place something into the queue. However, dependence is also a critical concern (albeit relatively easily solved) within the design of such a queue. Explain why.

F. As you may recall, the bulk-synchronous paradigm (BSP, where P has many possible meanings) is an approach in which all threads in a program periodically stop and wait for one another (typically using barriers).

F.1 (5 points): Explain one benefit of using a BSP approach when writing large parallel programs.

F.2 (5 points): Explain one drawback of using a BSP approach. *Hint: one such drawback is likely to be more important on multi-core chips than on clusters.*

G (5 points): One approach to parallelism is to develop independent tasks, put them into a queue, and have processors dynamically claim and execute the tasks. If such tasks have variable execution time, this dynamic mechanism often provides better load balance (and thus overall execution time) than does a statically-scheduled approach. Similarly, use of many small tasks is often more effective than use of a few large tasks. Explain the cost in human terms of defining many small tasks for a parallel code (as opposed to cost of run-time overhead for queues, forwarding results, locality, etc.).

H. One way in which we can divide models of parallelism is by whether or not arbitrary threads can access data asynchronously with respect to one another. Consider, for example, a model in which each datum is associated with an owner thread, and accesses by threads other than the owner thread are only made at specific points in the owner thread's execution.

H.1 (5 points): Explain one benefit of the approach described relative to one in which all threads are allowed asynchronous access to all data.

H.2 (5 points): Explain one drawback of the approach described relative to one in which all threads are allowed asynchronous access to all data.

I. Consider a model of parallelism that forces the programmer to specify a total ordering on all operations (an extreme version of implicit parallelism, with no ambiguity in the language).

I.1 (5 points): Explain one benefit of such a constraint.

I.2 (5 points): Explain one drawback of such a constraint.

J (5 points): As you may recall, the Linux scheduler time slices "fairly" between all threads in a process. A friend of yours builds a ticket lock for use with a program based on Posix threads and finds using a profiler that threads are spending far more time than expected spinning on the ticket locks. Explain the interaction that might cause such behavior, and how use of Posix mutexes (with appropriate settings) circumvents this interaction.

K (5 points): A friend develops a parallel data structure for which many operations require reading large parts of the structure and then modifying a small portion. Frustrated by the lack of concurrency available with mutually exclusive access, your friend makes use of a reader-writer lock. Unfortunately, releasing the reader lock and then acquiring a writer lock in mid-operation does not meet the need for the data structure's operations to be atomic with respect to one another. Your friend wants to implement a "lock upgrade" operation in which a thread holding a read lock waits until it can convert its read lock to a write lock. Explain the problem with this idea.

L. Guarded atomic execution is an abstraction that allows a programmer to control execution of a particular piece of code by providing a *guard clause*, a language-level function of system state that evaluates to true or false. The code runs when its guard clause evaluates to true. Consider a run-time system in which each section of guarded code operates in a separate thread. These threads should sleep whenever their guard clause expressions are false.

L.1 (5 points): After a piece of code executes, the same code may need to execute again, but must first re-check its guard clause. If the clause is false, the thread should sleep until another thread wakes it (after having changed or potentially changed the value of the guard clause). Explain the need for atomicity in these actions and how this need can be addressed.

L.2 (5 points): In practice, only limited forms of guarded atomic execution have been implemented because it can be difficult to figure out which threads to wake up after changing some state. Explain why this problem matters and why it is hard to solve.

M (5 points): What was the most interesting thing that you found in parallelizing image segmentation for Lab #3? *Hint: these should be free points!*

N (5 points): The **`square_it`** operation below is based on the universal synchronization primitive compare-and-swap (CAS), for which equivalent code is also shown below. Prove that the implementation is non-blocking, meaning that some thread will make progress in a finite number of the overall system's time steps. *Hint: you should be able to write a rigorous proof of this property in a sentence.*

```
void square_it (int32_t* addr)
{
    int32_t old_val;
    int32_t new_val;

    do {
        old_val = *addr;
        new_val = old_val * old_val;
    } while (!CAS (addr, old_val, new_val));
}

int32_t CAS (int32_t* addr, int32_t old_val, int32_t new_val)
{
    if (old_val == *addr) {
        *addr = new_val;
        return 1;
    }
    return 0;
}
```

O (5 points): In lecture, we went over a wait-free implementation of fetch-and-increment and argued that checking for two changes by reading the variables twice sufficed to indicate completion. Many have argued for a vaguely similar approach to avoiding synchronization overhead in languages such as Java. Initialization of a static variable, for example, must be done exactly once; if, for example, an access function returns a reference to the static variable, the right approach in Java is to synchronize all calls to the access function with a mutex, and to allocate the variable on the first such call. The version below (in C++ with a Posix mutex) tries to avoid the overhead of always using a mutex. **`PTHREAD_MUTEX_INITIALIZER`** is a constant value for an unlocked lock. Explain how this code can fail. *Hint: it may help to assume that BETA's constructor is inlined.*

```
class ALPHA {
    private:
        static BETA* b;
        static pthread_mutex_t lock;
    public:
        const BETA& getBETA () {
            if (NULL == b) {
                (void)pthread_mutex_lock (&lock);
                if (NULL == b) {
                    b = new BETA (42);
                }
                (void)pthread_mutex_unlock (&lock);
            }
            return *b;
        }
};

BETA*           ALPHA::b    = NULL;
pthread_mutex_t ALPHA::lock = PTHREAD_MUTEX_INITIALIZER;
```

N.B. If you are interested, you can find a full treatise online on the problem of double-checked locking in Java.