ECE498SL: Engineering Software Systems

Final Examination

14 May 2010

• You may use your brain and assorted writing implements.

• You may not use books, notes, calculators, other people’s brains, etc.

• You may have two 8.5×11-inch pages (double-sided) of handwritten notes.

• This exam has 7 pages, including this one.

“He spoke. And drank rapidly a glass of water.”
—ee cummings

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 (5 points): Historically, the main reason that people spent time to write parallel code was to reduce execution time. Mention and briefly explain another reason that can drive people to write parallel code.

B (5 points): Recall the master-slave model of parallelism, in which a single master process assigns work to a number of slave processes and then collects and integrates their results. Explain how this model can be adapted to situations in which slave processes are unreliable or untrustworthy.

C: You are given a working parallel stack class in C++ with push and pop operations and asked to implement a function that atomically swaps the top elements of two stacks (atomically with respect to pushes/pops on either stack).

C.1 (5 points): The synchronization anomaly says that you will need to read and perhaps even modify some of the existing code in order to add this new function. Explain why in terms of the goal of atomicity.

C.2 (5 points): The stack class uses a lock field to synchronize push and pop operations. In particular, critical sections for a queue Q1 are wrapped with lock (&Q1->lock) and unlock (&Q1->lock). Explain how you can perform the desired swap operation on Q1 and Q2. Be careful: solutions that deadlock will not receive credit.
D: Consider the problem of extracting implicit parallelism from a language with sequential semantics. The C++ code below is intended to extract a subset of “Things” from an array, convert them to “Objects,” and store the objects in a hash table. The bins of the hash table need not be sorted in any way (in other words, if several objects have the same hash value, they can be chained together in any order in the bin).

```cpp
void fillHashTable (HashTable* h, Thing* t, int32_t nThings) {
    obj = new Object[nThings];
    for (int32_t i = 0; nThings > i; i++) {
        if (should_be_hashed (t[i])) {
            obj[i] = t[i]; // Object assign op. from const Thing&
            h.insert (&obj[i]);
        }
    }
}
```

The programmer who wrote the code believes that the loop iterations can be parallelized, provided that hash table insertions are performed atomically with respect to one another.

**D.1 (5 points):** Explain the need for mutually atomic hash table insertions.

**D.2 (10 points):** Sequential execution of the code produces two predictable orderings on the elements in each hash table bin based on the sequential order of loop iterations. Since either of these orderings may be semantically important to the programmer, a compiler cannot allow the loop iterations to execute fully in parallel. Explain the two orderings.

**D.3 (5 points):** Suggest a locking methodology that enables both orders to be maintained (the strategy need not be one that a compiler might reasonably be expected to follow).
Consider the execution of the two threads below. The operations inside the two critical sections are deliberately left vague. A race detection tool observes that the unlock operation in thread 1 happens before the lock in thread 2 and uses this fact to reason about data races.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 42;</td>
<td></td>
</tr>
<tr>
<td>lock (&amp;L);</td>
<td></td>
</tr>
<tr>
<td>// critical section 1</td>
<td>lock (&amp;L);</td>
</tr>
<tr>
<td>unlock (&amp;L);</td>
<td>// critical section 2</td>
</tr>
<tr>
<td></td>
<td>unlock (&amp;L);</td>
</tr>
<tr>
<td></td>
<td>printf (&quot;%d\n&quot;, X);</td>
</tr>
</tbody>
</table>

(happens before)

Does this program have a data race on the variable X? The answer depends on what happens inside the critical sections.

**E.1 (5 points):** Give an example of a program for which the two accesses to variable X form a data race. Describe the program in terms of the type of operations in the critical sections.

**E.2 (5 points):** Give an example of a program for which the two accesses to variable X must happen in the order shown above. Describe the program in terms of the type of operations in the critical sections.
**F (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.

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

**G:** Certain parallel languages use a model called *data-driven execution*, in which parallel tasks execute as soon as their operands are ready. Tasks can spawn other tasks by sending messages containing operands to data objects, and these new tasks can execute once the operands have arrived. The runtime manages and tracks motion of task operands and decides on the schedule of task execution. To keep complexity manageable, the tasks associated with any given data object are executed sequentially and atomically with respect to one another. As with monitors, the data in a data object are accessible (scoped) only within the operations on the data object; these operations, as mentioned, are invoked by sending a message to the object.

**G.1 (5 points):** Describe one advantage of data-driven execution relative to the shared memory threads model.

**G.2 (5 points):** Describe one disadvantage of data-driven execution relative to the shared memory threads model.
H (5 points): What was the most interesting thing that you found in parallelizing image segmentation for Lab #3? *Hint: these should be free points!*

I (5 points): A friend makes the following argument to you: when only two threads are involved, one can achieve non-blocking mutual exclusion using only loads and stores. Explain why your friend’s code, as shown below, is not non-blocking.

```c
// shared variables A and B, both initialized to 0
void lock_thread_A ()
{
    while (1) {
        A = 1;
        if (0 == B) { return; }
        A = 0;
    }
}

void lock_thread_B ()
{
    while (1) {
        B = 1;
        if (0 == A) { return; }
        B = 0;
    }
}

void unlock_thread_A ()
{
    A = 0;
}

void unlock_thread_B ()
{
    B = 0;
}
```

J (5 points): What is priority inversion, and how do the Posix mutex interfaces attempt to enable programmers to avoid it?
K (5 points): One of David Bailey’s ways to fool the masses is to…

“Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.”

Explain why such an approach might mislead readers in terms of the parallel speedup achieved by the results being reported.

L (5 points): With a fixed problem size, parallel speedup typically flattens and eventually declines with increasing number of processors. Explain why.

M: Parallel grain size refers to the amount of work involved in a single task that can be executed in parallel with other tasks. Metrics for work can be the number of instructions, the number of floating-point operations, or something similar. With most high-level problem specifications, several possible sources of parallelism are available, each with a different grain size.

M.1 (5 points): Explain one advantage of choosing smaller units of work (a smaller grain size) over larger units of work.

M.2 (5 points): Explain one advantage of choosing larger units of work (a larger grain size) over smaller units of work.