CSE PMP 548 – Computer Architecture – Homework #4

Some general notes:

- There are 5 questions. Each is worth 5 points. We'll give partial credit for partially correct answers.
- Try to be as thorough as possible, and justify your answers.
- If you rely on any assumptions to develop your answer, be sure to say what those assumptions are (and why they're reasonable!).
- If you have questions, feel free to contact the TA.

Problem #1 – Concurrent Programming Models

(a) *Shared memory* and *message passing* are fundamentally different models for multiprocessing. Each implies a different hardware implementation as well as a different view from the programmer's perspective. In about two paragraphs, discuss the differences between shared memory and message passing. Consider the aspects of *programmability* (ease and expressiveness of programming) and *performance*. (There's no need to argue that one model is "better" than the other; just discuss the trade-offs involved.)

(b) Shared-memory multiprocessors and message-passing computers are implementations of the above concurrency models. In addition, *vector machines* are computers that support "single instruction, multiple data" (SIMD) programs. For each of these three classes of machines, give one example of an application that is well-suited to the architecture. For each, briefly argue why the architecture in question is better-suited to the application than the other two architectures.

(c) The *barrier* synchronization primitive is a simple construct for coordinating multiple threads of execution. Describe a situation in which a barrier is an appropriate form of synchronization (i.e., more appropriate than other primitives such as mutual exclusion locks). Then describe a scenario in which the use of a barrier can lead to poor performance.

Problem #2 – Consistency Cook-Off

For the first part below, assume an architecture with a *Weak Ordering* memory consistency model:

- All loads and stores to different addresses can be reordered.
- Accesses to the same location cannot be re-ordered.
- Nothing can be reordered over a lock operation

(a) **Lazy Initialization & Double-Checked Locking**
There is a common idiomatic programming technique called "lazy initialization" of an object. A pointer is initially NULL, and remains NULL until its first use, at which point a new object is created, and the pointer is set to point to the new object. This obviously requires a NULL check before every access to this pointer—if it is NULL, then the object must be created, and if not, then the object can be used.

As an optimization for lazy initialization with multiple threads, some clever person invented "double-checked locking." In double-checked locking, the NULL check is made efficient by accessing the pointer outside of a critical region (i.e., without holding the lock associated with the pointer). If this check indicates the pointer is NULL, then the lock is acquired, and a second NULL check is performed (to be sure nothing changed between the first NULL check and the lock acquire). Then, under the protection of the lock, the object's constructor is called, and the pointer is set to point to the new object. The relevant code is listed below:

```
// Initially x = NULL and x is shared.
// Mutual exclusion is enforced with lock L.
...
if (x == NULL) {
    LOCK(L);
    if (x == NULL) {
        x = new X();
    }
    UNLOCK(L);
}
print x.a + " " + x.b + " " + x.c;
...
class X {
  ...
  X() {
    this.a = 0;
    this.b = 0;
    this.c = 0;
  }
}
```

Is this optimization safe? If not, what could possibly go wrong?

(b) **Identify the Outputs**

Consider this code for Thread 1 and Thread 2. Initially, `X == Y == 0` and X and Y are both shared by both threads.

Thread 1

```
X = 1
if(Y == 1)
    print "Y = 1"
else
    print "Y = 0"
```

Thread 2

```
Y = 1
if(X == 1)
    print "X = 1"
else
    print "X = 0"
```

What output is possible from executing this program on a multiprocessor with *Weak Ordering* consistency? What about one with with *Sequential Consistency*?

You can use a table like this one and check the boxes corresponding to possible outputs:

T1	T2	W.O.	S.C.
Y = 0	X = 0		
Y = 0	X = 1		
Y = 1	X = 0		
Y = 1	X = 1		

Problem #3 – Why Is This Program Slow?

You are TAing a class, and your students are writing their very first parallel program – It creates a couple of objects, and a couple of threads, and each thread manipulates each object. One student's code looks like this (in C-like code):

```
struct Foo {
    byte b[10];
}
...
main() {
    Foo f;
    Foo g;

    Thread t1 = createThread(doStuff, &f /*f is an arg to doStuff in t1*/);
    Thread t2 = createThread(doStuff, &g /*g is an arg to doStuff in t2*/);

    t1.start();
    t2.start();
}
...
doStuff (Foo *foo) {
    for(i in 1 to 1000){
        for(j in 0 to 9){
            foo->b[j]++;
        }
    }
}
```

The code is pretty simple. However, because of the budget cuts at UW this year, you're forced to compile your code with GCC--, a cheap knockoff of the GCC compiler written by a bunch of undergrads at Rural Ontario Regional Mining Institute (RORMI). You happen to have a real copy of GCC on your personal computer, and you noticed that when compiled using GCC, the performance of the application is significantly better than with GCC--. Explain a possible reason for this, and how, if you were stuck with GCC--, you could change your code to eliminate the poor performance.

Problem #4 – Sequentially Consistent Hardware

Suppose you have a multiprocessor that guarantees sequential consistency. Describe how a programmer could still end up with violations of sequential consistency. Include a code example if you can, and discuss the cycle in the happens-before graph. *Hint: Think about memory ordering at all levels of the system stack.*

Problem #5 – Efficient Coherence

In an implementation of the MSI cache coherence protocol, the initial state of a line L, currently uncached by any processor in the system, must be specified somehow when it is first cached by some processor P. That is, if P is the first processor to request a copy of L, should P get L in M state or in S state? Describe two situations: one in which it would be advantageous to always hand out lines in M state, and one in which it would be advantageous to always hand out lines in the S state. Which of these policies seems like it would have better performance, and why?