CSE 548 – Computer Architecture – Homework #3

Due on the last day of class (March 10th, 2010)

1. Problem #1 – Consistency Cook-Off

   For these two problems, 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

   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 intialization 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 NULL 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
For this problem, you will look at the code for Thread 1 and Thread 2. List all the different possible outputs of this code. Use a table like the one below to indicate the valid outputs.

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"
```

(c) Sequential Consistency
Redo part (b), but assume Sequential Consistency instead of Weak Ordering.

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

Table 1: Fill out this table for parts (b) and (c). There is a row for each possible outcome – Columns 1 & 2 show what is printed in each of the cases.

2. Problem #2 – Synchronization and Coherence
For this problem, you're going to think about how synchronization and coherent caches interact with one another.

Imagine you're designing a new multiprocessor system. Each processor has a cache, and they are kept coherent using an MESI coherence protocol.

Your architecture uses Load-Linked/Store-Conditional (LL/SC) to implement atomic test-and-set operations. An LL instruction reads a value, and causes the thread to hold a "reservation" for that memory location. When the thread executes an SC instruction , if the thread still holds the reservation for that location, then the update is made. If any other thread has updated the memory location in the meantime, the first thread's SC will fail. This ensures that the test-and-set update made using LL/SC is atomic. The pseudo-code for acquiring a lock looks like this:

```
do{

    do{
        //Spin in a loop waiting for the lock to be unheld
        lock_state = LoadLinked(L)
    }while(lock_state != 0)

    //Store a ''1'' in the lock. StoreConditional returns 1 on success, and 0 on failure
    acq_result = StoreConditional(1,L)

}while(acq_result != 1)
```

You decide to use this mechanism to implement mutual exclusion locks. A lock is a data word in memory, and to acquire it, a thread spins, reading the value of the lock with LL until it is "0" (unheld). Then, the thread executes an SC, attempting to write a "1" to the lock. On a successful SC, the thread has acquired the lock, and can enter the critical region. If the SC fails, then the thread must continue to loop. As a convenience, each lock word has a "syncbit" – a single bit that you can read to identify which lines hold locks, and this bit moves around with the lines themselves, as well as with requests for the lines.

Finally, because you're clever, and you like things to take longer sometimes, you have designed your caches in such a way that they can delay coherence requests for a short, bounded interval after their arrival.

Your goal is to use this delay mechanism to *optimize* the performance of your multiprocessor system. Upon the receipt of a coherence request, when is it advantageous to delay that request, and why would it be advantageous?

3. 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 bootleg 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 was 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.

4. 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*

5. 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 higher performance, and why?

General Notes

- Each problem is worth 20 points. We'll give partial credit for partially correct answers.
- Please write up your solutions (preferrably using latex), and submit them as a pdf file.
- 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.