CSE 471 Homework #3: Concurrency

Due Thursday, May 24 before class

1. **Problem #1 – Consistency Cook-Off**
   For these three 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.
   - Accesses cannot be moved before a lock operation or after an unlock operation.
   - Write atomicity is ensured: if any processor reads the result of a write, then all subsequent reads by any other processor must see the same result, or a later result.

   (a) Lazy Initialization
   There is a common idiomatic programming technique called "lazy initialization" of an object. A shared 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 NULL check indicates that the pointer is NULL, then the lock is acquired, and a second NULL check is performed inside the critical section (to be sure nothing changed between the first NULL check and the lock acquire). This checking scheme is similar to test-and-test-and-set, which we learned about in class. Under the protection of the lock, a new object is allocated and initialized, and the shared pointer is updated 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 tmp = new X();
        tmp.a = 0;
        tmp.b = 0;
        tmp.c = 0;
        x = tmp;
    }
    UNLOCK(L);
}
print ""+x.a+" "+x.b+" "+x.c;
```

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

(b) Identify the Outputs

For this problem, consider a weakened version of the weak ordering memory model: *weaker ordering*. Weaker ordering permits the reorderings permitted by weak ordering, but does not provide *write atomicity*. Recall that write atomicity ensures that if any processor reads the result of a write, then all other processors must also read that value.

Look at the code for Threads 1, 2, 3 and 4. Threads 3 and 4 are reading the shared variables X and Y into their registers, r1, r2, r3 and r4. Initially, X and Y are 0. Fill in the W.O. column in the table below to indicate the valid output register values under the weaker ordering memory model. If the outcome is valid, mark the table cell with a check; if it is invalid, leave it empty.

Thread 1	Thread 2	Thread 3	Thread 4
`X = 1`	`Y = 1`	`r1 = X`	`r3 = Y`
		`r2 = Y`	`r4 = X`

r1	r2	r3	r4	W.O.	S.C.
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

Table 1: Fill out this table for parts (b) and (c). There is a row for each possible outcome.

(c) Sequential Consistency

Redo part (b), but assume Sequential Consistency instead of Weaker Ordering. Fill in the S.C. column in the table below.

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 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 section. If the SC fails, then the thread must try again.

As a convenience, each lock word has a "syncbit" – a single bit that you can read to identify which cache lines hold locks. This bit moves around with the cache line that contains a lock, and is included in all coherence requests for the line.

Finally, you have designed your caches in such a way that they can delay coherence requests from other processors 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();

    join(t1,t2);
}
...
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 that has a low optimization level, 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.

General Notes

- Each problem is worth 20 points. We'll give partial credit for partially correct answers.
- Please write up your solutions 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!).