CSC 258/458 Written Assignment

Due Monday, February 24, 2014

1. (10 points) We say that write-after-read and write-after-write dependencies are *not* true dependencies. Why? Provide sufficient details to justify your answer.

   Answer sketch: Consider write-after-read and write-after-write dependencies on a named variable. The dependencies can be removed if the following read/write is redirected to a new location (away from the target location of the preceding write). To ensure correctness, subsequent reads/writes to the named variable should access the new location. The register renaming technique performs this optimization.

2. (10 points) A cache coherence protocol may invalidate a cached data item at processor $\mathcal{X}$ that is being modified by another processor $\mathcal{Y}$. Alternatively, the cache coherence protocol can update the cached copy while keeping it valid. What is the advantage of updating over invalidation? What is the disadvantage of updating?

   Answer sketch: The advantage of updating is that if processor $\mathcal{X}$ later tries to read the data, it would be available in the local cache. Its disadvantage is that if processor $\mathcal{Y}$ modifies the data repeatedly before processor $\mathcal{X}$ reads it (if at all), then the system will pay the overhead of updating with little benefit.

3. (10 points) Consider the following execution of two parallel programs on a shared-memory multiprocessor.

```
                /* Initially flag1 = flag2 = 0 */

/* P1 */                            /* P2 */
flag1 = 1;                         flag2 = 1;
turn = 2;                          turn = 1;
while (flag2 && turn==2) ;         while (flag1 && turn==1) ;

/* ... critical section */         /* ... critical section */

flag1 = 0;                         flag2 = 0;
```

   Assume that the processor and compiler does not reorder any memory operations. However, the multiprocessor has processor-local caches **without** cache coherence. Does the above program guarantee mutually exclusive executions of the critical sections (e.g., the processes do not enter their respective critical sections at the same time)? Explain your answer.

   Answer sketch: Without cache coherence, the above program does not guarantee mutual exclusion. Specifically, if P1 updates its local cached copy of flag1 without being seen by P2, and that P2 updates its local cached copy of flag2 without being seen by P1, then both processes may enter their critical sections at the same time.

4. (10 points) Consider the following execution of three parallel programs on a shared-memory multiprocessor.

```
                        /* Initially A = B = 0 */

/* P1 */                /* P2 */                                /* P3 */
A = 1;                  while (A == 0) ; /* wait */             if (B == 1)
                        B = 1;                                      output(A);
```

If the multiprocessor supports sequential memory consistency, what are possible outputs for P3?

Answer sketch: If P3 does produce an output, it must output 1. Sequential memory consistency means that memory accesses effectively follow a global sequential order across all processes. P3's output instruction, if it runs, must be after P2's instruction `B = 1`, which must be after P1's instruction `A = 1`. This ordering means P3 must output 1.

5. The atomic instruction `test_and_set` assigns a value to a location and returns the old value of the location. Consider the lock/unlock routines below that protect a critical section (acquire lock before entering; release lock after leaving).

```
acquire_lock(L *location) {
    while (test_and_set(location, locked) == locked) {
        while (*location == locked) ;
    }
}

release_lock (L *location) {
    *location = unlocked;
}
```

   (a) (10 points) If the multiprocessor supports sequential memory consistency, show that the above lock/unlock routines ensure mutual exclusion of protected critical sections.

   Answer sketch: Sequential memory consistency means that memory accesses effectively follow a global sequential order across all processes. Each `acquire_lock` succeeds when the `test_and_set` instruction returns `unlocked`. Therefore each critical section execution is preceded by a `test_and_set` instruction that returns `unlocked`. In the sequential ordering of memory operations, two `unlocked`-returning `test_and_set`s must be separated by at least an unlock instruction in `release_lock`. Therefore the mutual exclusion of critical sections is ensured.

   (b) (10 points) Dr. Foobar says that the second while loop in `acquire_lock` is unnecessary for correct synchronization. Is Dr. Foobar right? Explain your answer.

   Answer sketch: Dr. Foobar is right. The correctness argument for the last sub-question does not need the second while loop in `acquire_lock`.

   (c) (10 points) Dr. Foobar also says that the second while loop in `acquire_lock` may help improve the synchronization performance. Is Dr. Foobar right? Explain your answer.

   Answer sketch: Dr. Foobar is right. `test_and_set` instructions incur traffic on the memory bus. The second while loop in `acquire_lock` helps to reduce such memory bus traffic, and consequently it may improve the synchronization performance.

6. (10 points) One of the earliest "read-modify-write" atomic instructions is `compare_and_swap`. The instruction takes three operands: the location to be modified, a value that the location is expected to contain, and a new value to be placed there if (and only if) the expected value is found. The instruction returns an indication of whether it succeeded. Show that `compare_and_swap` can be used to emulate `test_and_set` which atomically assigns a value to a location and returns the old value of the location.

   Answer sketch:

```
test_and_set(location, value) {
    R2 = value;
    do {
        R1 = *location;
    }
    while (! compare_and_swap(location, R1, R2)) ;
    return R1;
}
```