Solutions to Quiz 07

Problem 1

Consider a 2MB L2 cache with a 32-byte cache line. If the virtual address space is 64-bit, and the physical address space is 44-bit, compute the size of a single cache line tag if the cache is direct mapped. What if the cache is 2-way set associative?

The physical address has a mask that identifies the correct cache line. Once a cache line is identified, we compare the bits in the tag with the address. But we do not need to store the offset within the cache line, so we can conserve 5 bits. Similarly, we do not need to compare the bits that constitute a mask (they have been used to identify the cache line already). So the necessary number of bits is $44 - 5 - \log_2(2^{21}/2^5) = 44 - 5 - 16 = 23$ bits [$\log_2(2^{21}/2^5)$ is the number of bits in the mask that is used to identify which cache line we should use for comparison]. With a 2-way set-associative, the number of entries in the cache line is halved, but the comparison needs to still consider all the bits, so the size of the entries is 23 bits.

Problem 2

Is the following a good idea?

```
const n = 0x100000;
double shared[n];
main() {
        t0 = CreateThread(Compute, 0);
        t1 = CreateThread(Compute, 1);
        JoinThread(t0);
        JoinThread(t1);
        cout << shared << endl;
}
Void Compute(int my_id) {
        for(i = my_id; i < n; i+= 2)
                shared[i] = Gamma(32.0, i);
}
```

If it is a good idea, explain why. If not, clean it up.

This code allows two threads to share the operation of updating a vector using a function called Gamma. It turns out that this will perform very poorly because each thread operates on every other element of the vector. On any modern cache, many of these adjacent elements will be placed in the same cache lines, causing an excessive number of invalidation requests between

the two threads as they continuously update the vector. It would be better to split the vector into two independent ones and give each thread an independent vector to compute.

Problem 3

You have been hired to evaluate a design for a cache coherence protocol. The designer theorized that if caches are made to be write-through, then the memory copy of any data item will be always up to date. Other caches that want to access the data can easily get it from main memory in this case. This can simplify the cache coherence protocol. To read an item that is not in the cache, it is simply fetched from main memory. Any updates are made directly to the main memory. The performance degradation due to using write through caches can be justified by the simplicity obtained by eliminating the complex cache coherence protocol. What do you think?

At face value, it appears that this protocol will eliminate the need for a coherence protocol, since all memory values are updated synchronously. However, this does not work. It is possible for two threads to violate the sequential consistency model and provide incoherent data. For example, assume that the threads are sharing a variable "a" like below:

```
a = a + 3;
```

The two threads can both load the memory location of "a" and proceed to do the updates. If thread 0 updates the variable "a" in memory, the copy in thread 1's cache becomes obsolete. So, thread 1 will use an "old" value of "a", violating the coherence requirements. For the algorithm above to work, the variable "a" should simply not be cached. In this case, every read will indeed get the value from memory, which is updated with any writes. This will provide cache coherence (although the programmer will have to ensure concurrency control at the program level). However, this essentially eliminates caching, which will give us serious performance degradation.

Problem 4

```
#include <iostream>
#include <vector>
#include <thread>
int f()
{
    static int i = 0;
    synchronized { // begin synchronized block
        std::cout << i << " -> ";
        ++i;       // each call to f() obtains a unique value of i
        std::cout << i << '\n';
        return i; // end synchronized block
    }
```

```
}
int main()
{
    std::vector<std::thread> v(10);
    for(auto& t: v)
        t = std::thread([]{ for(int n = 0; n < 10; ++n) f(); });
    for(auto& t: v)
        t.join();
}
```

In the code above, each of 10 threads produces 10 unique numbers concurrently. Can you identify any case of false sharing?

From the code above, we cannot identify a case of false sharing. The variable “n” is local to every thread and is typically allocated on the thread’s stack. These are sufficiently allocated far from each other.

The variable “i” is actually shared among all threads. But this is legitimate sharing as it serves the function required from the program.

The last data item here is the vector “v”. It is not clear from the code if the threads access the vector “v” at all (the beauty of using libraries, where you have no clue what the programmer did and will have to chase the documentation ☺).