ECE459: Programming for Performance Winter 2015

Lecture 20 — February 25, 2015

Patrick Lam *version 1*

Memory Consistency, Memory Barriers, and Reordering

Today we'll talk a bit more about memory consistency, memory barriers and reordering in general. We'll start with instruction reordering by the CPU and move on to reordering initiated by the compiler. I'll also touch on some CPU instructions for atomic operations.

Memory Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model.

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." — Leslie Lamport

What are the possible values for the variables?

Another view of sequential consistency:

- each thread induces an *execution trace*.
- always, the program has executed some prefix of each thread's trace.

It turns out that sequential consistency is too expensive to implement. (Why?) So most systems actually implement weaker memory models, such that both `r1` and `r2` might end up unchanged. Recall the **flush** example from last time.

Reordering. Compilers and processors may reorder non-interfering memory operations within a thread. For instance, the two statements in `T1` appear to be independent, so it's OK to execute

them—or, equivalently, to publish their results to other threads—in either order. Reordering is one of the major tools that compilers use to speed up code.

When is reordering a problem?

Memory Consistency Models

Here are some flavours of memory consistency models:

- Sequential consistency: no reordering of loads/stores.
- Sequential consistency for datarace-free programs: if your program has no data races, then sequential consistency.
- Relaxed consistency (only some types of reorderings):
  - Loads can be reordered after loads/stores; and
  - Stores can be reordered after loads/stores.
- Weak consistency: any reordering is possible.

In any case, **reorderings** only allowed if they look safe in current context (i.e. they reorder independent memory addresses). That can still be problematic, though.

Compilers and reordering. When it can prove that a reordering is safe with respect to the programming language semantics, the **compiler** may reorder instructions (so it's not just the hardware).

For example, say we want thread 1 to print value set in thread 2.

```
f = 0
```

```
/* thread 1 */                          /* thread 2 */
while (f == 0) /* spin */;               x = 42;
printf("%d", x);                         f = 1;
```

If thread 2 reorders its instructions, will we get our intended result? *No!*

Memory Barriers

We previously talked about OpenMP barriers: at a `#pragma omp barrier`, all threads pause, until all of the threads reach the barrier. Lots of OpenMP directives come with implicit barriers unless you add `nowait`.

A rather different type of barrier is a *memory barrier* or *fence*. This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occuring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible.

The x86 architecture defines the following types of memory barriers:

- `mfence`. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- `sfence`. All stores before the barrier become visible before all stores after the barrier become visible.
- `lfence`. All loads before the barrier become visible before all loads after the barrier become visible.

Note, however, that while an `sfence` makes the stores visible, another CPU will have to execute an `lfence` or `mfence` to read the stores in the right order.

Consider the example again:

```
f = 0

/* thread 1 */                    /* thread 2 */
while (f == 0) /* spin */;        x = 42;
// memory fence                  // memory fence
printf("%d", x);                  f = 1;
```

This now prevents reordering, and we get the expected result.

You can use the `mfence` instruction to implement *acquire barriers* and *release barriers*. An acquire barrier ensures that memory operations after a thread obtains the mutex doesn't become visible until after the thread actually obtains the mutex. The release barrier similarly ensures that accesses before the mutex release don't get reordered to after the mutex release. Note that it is safe to reorder accesses after the mutex release and put them before the release.

Preventing Memory Reordering in Programs: Compiler Barriers. First: Don't use volatile in C/C++ on variables [1]. However, you can prevent reordering using compiler-specific calls.

- Microsoft Visual Studio C++ Compiler:

  ```
  _ReadWriteBarrier()
  ```

- Intel Compiler:

  ```
  __memory_barrier()
  ```

- GNU Compiler:

  ```
  __asm__ __volatile__ ("" ::: "memory");
  ```

The compiler also shouldn't reorder across e.g. Pthreads mutex calls.

[1] http://stackoverflow.com/questions/78172/using-c-pthreads-do-shared-variables-need-to-be-volatile.

Aside: `gcc` Inline Assembly. Just as an aside, here's `gcc`'s inline assembly format

```
__asm__ ( assembler template
        : output operands                   /* optional */
        : input operands                    /* optional */
        : list of clobbered registers       /* optional */
        );
```

Note that we've just seen **__volatile__** with __asm__. This isn't the same as the normal C volatile. It means:

- The compiler may not reorder this assembly code and put it somewhere else in the program.

Back to Memory Reordering in Programs. Fortunately, an OpenMP **flush** (or, better yet, mutexes) also preserve the order of variable accesses. Stops reordering from both the compiler and hardware. For GNU, flush is implemented as `__sync_synchronize();`

`volatile`**.** This qualifier ensures that the code does an actual read from a variable every time it asks for one (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Note: proper use of memory fences makes `volatile` not very useful (again, `volatile` is not meant to help with threading, and will have a different behaviour for threading on different compilers/hardware).

Atomic Operations

We saw the **atomic** directive in OpenMP as well as C++11 atomics. Most OpenMP atomic expressions map to atomic hardware instructions. However, other atomic instructions exist, and we've seen the C++11 ones earlier as well.

Compare and Swap. This operation is also called **compare and exchange** (implemented by the `cmpxchg` instruction on x86). Here's some pseudocode for it.

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

Afterwards, you can check if the CAS returned `oldval`. If it did, you know you changed it.

Implementing a Spinlock. You can use compare-and-swap to implement spinlock:

```
void spinlock_init(int* l) { *l = 0; }

void spinlock_lock(int* l) {
    while(compare_and_swap(l, 0, 1) != 0) {}
    __asm__ ("mfence");
}

void spinlock_unlock(int* int) {
    __asm__ ("mfence");
    *l = 0;
}
```

You'll see **cmpxchg** quite frequently in the Linux kernel code.

ABA Problem

Sometimes you'll read a location twice. If the value is the same both times, nothing has changed, right?

No. This is an **ABA problem**.

You can combat this by "tagging": modify value with nonce upon each write. You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce.

Just something to be aware of. "Not on exam".