ECE459: Programming for Performance Winter 2014

Lecture 14 — February 27, 2014

Patrick Lam *version 1*

1 Locking Granularity

Locks prevent data races.

However, using locks involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead and is more prone to bugs. Locks' extents constitute their **granularity**. In coarse-grained locking, you lock large sections of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections.

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e. remaining race conditions).

Lock Overhead. Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention. Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks. Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Consider, for instance, two processors trying to get two locks.

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**. You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
void f1() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

```
void f2() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it's harder to ensure a consistent deadlock when lock identity is not statically visible.

Alternately, you can use trylock. Recall that Pthreads' `trylock` returns 0 if it gets the lock. So, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

```
void f1() {
    lock(&l1);
    while (trylock(&l2) != 0) {
        unlock(&l1);
        // wait
        lock(&l1);
    }
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

(Incidentaly, using trylocks can also help you measure lock contention.)

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). This is *coarse-grained locking*. It does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

It also, however, has one big disadvantage in terms of programming for performance:

- your parallel program will quickly become sequential.

Example: Python (and other interpreters). Python puts a lock around the whole interpreter (known as the *global interpreter lock*). This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for IO.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking.* The big advantage:

- it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Examples. The Linux kernel used to have **one big lock** that essentially made the kernel sequential. This worked fine for single-processor systems! The introduction of symmetric multiprocessor systems (SMPs) required a more aggressive locking strategy, though, and the kernel now uses finer-grained locks for performance.

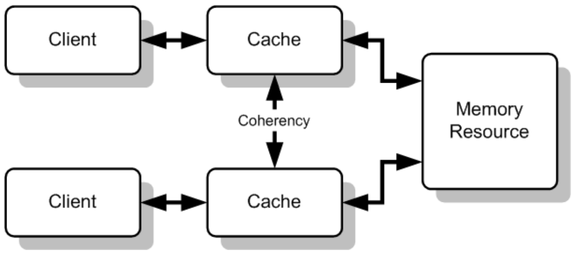
Databases may lock fields / records / tables. (fine-grained $\rightarrow$ coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

Live Coding Example. I ran the code from the midterm (augmented with code to populate the tree) using 1) a coarse-grained lock; and 2) per-item fine-grained locks.

Fine-grained locks were suspiciously fast.

Cache Coherency



—Wikipedia

We talked about memory ordering and fences last time. Today we'll look at what support the architecture provides for memory ordering, in particular in the form of cache coherence. Since this isn't an architecture course, we'll look at this material more from the point of view of a user, not an implementer.

Cache coherency means that:

- the values in all caches are consistent; and
- to some extent, the system behaves as if all CPUs are using shared memory.

Cache Coherence Example. We will use this example to illustrate different cache coherence algorithms and how they handle the same situation.

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache.
2. `CPU3` reads x, puts the value in its cache.
3. `CPU3` modifies `x := 42`
4. `CPU1` reads x ... from its cache?
5. `CPU2` reads x. Which value does it get?

Unless we do something, `CPU1` is going to read invalid data.

High-Level Explanation of Snoopy Caches. The simplest way to "do something" is to use snoopy caches. The setup is as follows:

- Each CPU is connected to a simple bus.
- Each CPU "snoops" to observe if a memory location is read or written by another CPU.
- We need a cache controller for every CPU.

Then:

- Each CPU reads the bus to see if any memory operation is relevant. If it is, the controller takes appropriate action.

Write-Through Caches

Let's put that into practice using write-through caches, the simplest type of cache coherence.

- All cache writes are done to main memory.
- All cache writes also appear on the bus.
- If another CPU snoops and sees it has the same location in its cache, it will either *invalidate* or *update* the data.
  (We'll be looking at invalidating.)

Normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**).

Write-Through Protocol. The protocol for implementing such caches looks like this. There are two possible states, **valid** and **invalid**, for each cached memory location. Events are either from a processor (**Pr**) or the **Bus**. We then implement the following state machine.

State	**Observed**	**Generated**	**Next State**
Valid	PrRd		Valid
Valid	PrWr	BusWr	Valid
Valid	BusWr		Invalid
Invalid	PrWr	BusWr	Invalid
Invalid	PrRd	BusRd	Valid

Example. For simplicity (this isn't an architecture course), assume all cache reads/writes are atomic. Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache. (valid)
2. `CPU3` reads x, puts the value in its cache. (valid)
3. `CPU3` modifies `x := 42`. (write to memory)
   - `CPU1` snoops and marks data as invalid.
4. `CPU1` reads x, from main memory. (valid)
5. `CPU2` reads x, from main memory. (valid)

Write-Back Caches

Let's try to improve performance. What if, in our example, `CPU3` writes to `x` 3 times?

Main goal. Delay the write to memory as long as possible. At minimum, we have to add a "dirty" bit, which indicates the our data has not yet been written to memory. Let's do that.

Write-Back Implementation. The simplest type of write-back protocol (MSI) uses 3 states instead of 2:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Shared**—location is unmodified, up-to-date with main memory; may be present in other caches (also up-to-date).
- **Invalid**—same as before.

The initial state for a memory location, upon its first read, is "shared". The implementation will only write the data to memory if another processor requests it. During write-back, a processor may read the data from the bus.

MSI Protocol. Here, bus write-back (or flush) is **BusWB**. Exclusive read on the bus is **BusRdX**.

State	**Observed**	**Generated**	**Next State**
Modified	PrRd		Modified
Modified	PrWr		Modified
Modified	BusRd	BusWB	Shared
Modified	BusRdX	BusWB	Invalid
Shared	PrRd		Shared
Shared	BusRd		Shared
Shared	BusRdX		Invalid
Shared	PrWr	BusRdX	Modified
Invalid	PrRd	BusRd	Shared
Invalid	PrWr	BusRdX	Modified

MSI Example. Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x from memory. (BusRd, shared)
2. `CPU3` reads x from memory. (BusRd, shared)
3. `CPU3` modifies `x = 42`:

   - Generates a BusRdX.
   - `CPU1` snoops and invalidates x.

4. `CPU1` reads x:
   - Generates a BusRd.
   - `CPU3` writes back the data and sets x to shared.
   - `CPU1` reads the new value from the bus as shared.

5. `CPU2` reads x from memory. (BusRd, shared)

An Extension to MSI: MESI

The most common protocol for cache coherence is MESI. This protocol adds yet another state:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Exclusive**—only this cache has a valid copy; main memory is **up-to-date**.
- **Shared**—same as before.
- **Invalid**—same as before.

MESI allows a processor to modify data exclusive to it, without having to communicate with the bus. MESI is safe. The key is that if memory is in the E state, no other processor has the data.

MSEIF: Even More States!

MESIF (used in latest i7 processors):

- **Forward**—basically a shared state; but, current cache is the only one that will respond to a request to transfer the data.

Hence: a processor requesting data that is already shared or exclusive will only get one response transferring the data. This permits more efficient usage of the bus.

Cache coherence vs flush

> Cache coherency seems to make sure my data is consistent. Why do I have to have something like flush or fence?

Sadly, no. Cache coherence isn't enough. Writes may be to registers rather than memory, and those won't be coherent. Use fences or flushes.

> Well, I read that `volatile` variables aren't stored in registers, so then am I okay?

Again, sadly, no. Recall that `volatile` in C was only designed to:

- Allow access to memory mapped devices.
- Allow uses of variables between `setjmp` and `longjmp`.
- Allow uses of `sig_atomic_t` variables in signal handlers.

Remember, things can also be reordered by the compiler, and `volatile` doesn't prevent reordering. Also, it's likely your variables could be in registers the majority of the time, except in critical areas.

Coherence summary. We saw four cache coherence protocols, from MSI through MESIF. There are many other protocols for cache coherence, each with their own trade-offs.

Recall: OpenMP flush acts as a **memory barrier/fence** so the compiler and hardware don't reorder reads and writes.

- Neither cache coherence nor `volatile` will save you.