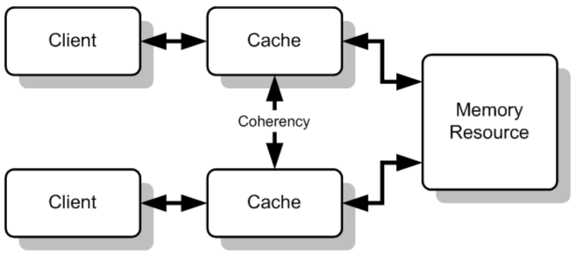
ECE459: Programming for Performance Winter 2013

Lecture 13 — February 26, 2013

Patrick Lam *version 1*

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 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.

Building Servers: Concurrent Socket I/O

Switching gears, we'll talk about building software that handles tons of connections. From a Quora question:

> What is the ideal design for server process in Linux that handles concurrent socket I/O?

So far in this class, we've seen:

- processes;
- threads; and
- thread pools.

We'll analyze the answer by Robert Love, Linux kernel hacker[1].

[1] `https://plus.google.com/105706754763991756749/posts/VPMT8ucAcFH`

The Real Question.

How do you want to do I/O?

The question is not really "how many threads should I use?".

Your Choices. The first two both use blocking I/O, while the second two use non-blocking I/O.

- Blocking I/O; 1 process per request.
- Blocking I/O; 1 thread per request.
- Asynchronous I/O, pool of threads,
  callbacks,
  each thread handles multiple connections.
- Nonblocking I/O, pool of threads,
  multiplexed with select/poll, event-driven,
  each thread handles multiple connections.

Blocking I/O; 1 process per request. This is the old Apache model.

- The main thread waits for connections.
- Upon connect, the main thread forks off a new process, which completely handles the connection.
- Each I/O request is blocking, e.g., reads wait until more data arrives.

Advantage:

- "Simple to undertand and easy to program."

Disadvantage:

- High overhead from starting 1000s of processes. (We can somewhat mitigate this using process pools).

This method can handle ∼10 000 processes, but doesn't generally scale beyond that, and uses many more resources than the alternatives.

Blocking I/O; 1 thread per request. We know that threads are more lightweight than processes. So let's use threads instead of processes. Otherwise, this is the same as 1 process per request, but with less overhead. I/O is the same—it is still blocking.

Advantage:

- Still simple to understand and easy to program.

Disadvantages:

- Overhead still piles up, although less than processes.
- New complication: race conditions on shared data.

Asynchronous I/O. The other two choices don't assign one thread or process per connection, but instead multiplex the threads to connections. We'll first talk about using asynchronous I/O with select or poll.

Here are (from 2006) some performance benefits of using asynchronous I/O on lighttpd[2]:

version		fetches/sec	bytes/sec	CPU idle
1.4.13	sendfile	36.45	3.73e+06	16.43%
1.5.0	sendfile	40.51	4.14e+06	12.77%
1.5.0	linux-aio-sendfile	72.70	7.44e+06	46.11%

(Workload: 2 × 7200 RPM in RAID1, 1GB RAM, transferring 10GBytes on a 100MBit network).

The basic workflow is as follows:

1. enqueue a request;
2. ... do something else;
3. (if needed) periodically check whether request is done; and
4. read the return value.

Some code which uses the Linux asynchronous I/O model is:

```
#include <aio.h>

int main() {
    // so far, just like normal
    int file = open("blah.txt", O_RDONLY, 0);

    // create buffer and control block
    char* buffer = new char[SIZE_TO_READ];
    aiocb cb;

    memset(&cb, 0, sizeof(aiocb));
    cb.aio_nbytes = SIZE_TO_READ;
    cb.aio_fildes = file;
    cb.aio_offset = 0;
    cb.aio_buf = buffer;

    // enqueue the read
    if (aio_read(&cb) == -1) { /* error handling */ }
```

[2] http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/

```
  do {
    // ... do something else ...
  while (aio_error(&cb) == EINPROGRESS); // poll

  // inspect the return value
  int numBytes = aio_return(&cb);
  if (numBytes == -1) { /* error handling */ }

  // clean up
  delete[] buffer;
  close(file);
```

Using Select/Poll. The idea is to improve performance by letting each thread handle multiple connections. When a thread is ready, it uses select/poll to find work:

- perhaps it needs to read from disk into a mmap'd tempfile;
- perhaps it needs to copy the tempfile to the network.

In either case, the thread does work and updates the request state.

Callback-Based Asynchronous I/O Model

Finally, we'll talk about a not-very-popular programming model for non-blocking I/O (at least for C programs; it's the only game in town for JavaScript and a contender for Go). Instead of select/poll, you pass a callback to the I/O routine, which is to be executed upon success or failure.

```
void new_connection_cb (int cfd)
{
  if (cfd < 0) {
    fprintf (stderr, "error␣in␣accepting␣connection!\n");
    exit (1);
  }

  ref<connection_state> c =
    new refcounted<connection_state>(cfd);

  // what to do in case of failure: clean up.
  c->killing_task = delaycb(10, 0, wrap(&clean_up, c));

  // link to the next task: got the input from the connection
  fdcb (cfd, selread, wrap (&read_http_cb, cfd, c, true,
                    wrap(&read_req_complete_cb)));
}
```

`node.js`: A Superficial View. We'll wrap up today by talking about the callback-based `node.js` model. `node.js` is another event-based nonblocking I/O model. Given that JavaScript doesn't have threads, the only way to write servers is using non-blocking I/O.

The canonical example from the `node.js` homepage:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello␣World\n');
}).listen(1337, '127.0.0.1');
console.log('Server␣running␣at␣http://127.0.0.1:1337/');
```

Note the use of the callback—it's called upon each connection.

However, usually we don't want to handle the fields in the request manually. We'd prefer a higher-level view. One option is expressjs[3], and here's an an example from the Internet[4]:

```
app.post('/nod', function(req, res) {
  loadAccount(req,function(account) {
    if(account && account.username) {
      var n = new Nod();
      n.username = account.username;
      n.text = req.body.nod;
      n.date = new Date();
      n.save(function(err){
        res.redirect('/');
      });
    }
  });
});
```

[3] `http://expressjs.com`

[4] `https://github.com/tglines/nodrr/blob/master/controllers/nod.js`