CMPSCI 377 Operating Systems **Fall 2009**

Lecture 11

Lecturer: Emery Berger *Scribe: Alexandre Passos*

11.1 Concurrency versus parallelism

A high-level distinction: concurrency and parallelism. MESA is about concurrency. The distinction is subtle: parallelism is about exploiting multiple processors, while concurrency is only about running different pieces of code at the same time.

Dealing with condition variables, for example, is easier with parallelism, as there is no need to solve the hard problem of dispatching.

Parallelism is about speedup, while concurrency is about the illusion of simultaneity or handling real-world concurrent events, essentially hiding latency.

Butler talks about the latency issue in the part where he handles IO devices, which could be represented as monitors but weren't to reduce latency and avoid getting stuck behind the slowest device in the system.

11.2 Shared memory versus messages

The distinction between threads and processes is whether or not you have a shared address space. It is about context: does it have a pointer to a shared memory block or its own page table? This is orthogonal to the concurrency vs parallelism distinction.

Butler talks about a long-standing dispute in the PL community on the distinction between message passing and shared memory. There is a duality between events and parallelism, and you can usually substitute one for the other. Threads and messages are equivalently powerful in a shared-memory machine.

The processes model encourages a design where you optimize access to shared state, known by DB people as shared-nothing, which scales. Having this one big chunk of memory shared by everybody creates management challenges and this costs a lot of time.

You can build a scalable shared-memory system, but the larger you get the closer you resemble a distributed architecture. Programming in distributed style can be distasteful, like medicine which is good for you but doesn't taste good.

Sockets in a single machine are effectively shared-memory, so you can scale software by first using local sockets and then distributing everything.

11.2.1 DSM

In the 90s there was this idea DSM - distributed shared memory - whose main idea was to to write shared memory programs (easier than passing messages) and then distributing it automatically. The abstraction of a single shared memory is generally seen as simple. Simple doesn't tend to scale up very well in a software

engineering sense. Think dynamic languages, which make it easy to write hello world but hard to write anything else.

DSM is about using a shared-memory model and running on a cluster. Alexandre asks about the similarity of this and COM and CORBA, and Emery explains COM and CORBA, which is just a way of distributing method calls and objects. C# has language-level annotation support for marshalling. This whole thing is RPC or RMI (remote procedure call or method invocation). Still bad in java but better in C#.

COM and CORBA only preserve the notion of object, but not the notion of a shared address space, so it's a limited view of DSM.

The challenge is not difficulty of implementation but cost model mismatch: when you write a shared-memory program the cost model is that memory access is free, constant-time. There is no way to preserve this abstraction in a distributed way, so you have to indirect everything and/or copy stuff all over the place. So this transforms all memory accesses into messages over the network. You can totally do this, but this is hugely expensive.

11.3 Cost models

The cost model mismatch means you live in one world but think in another one, and then why bother?

There is always the challenge when you program for performance where you'd like to abstract all details of the machine. You'd like to program in some high-level language and not care in which machine you're running. Issues like locality dominate at some point due to this annoying thing called physics, which means things take time to move around, so one has to be aware of memory hierarchy, including disk and network.

Going back to monitors.

11.4 Monitors and recursive locks

When you do threads in a shared-memory world you have the problem of concurrent updates to the same memory space. The solution is usually mutual exclusion. There is the issue of conditions, signaling something to happen, with semaphors.

Condition variables: no conditions, no variables in the actual thing.

Per-Brinck Hansen has some interesting comments, and we should check it out. He invented concurrent PASCAL.

The nice feature of monitors is that they elegantly combine mutual exclusion and "condition variables". MOnitors in java are part of the language. You can say that any method can be synchronized, and mutual exclusion happens automatically when you access it. Every object in hava secretly has a lock and a "condition variable" built-in. Whenever you invoke a synchronized method it acquires the object's lock, executes foo, and releases the lock afterward. There is a synchronized operator. These locks are recursive. Monitors are not recursive locks.

The problem with unrecursive locks is that if everything has a monitor associated with it, you can lock yourself out of the thing you want to do. This is almost impossible with object-oriented programming. You'd always deadlock, says Charlie, if you called a child who called a parent. Recursive locks fix this. Not making everything synchronized by default is the source of many many bugs and race conditions.

Emery says if you're driving a Chevette you really don't want to be slower still. Charlie says this is similar to the virtual keyword, which turns compile-time things into runtime errors if you forget to add it.

Steve Jobs made a mouse that has one button because you never know what the right thing to do is if you have two choices. Emery says that the existence of null in a language is a mistake noone knows how to fix. It's a problem with any imperative language, one way or the other.

Java was originally called Oak, and it was intended to be an embedded language. John says your toaster is now garbage collecting. You don't want people to have to debug your toaster. If devices have RAM you should be able to update things at runtime. The JVM was never intended to be a compiler. When the web came around security became a concern, and things became complex. The java security model makes it difficult to do bad things. Oak became java in a browser called the HotJava browser meant to run applets, safe code you could run in your browser, so people should use it.

You have to think back to the computers of 1995 and to the JVMs of 1995. The JVM would lumberingly haul itself up for literally a minute, while a little dancing dude on the screen made you angry, and finally your applet started running. The failure of java in the browser led directly to what became javascript, which started with a "let's build an interpreter inside the browser". Netscape was the first nice browser. The web was created, Emery remembers seeing the first web stuff, ftp with clickable links, hooray. From the University of Illinois' high-performance center came Mosaic, which was amazing, and could even display graphics! Then they added stuff after stuff, created netscape, which became huge and was then crushed by micrsoft.

Brendan Eich was then asked to make a language in two days. It was originally a lisp variant, which was then rebranded into JavaScript!

11.4.1 Thin locks

If only one thread ever acquires a lock you don't need to do anything. IBM came up with this idea idea, which could use a single bit to detect multiple threads accessing the same object. These were called Bacon bits, but became thin locks. You can just steal the low-order bit of a pointer to the object's lock structure.

The java memory model has a very well-defined model of what happens when you have unsynchronized access to certain types of variable called volatile. Volatile is crucial to the Java memory model and is necessary to most system programming in C/C++. It originated from memory-mapped I/O, as a way to prevent the C compiler from caching certain variables to registers or optimizes them away. Jokes about really quickly blinking lights.

Suppose you have concurrent programs communicating through memory, one thread setting a variable and another looping to check, volatile is necessary but not sufficient for things to work as expected. You also need to force the processors to buffer their writes. There are fences/memory barriers, instructions which flush the processor's write buffer. In C++0x you can do this in language. In Java, volatile does the right thing. Jokes about this being similar to keeping kosher.

According to the C++ standard any racy program is undefined. This is convenient when you're writing memory models, but as every concurrent program probably has races, this is quite meaningless.

Optimizing in concurrency is a huge motivation for memory models, as all optimizations fall apart in the presence of threads. Hans Boehm's paper that threads cannot be implemented as libraries is true because of this. This paper should be added to the list.

11.5 Global Interpreter Lock and Concurrency models

Twomore things to talk about.

Somebody mentioned the global interpreter lock. They try to do concurrency and not parallelism. The goal of parallelism up until recently was the domain of very few people, as nobody had multiple processes. Just now with multicore has this become a commonplace issue. Python and Ruby are cult of personality languages. There are PLs coming out of academia or industry, and languages coming out of ONE DUDE. There is an issue with kool-aid drinking.

These languages were designed to be scripting languages and they have an issue with concurrency. For implementation reasons it's very simple to use one big lock, and they're still struggling with this problem. Twitter had to abandon rails because it didn't scale due to this issue. [lost]

The other thing is the issue of concurrency models. There are vague models where anything can happen. Semaphores are wildly general. Monitors are less general, but still allow for all kinds of weird interactions. You get a kind of spaghetti code of concurrency.

As long as you have locks you have a risk of deadlocks, and there is not much you can do. Monitors won't help, and they can even make it worse.

You don't need a wild do-everything model of parallelism, which are great for parallelism but not so much for concurrency. Fully structured parallelism versus unstructured parallelism. In unstructured the kind of parallelism you get is very very dynamic, depending on races. You get a very complicated graph in terms of happens-before edges, as they are a dynamic property of the program's execution, and you should care because this makes analysis difficult by invoking non-determinism.

11.5.1 The issue of nondeterminism

Nondeterminism arises not only because of data races but because of locks. By eliminating some constructs you can still get usable model of concurrency, which is structured parallelism. Assume you only have fork and join. Even still you can have arbitrary wait-edges, in things like pthreads. However, if you can only wait for children you created you can prove all sorts of nice properties and implement this quite efficiently. All the happens-before edges are explicit, so if you ignore data races (a huge if) the execution is deterministic. To make this more deterministic you need to solve the issue of data races.

Dthreads eliminates all remaining vestiges of nondeterminism. All nondeterminism cannot be constrained, so dthreads deals with scheduler-based nondeterminism. You can use separate address spaces and a deterministic commit protocol to solve a lot of the nondeterminism.

Nondeterminism falls back into the concurrency vs parallelism issue. If you want to deal with IO, it's bad. It comes back to hiding latency versus increasing throughput. Unless you constrain event arisal the whole thing will be nondeterministic.

In distributed systems you can have a replicated state machine approach. This requires deterministic event ordering. By sequencing all events you control all external nondeterminism. If you use replicated state machines then dthreads are a very good thing, but for general systems it's harder.

Administrative notes

The following week there will be no classes and when we return there will be a midterm. The idea is: we're going to be faced with a few design problems, and we'll describe how would we design a system and what the main issues would be.