Threading

Concurrency Trends

Faster Computers

How is it that computers are faster now than 10 years ago?

a. Process improvements -- chips are smaller and run faster

b. Superscalar pipelining parallelism techniques -- doing more than one thing at a time from the one instruction stream.

Instruction Level Parallelism (ILP)

There is a limit to the amount of parallelism that can be extracted from a single instruction stream

The limit is around 3x or 4x

We are well in to the diminishing-returns region of ILP technology.

Hardware Trends

Moore's law: the density of transistors that we can fit per square mm seems to double about every 18 months -- due to figuring out how to make the transistors and other elements smaller and smaller.

Here are some hardware factoids to illustrate the increasing transistor budget.

The cost of a chip is related to its size in mm^2. It's a super-linear function -- doubling the size more than doubles the cost.

1989: 486 -- 1.0 um -- 1.2M transistors -- 79mm2

1995: Pentium MMX 0.35 um -- 5.5 M trans -- 128 mm2

1997: AMD athlon -- 0.25 um -- 22M trans -- 184mm2

2001: Pentium 4 -- 0.18um -- 42M trans -- 217 mm2

Q: what do we do with all these transistors?

A: more cache

A: more functional units (ILP)

A: multiple threads

1 Billion Transistors

How do you design a chip with 1 billion transistors?

What will you do with them all?

Extract more ILP? -- not really

More and bigger cache -- ok, but there are limits

Explicit concurrency -- YES

In 2002, Intel speculated that they could build a 1 billion transistor Itanium chip made of 4 Itanium cores and a huge shared cache.

Hardware vs. Software -- Hard Tradeoff

Writing single-thread software is much easier

Therefore, hardware thus far has largely been spent in extracting more ILP from the single thread.

That is, we put the burden on the hardware, and keep the software simple.

But we are hitting a limit there

For better performance, we can now move the problem to the programmers -- they must write explicitly parallel code. The code is much harder to write, but it can extract much more work from a given amount of hardware.

Hardware Concurrency Trends

1. Multiple CPU's -- cache coherency must make expensive off-chip trip
2. "Multiple cores" on one chip
   - They can share some on-chip cache
   - A goo d way to use up more transistors, without doing a whole new design.
3. Simultaneous Multi-threading (SMT)
   - One core with multiple sets of registers
   - The core shifts between one thread and another quickly -- say whenever there's an L1 miss.
   - Neat feature: hide the latency by overlapping a few active threads -- important if your chip is 10x faster than your memory system.
   - This is called "hyperthreading" by Intel
4. In 2005 you may have 2-4 cores, where each core is 2-4 way multithreaded (so the machine will appear to have 4-16 CPUs to the software).

Threads vs. Processes

Processes
- Heavyweight-- large start-up costs
- e.g. Unix process launched from the shell, piped to another process
- Separate addr space
- Cooperate with read / write streams (aka pipes)
- Synchronization is easy -- typically don't have shared address space

Threads
- Lightweight -- easy to create / destroy
- All in one addr space
- Can share memory (variables) directly
- May require more complex synchronization logic to make the shared memory work

Using Threads

Advantages to multiple threads...

1. Use Multiple Processors

Re-write the code to use concurrency -- so it can use n processors at once.
At present, this is still a little exotic.
Problem: writing concurrent code is hard, but Moore's law may force us this way as multiple CPU's are the inevitable way to use more transistors.

2. Network / Disk -- Hide The Latency

Use concurrency to efficiently block when data is not there -- can have hundreds of threads, waiting for their data to come in.
Even with one CPU, can get excellent results
The CPU is so much faster than the network, need to efficiently block the connections that are waiting, while doing useful work with the data that has arrived.
Writing good network code inevitably depends on an understanding of concurrency for this reason. This is no longer an exotic application.

3. Keep the GUI Responsive

Keep the GUI responsive by separating the "worker" thread from the GUI thread -- this helps an application feel fast and responsive.

Why Concurrency Is Hard

No language construct can make the problem go away (in contrast to mem management which is largely solved by GC). The programmer must be involved.
Counterintuitive -- concurrent bugs are hard to spot in the source code. It is difficult to absorb the proper "concurrent" mindset.
There is no fixed programmer recipe that will just make the problem go away.
Hard for classes to pass the "clueless client" test -- the client may really need to understand the internal lock model of a class to use it correctly.
Concurrency bugs are very, very latent. The easiest bugs are the ones that happen every time.
In contrast, concurrency bugs show up rarely, they are very machine, VM, and current machine loading dependent, and as a result they are hard to repeat.
"Concurrency bugs -- the memory bugs of the 21st century."
Rule of thumb: if you see something bizarre happen, don't just pretend it didn't happen. Note the current state as best you can.

Java Threads

Current Running Thread

A thread of execution -- executing statements, sending messages
Has its own stack, separate from other threads
Also known as a "thread of control" to distinguish from a java Thread object.
What we think of as "running" in C...
When have a sequence of statements

```
int i =7;
while (i<10) {
  foo.a();
  ...
}
```

What we think of as "excute" or "run" -- there is a thread of control which is executing the statements -- the "current running thread".
A message send, in essence, sends the current running thread over to execute against the receiver.

static void main(String[] args)

A Java program begins with a thread executing main(), and that one thread executes the whole program.
We will see how to create and run other threads which will run concurrently.

Threads -- Virtual Machine

Threads in Java are a little easier to deal with than other languages -- there is thread support built in to the language at a low level. Other languages have threads bolted-on to an existing structure.
The VM keeps track of all the threads and schedules them to get CPU time.
The scheduling may be preemptive (modern) or cooperative (old, but easier to implement)

Thread Class

A Thread object is just a regular Java object -- it has an address, responds to messages, etc.
A Thread object is a token which represents a thread of control in the VM
We send messages to the Thread object -- the VM interprets these and does the appropriate operations on the underlying threads in the VM

Thread Class Use

1. Subclass off Thread and implement the run() method
2. Create an instance of your Thread subclass. It is not running yet, so you can set things up
3. Send the thread object the start() message -- at this point the VM can allocate a real thread of control, and schedule it to execute the Thread object's run() method

4. A thread of control begins executing the run() method of the Thread object
5. Eventually, the thread of control finishes / exits the run() method

Thread.currentThread()

A static utility method in the Thread class
Returns a pointer to the Thread object that represents the currently running thread of control...

```
int i = 6;
int sum = 7 + 12;    // regular computation

Thread me = Thread.currentThread();
// "me" is the Thread object that represents our thread of
// control (the thread that computed the sum above)
```

Joining

A thread wishes to wait until another thread completes its run()
Send the t.join() message -- causes the current running thread to block efficiently until t finishes its run
Must catch the InterruptedException

```
// start a thread
Thread t = new ...
t.start();

// at this point, two threads may be running -- me and t

// wait for t to complete its run
try {
   t.join();
}
catch (InterruptedException ignored) {}

// now t is done (or we were interrupted)
```

Simple Thread Example

Strategy: Subclass Thread, define the run() method

```
/*
 Demonstrates creating a couple worker threads, running them,
 and waiting for them to finish.

 Threads respond to a getName() method, which returns a string
 like "Thread-1" which is handy for debugging.
*/
public class Worker1 extends Thread {
   public void run() {
      long sum = 0;
      for (int i=0; i<100000; i++) {
         sum = sum + i; // do some work
```

```
            // every n iterations, print an update
            // (a bitwise & would be faster -- mod is slow)
            if (i%10000 == 0) {
               System.out.println(getName() + " " + i);
            }
         }
      }


      public static void main(String[] args) {
         Worker1 a = new Worker1();
         Worker1 b = new Worker1();

         System.out.println("Starting...");
         a.start();
         b.start();

         // The current running thread (executing main()) blocks
         // until both workers have finished
         try {
            a.join();
            b.join();
         }
         catch (Exception ignored) {}

         System.out.println("All done");
      }
       /*
            Starting...
            Thread-0 0
            Thread-1 0
            Thread-0 10000
            Thread-0 20000
            Thread-1 10000
            Thread-0 30000
            Thread-1 20000
            Thread-0 40000
            Thread-1 30000
            Thread-0 50000
            Thread-1 40000
            Thread-0 60000
            Thread-1 50000
            Thread-0 70000
            Thread-1 60000
            Thread-0 80000
            Thread-0 90000
            Thread-1 70000
            Thread-1 80000
            Thread-1 90000
            All done
       */

}
```