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) -- limit of 3-4x

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

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

Concurrency Support

Chip

The chip(s) can support multiple threads -- especially cache coherency

Software

The software must be coded to use multiple threads -- this is a significant cost, but we're getting better at it.

CPU 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. Chip Multi-threading
    - 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" in the next gen Pentium 4

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

1. Use Multiple Processors

Re-write the code to use concurrency -- so it can use n processors at once
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
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.

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.

Threads

Thread of Control

A thread of execution -- executing statements, sending messages

Has its own stack, separate from other threads

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 pre-emptive (modern) or co-operative (old, but easier to implement)

Thread Class

A Java object of the Thread class that represents a thread of control

Thread 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 -- now it can get scheduled to run

4. A thread of control begins executing the run() method of the new thread
5. Eventually, the thread of control finishes / exits run(), and the thread of control is done.

Joining

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

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

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

// now t is done (or an interrupt woke us  up)
```

Simple Thread Example

Strategy: Subclass Thread, define the run() method

```
/*
 Demonstrates creating a couple worker threads, running them,
 and waiting for them to finish.
*/
class Thread1 {
   // Subclass off Thread and override run()
   static class Worker extends Thread {
      public void run() {
         long sum = 0;
         for (int i=0; i<50000; i++) {
            sum = sum + i; // do some work

            // every n iterators, print an update
            if (i%10000 == 0) {
               System.out.println("Working:" + i);

               // not strictly necessary, but helps the VM switch among the threads
               Thread.yield();
            }
         }
      }
   }

   public static void demo() {
      Worker a = new Worker();
      Worker b = new Worker();
```

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

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

      System.out.println("All done");
   }
   /*
      Starting...
      Working:0
      Working:10000
      Working:20000
      Working:0
      Working:30000
      Working:40000
      Working:10000
      Working:20000
      Working:30000
      Working:40000
      All done
   */

}
```

The Classic Threading Problem

Mutual exclusion

Keeping the threads from interfering with each other. Worry about memory shared by multiple threads.

Cooperation -- join / wait / notify

Get threads to cooperate. Typically this centers on handing information from one thread to another, or signaling one thread that another thread has finished doing something.

Race Condition / Critical Section

A section of code, that causes problems if two or more threads are executing it at the same time

Typically the problem revolves around some shared memory that both threads are using at the same time

Establish "mutual exclusion" -- only one thread is in the critical section at a time

Race Condition Example

```
class Foo {
   private int a, b;

   public Foo() {
```

```
        a = 0;
        b = 0;
    }

    // reader (should always return an even number)
    public int sum() {
        return(a+b);
    }

    // writer
    public void inc() {
        a++;
        b++;
    }
}
```

Reader / Writer conflict

e.g. inc() and sum() at the same time
Reader thread can get the sum() while inc() is midway through executing

Writer / Writer conflict

e.g. Thread A runs inc() at the same time thread B runs inc()
The two inc()'s can interleave to mess up the receiver state
(a++ is not **atomic,** it can interleave with another a++ -- this is true in most languages)

Random Interleave

Race conditions depend on two or more threads "interleaving" their execution in just the right way to exhibit the bug. It happens rarely and randomly, but it happens.
The likelihood of the interleave is seemingly random -- depending on the system load and the number of processors.
This is why locating concurrency bugs is so hard -- exhibit themselves sporadically

Object Lock + Synchronized Method

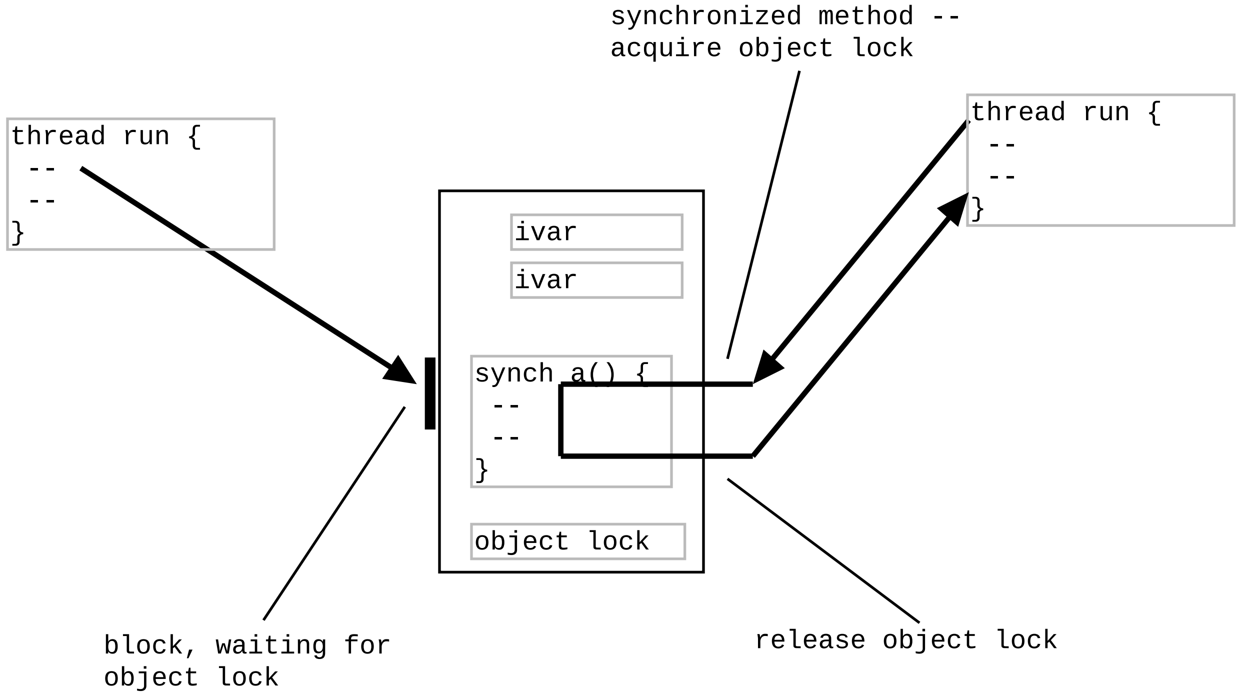
Every object has a "lock"
A "synchronized" method must acquire the lock of the receiver before executing
The lock is released when the method exits
If the lock is already held by another thread, we wait (efficiently) for the other thread to exit and so release the lock
Two threads cannot hold the receiver lock at the same time
  the second "blocks" until the first leaves -- they take turns

Receiver Lock

The lock is in the receiver object -- it provides mutual exclusion for multiple threads sending messages to **that** receiver. Other objects have their own locks.
If a method is not synchronized, it will ignore the lock and just go ahead.

Therefore if it makes sense for one of the getters / setters to be synchronized, probably they all should be synchronized

Synchronized Method Picture



Synchronized Method Example

```
/*
 Simple example of an object that uses synchronization
 keyword so multiple threads may send it messages.
 The sum() and incr() methods are "critical sections"
 -- they cannot be run simultaneously by multiple threads.
 The "a++" and "a+b" operations are not "atomic".
 Yhey do not produce correct results when run by
 multiple threads
 The sum() and inc() methods are declared "synchronized" --
 for a particular foo object, only one thread may execute
 sum() or inc() at a time.
*/
class Foo {
   private int a, b;

   public Foo() {
      a = 0;
      b = 0;
   }

   // reader (should always return an even number)
   public synchronized int sum() {
      return(a+b);
   }
```

```
   // writer
   public synchronized void inc() {
      a++;
      b++;
   }
}
/*
 A simple worker subclass of Thread.
 In its run(), sends 1000 inc() messages
 to its Foo object.
*/
class FooWorker extends Thread {
   private Foo foo;

   // ctor takes the foo we use
   public FooWorker(Foo foo) {
      this.foo = foo;
   }
   public void run() {
      for (int i=0; i<1000; i++) {
         foo.inc();
      }
   }

   /*
    Create a foo and 3 workers.
    Start the 3 workers -- they do their run() --
    and wait for the workers to finish.
   */
   public static void main(String args[]) {
      Foo foo = new Foo();
      FooWorker w1 = new FooWorker(foo);
      FooWorker w2 = new FooWorker(foo);
      FooWorker w3 = new FooWorker(foo);

      w1.start();
      w2.start();
      w3.start();
      // the 3 workers are running
      // all sending messages to the same object

      // we block until the workers complete
      try {
         w1.join();
         w2.join();
         w3.join();
      }
      catch (InterruptedException ignored) {}

      System.out.println(foo.sum());   // this will be 6000
      /*
       If inc() were not synchronized, the result would
       probably be something like 5992 because of the
       writer/writer conflicts of multiple threads trying
       to execute inf() on an object at the same time.
      */
   }
}
```