Hardware

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
2000: 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

Recall

From previous handout
Lock in the receiver
- every method must synch
- instances vs. static data
- instances with one shared object (similar to above)
- static methods vs. instance
- too fine grain

Get In and Get Out

It's generally regarded as good style to hold the lock as little as possible
1. acquire the lock
2. do the critical operation quickly
3. release the lock

Like a Database -- leave in good state

DB's have an idea of a "transaction" -- a change that happens in full or is "rolled back" to not have happened at all.

Leave in good state
Think of your messages that way.. a method gets the lock, makes all its changes (with sole possession of the lock), releases the lock leaving the object fully in the new state. Don’t release the lock when the object is partially in the new state.
Don't worry about order
If the above is true, you don't have to worry about the order the methods happened to acquire the lock.

One Big Lock
This example puts one lock over the whole thing

```java
public synchronized void foo() {
    int newValue = read1();
    newValue += read2();
    newValue = dict.lookup(newValue);
    String result = new String("foo: " + newValue);
    length++;
    array[length] = result;
}
```

Fine Locks
This example uses two locks to have smaller critical sections + some sections are done without any lock

```java
public void foo() {
    int value;
    synchronized(this) {
        value = read1();
        value += read2();
    }
    value = dict.lookup(newValue);
    String result = new String("foo: " + newValue);
    synchronized(array) {
        length++;
        array[length] = result;
    }
}
```

read1/read2 lock
Suppose that it's important that read1 and read2 reflect one state of the receiver. We obtain the receiver lock for the duration (depending on setters being synchronized).

read vs. lookup
Is it important that read() and lookup() happen without any state changes in between? They are not under one lock with the fine grain design.

read1/read2 replacement
Maybe a better design would be that there's a one method replacement that effectively does read1/read2 in one operation.

OOP design: methods should meet the needs of the caller
dict.lookup()
   The fine grain version assumes that lookup() is itself thread safe (or we
could synchronize it ourselves)
new String() + I/O
   new is very expensive -- try not to be holding a lock when you do it (or
I/O)
array lock
   We'll use the convention that the array-changing code gets the array
lock first.
array method
   Better would be to have a dedicated method, so the convention is more
explicit...
   public void addElt(String string) {
      synchronized(array) {
         length++;
         array[length] = string;
      }
   }

Fine locks Pro: More concurrency
   Having finer grain locks, allows more threads to be "in flight" at one
time.

Fine locks Con: More cost
   Acquiring each lock has a little cost. More locks -> more cost
   Especially painful in the common case where we didn't have multiple
   threads anyway -- we're still paying the cost.

Fine locks Con: More complex
   More locks to manage -- the "one big lock" model is conceptually
   simple

Fine locks Con: Deadlock

Design: Client vs. Implementation
   Client synch
      The operation, lookup or whatever, is not internally synchronized. The
client includes synchronization in their code to avoid calling the
method in an unsafe way.
   Implementation synch
      The method is either synchronized or includes internal
      synchronization so that it can be called by multiple threads.
The client can be unaware of the threading issues, and just works since
the implementation takes care of it.
Comparison
+ From a design point of view, doing the synch in the implementation is better. Reducing what the client needs to know is a better OOP design.
- Optimization -- it’s possible that the synchronization in the implementation then does not allow an intelligent and aggressive client from making certain optimizations -- e.g. using no locks in certain cases.

== Classic Deadlock Rules

Deadlock
Have locks x and y
One thread acquires x then y
Another thread acquires y then x

The Unhappy Caller
Some code you are calling (and didn’t write) may depend on some internal lock, and so create the (x, y) situation without your knowing

One Deep Ok
If the code you call does its own thing and returns (no call backs to you) then deadlock cannot occur -- Yay!
EG Vector.addElementAt() can never cause deadlock -- example of Get In Get Out rule

#1: One Big Lock
If there's just one big lock, you won't have deadlocks.
Further, it’s ok if you call things like new that have their own lock, but never come back and do something that depends on the one big lock

#2: Order The Locks
Establish by design, a fixed order for the locks
Everyone must acquire the locks in that order
If all the code follows the order, you can't get deadlock.

Conclusion
Most likely to have problems when mixing separate code modules, each with some lock logic in it, and each calls the other.
There is no simple recipe to avoid the problem, it just requires overall understanding.
Simple strategy: have the "one big lock" for correctness, and revisit the decisions if concurrent efficiency is a real problem.
--Java Thread Syntax

Thread Class

```java
class MyThread extends Thread {
    public void run() {
        // do whatever
    }
}
```

Subclass off Thread
Implement run()
Plan to fall through bottom
    Fall through the bottom of run() normally when done
1. Normally
2. Exception
    An uncaught exception will come back out to the run()
Debug: catch/print exceptions
    For debugging, you may want to Catch/print exceptions in your run()
        so your thread doesn't die off silently when it gets an error.
No re-use
    Once a Thread is done with its run() it cannot be used again.

Runnable Interface

```java
class MyClass extends Whatever implements Runnable {
    public void run() {
        ...
    }
    ...
}
```

Implement Runnable (an Interface)
Implement run() method -- same as Thread
Pass Runnable obj to Thread ctor

`start(), not run()`

```java
thread.start()
    At this point the thread may be scheduled for time.
Set up first
    You can avoid some concurrency problems by getting everthing all set up, and then calling start
never call thread.run()
```
Thread vs. System Thread

We'll say that a "system thread" represents the real underlying access to CPU -- something that's actually running.
A Thread object is Java object in memory that represents a system thread

Static Methods -- Tricky

Current running thread
These operate on the currently running system thread that is running the lines of code

Not Thread
Not necessarily the Thread object that happens to be the receiver

static Thread Thread.currentThread()
A Thread object representing the current running thread

static void Thread.sleep()
Force the current running thread to sleep

static void Thread.yield()
Force the current running thread to yield (allow other threads to get some time)

Tragic Syntax #1

Thread t1 = new ThreadSubclass();
t1.start();
t1.yield(); // this does not yield t1, it yields us
Tragic Syntax #2

The Thread object is still just a plain old object that messages can run against.

Some messages work on the current running thread, not the receiver Thread object -- yield() and sleep() (these are static in Thread)

class Foo extends Thread {
    int value;

    public void run() {
        for (int i = 0; i<1000; i++) {
            bar();
        }
    }

    public synchronized bar() {
        i = i +1;
        Thread.yield(); // Works on the caller thread
        this.yield(); // NOT the receiver Foo object
        // (Thread.currentThread() == this) is FALSE for the bar() call below
    }
}

test {
    Foo foo = new Foo();
    foo.start();

    foo.bar(); // This causes a yield() on our thread, not the Foo object
}

Swing Threading

There's a special Swing thread (we'll think of it as one thread, although it could be several threads cooperating with locks)
Dequeues real time user events
Translates to paint() and action notifications
Once a swing component is subject to pack() or setVisible, no other thread should send it Swing sensitive messages such as add(), setPreferredSize(), getText() ...

The Giant Swing Mutex

Like a giant mutex over all the Swing state -- only one thread (the Swing thread) is allowed to touch Swing state
EG how could paint(), and pack() work if values were simultaneously changing?
They are essentially using the One Big Lock strategy on all of Swing
I've come to decide that this is actually the best available way to design Swing
1. Swing Safe
   There are few special methods are valid to call against Swing, even if you are not the Swing thread...
   repaint(), revalidate(), addXXXListener(), removeXXListener()

2. On the Swing Thread Anyway
   As long as you are just responding in, say, actionPerformed(), you are in the Swing thread, so do whatever you want. All notifications are done on the swing thread, so if you are responding to a notification, you are ok.
   Just don’t do something that blocks or takes a long time – for something costly, create a separate thread and have report back (see below) when it has something.

3. Invoke The Swing Thread
   If you are not in the Swing thread, get the Swing thread to do something for you...
   Returns immediately : SwingUtilities.invokeLater(new Runnable() {
       public void run() { ... } }); -- runs the given code on the swing thread when it becomes available.
   Blocks: SwingUtilities.invokeLater(new Runnable() { ... })