Advances in Programming Languages

APL19: Cautionary Tales in Concurrency

David Aspinall
(slides by Ian Stark)

School of Informatics
The University of Edinburgh

Monday 16 March 2009
Semester 2 Week 10

Programming-Language Techniques for Concurrency

This is an addendum to the three lectures on programming language techniques for concurrency.

We looked at:

- Basic Java concurrency
- Concurrency abstractions
- Concurrency in some other languages (Scala, Polyphonic C#)

And we explained that even with good abstractions and language support, concurrent programming is difficult and things can go wrong.

This lecture describes three of the many thorny areas:

- Complex lock manipulations
- Scheduling priorities
- Weak memory models

Outline

Outline

Concurrency in Java and C#

Java

. . . has language facilities for spawning a new java.lang.Thread, with **synchronized** code blocks using per-object locks to protect shared data, and signalling between threads with wait and notify.

C#

. . . has language facilities for spawning a new System.Threading.Thread, to lock code blocks using per-object locks to protect shared data, and signalling between threads with Wait and Pulse methods.

Reentrant Locks

One **synchronized** method in a class may call another, so that it already has the lock it needs. Java requires *reentrant* locks.

Multiple locking

```
public class C {

  public synchronized void actionA(int n) {
     actionB(n); actionC(n);
  }

  private synchronized void actionB(int n) { ... }
  private synchronized void actionC(int n) { ... }
}
```

Reentrant Locks

When to release a reentrant lock cannot be determined statically; there must be a runtime count of how many times it is acquired and released.

Recursive locking

```
public class C {

  public synchronized void actionA(int n) {
    if (n>0) { actionB(n); actionC(n); actionA(n−1); }
  }

  private synchronized void actionB(int n) { ... }
  private synchronized void actionC(int n) { ... }
}
```

Locks within Locks

Sometimes an action will need to acquire exclusive use of two separate resources. This requires nested **synchronized** blocks.

Two locks

```
// Method to swap elements between two arrays
public void exchange(int[] p, int i, int[] q, int j) {

  synchronized(p){                                  // Claim first array p[]
    synchronized(q){                                // Claim second array q[]
      int v = p[i]; p[i] = q[j]; q[j] = v;          // Exchange elements
    }
  }
}
```

Locks within Locks

This is *thread-safe*, but cannot ensure *liveness*. If two concurrent instances of exchange try to obtain the same locks in opposite orders, they deadlock.

Two locks blocked

```
// Method to swap elements between two arrays
public void exchange(int[] p, int i, int[] q, int j) {

  synchronized(p){                                  // Claim first array p[]
    synchronized(q){                                // Claim second array q[]
      int v = p[i]; p[i] = q[j]; q[j] = v;          // Exchange elements
    }
  }
}
```

```
exchange(a,1,b,4);   |   exchange(b,5,a,2);
```

Networks of Locks

The problem arises in any situation where a routine needs exclusive access to multiple resources before it can proceed. Several instances of the routine can run concurrently without problem: but at any time two may deadlock because each holds a lock needed by the other.

This is known as the *Dining Philosophers' Problem*.

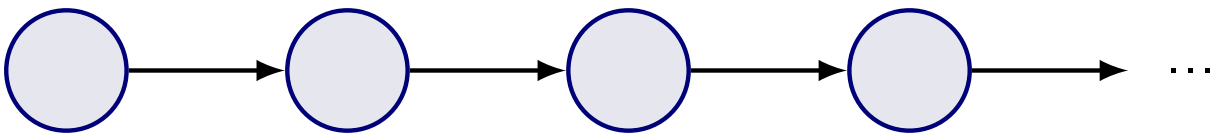
There are various solutions, including:

- Find some global ordering on the all resources, and then acquire the required resources according to that order.
- Add a centralized guard lock that must be held before attempting to reserve any resources.

These work, but they don't come for free: at the least, they have to be properly chosen and correctly implemented for the situation at hand.

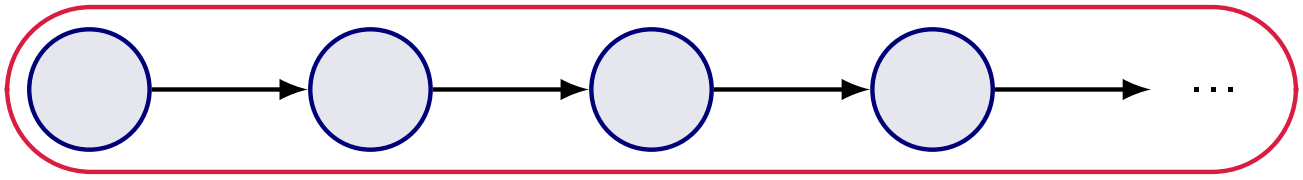
Non-Nesting Locks

If we want to modify a single node in a shared linked list, we might lock the whole thing:



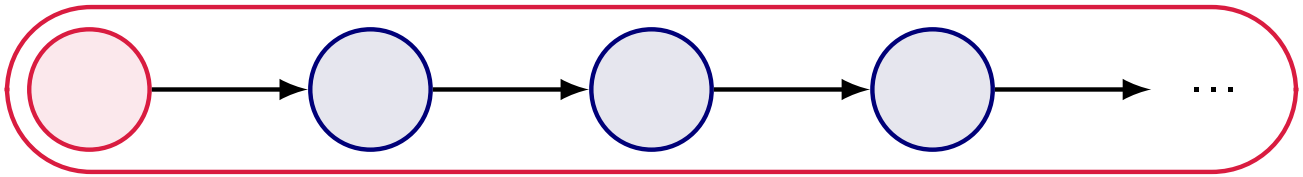
Non-Nesting Locks

If we want to modify a single node in a shared linked list, we might lock the whole thing:



Non-Nesting Locks

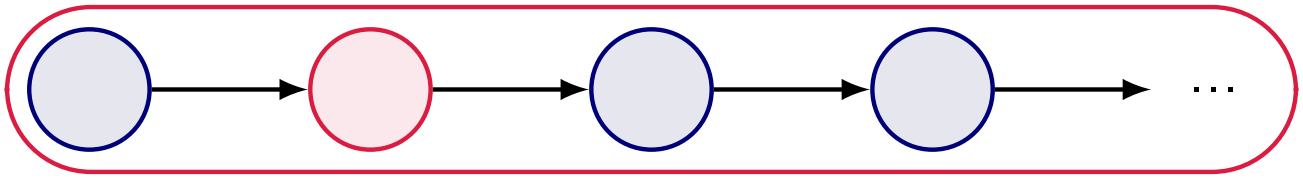
If we want to modify a single node in a shared linked list, we might lock the whole thing:



find the node we want,

Non-Nesting Locks

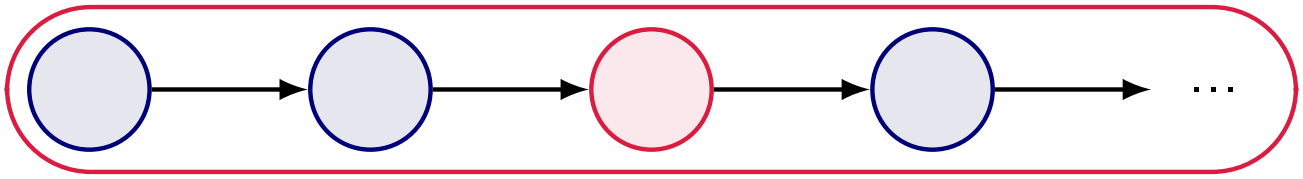
If we want to modify a single node in a shared linked list, we might lock the whole thing:



find the node we want,

Non-Nesting Locks

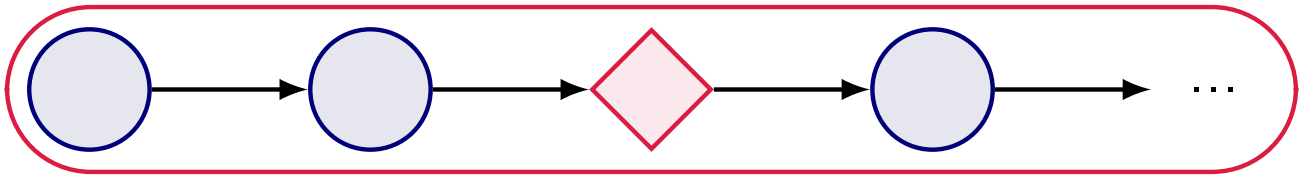
If we want to modify a single node in a shared linked list, we might lock the whole thing:



find the node we want,

Non-Nesting Locks

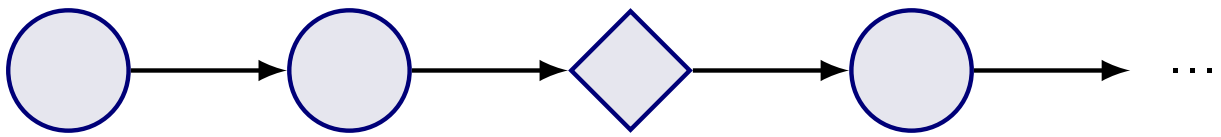
If we want to modify a single node in a shared linked list, we might lock the whole thing:



find the node we want, change it,

Non-Nesting Locks

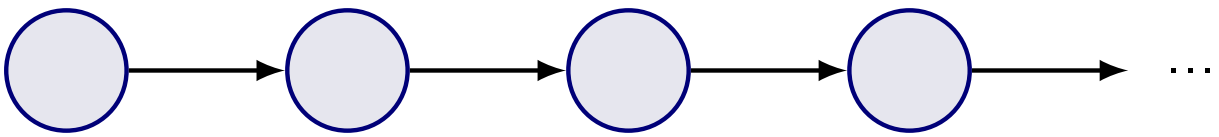
If we want to modify a single node in a shared linked list, we might lock the whole thing:



find the node we want, change it, and release the lock.

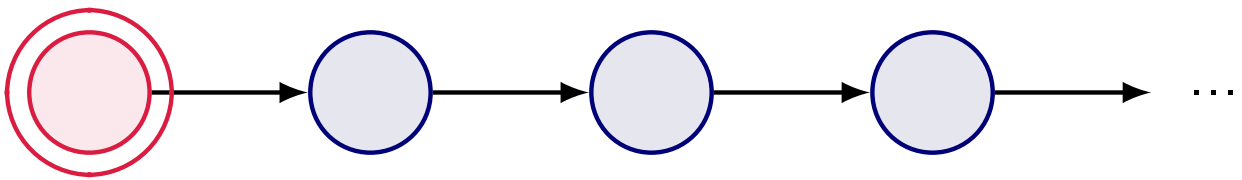
Non-Nesting Locks

Instead of locking the entire list, we could lock the nodes individually as we pass them:



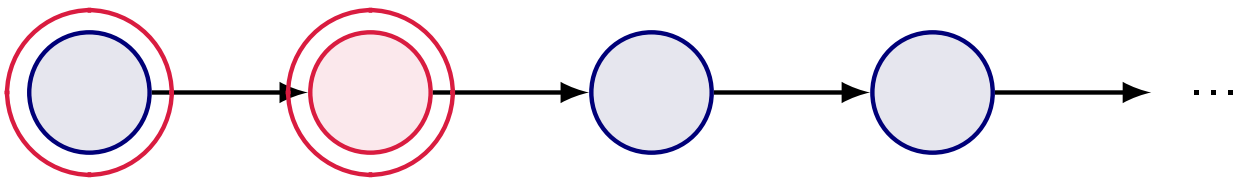
Non-Nesting Locks

Instead of locking the entire list, we could lock the nodes individually as we pass them:



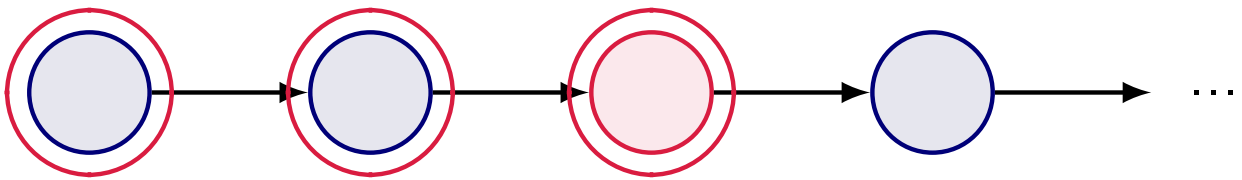
Non-Nesting Locks

Instead of locking the entire list, we could lock the nodes individually as we pass them:



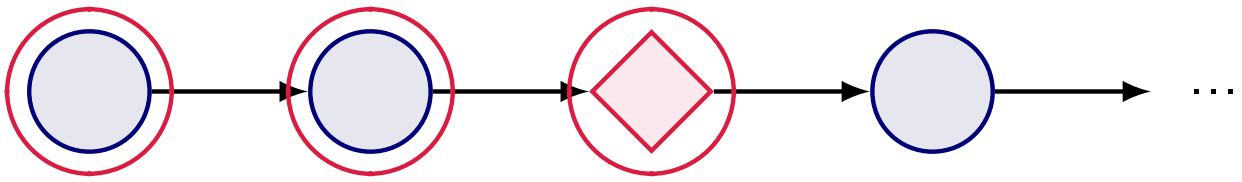
Non-Nesting Locks

Instead of locking the entire list, we could lock the nodes individually as we pass them:



Non-Nesting Locks

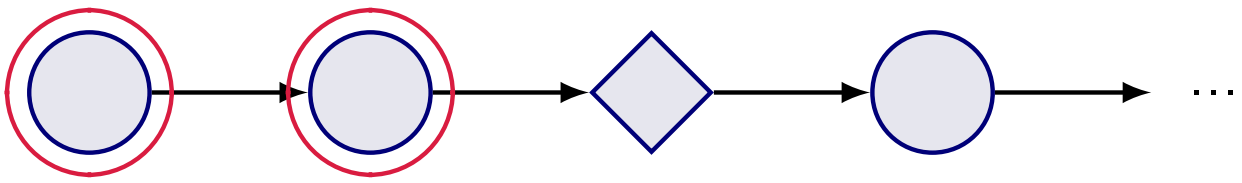
Instead of locking the entire list, we could lock the nodes individually as we pass them:



change the node,

Non-Nesting Locks

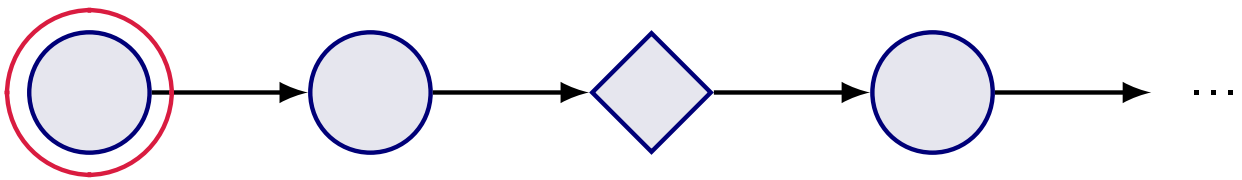
Instead of locking the entire list, we could lock the nodes individually as we pass them:



change the node, and unlock them on the way out.

Non-Nesting Locks

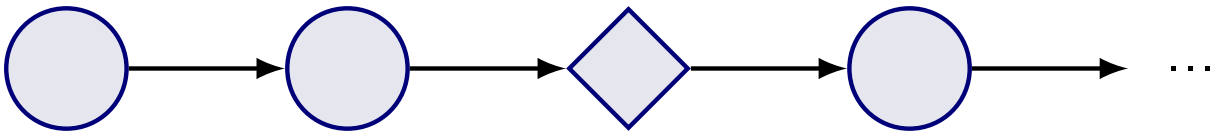
Instead of locking the entire list, we could lock the nodes individually as we pass them:



change the node, and unlock them on the way out.

Non-Nesting Locks

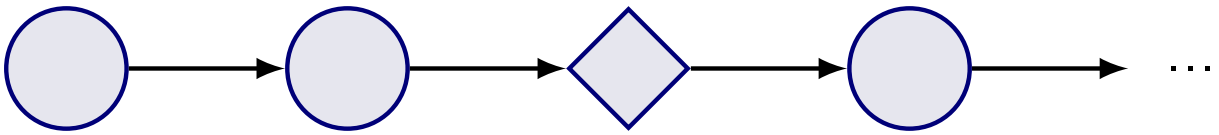
Instead of locking the entire list, we could lock the nodes individually as we pass them:



change the node, and unlock them on the way out.

Non-Nesting Locks

Instead of locking the entire list, we could lock the nodes individually as we pass them:

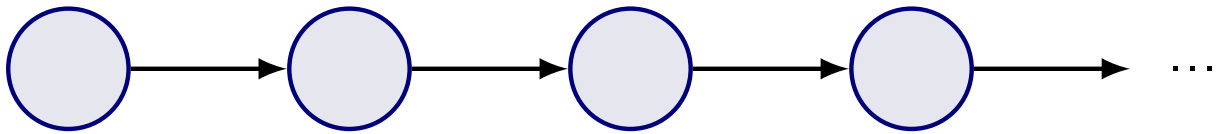


change the node, and unlock them on the way out.

Unfortunately, that still blocks the whole list from access by any other thread.

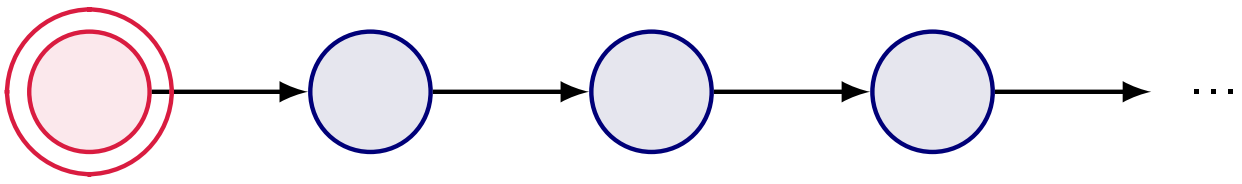
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



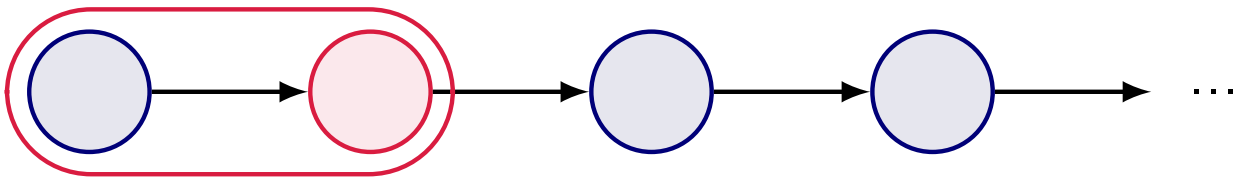
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



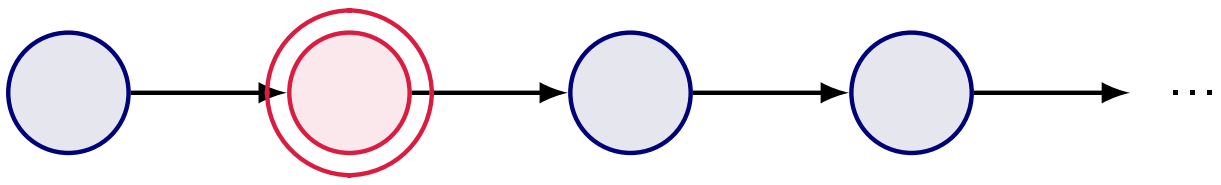
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



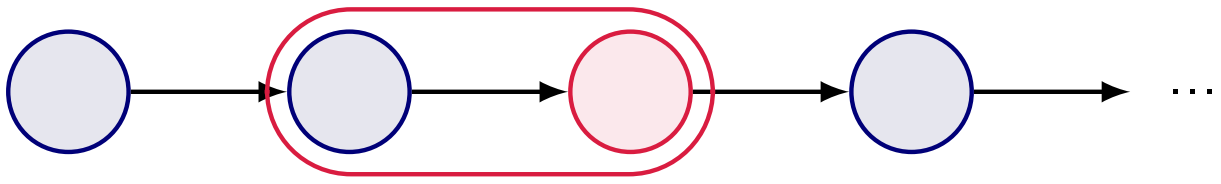
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



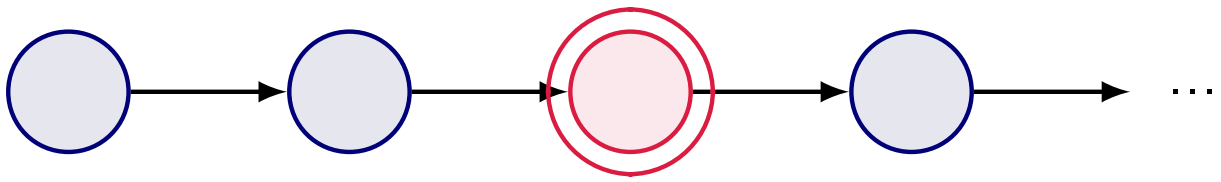
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



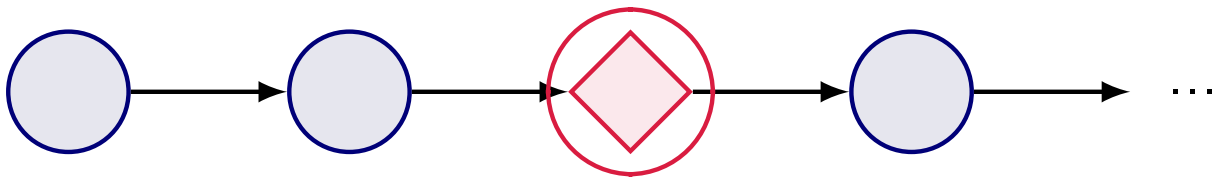
Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



Non-Nesting Locks

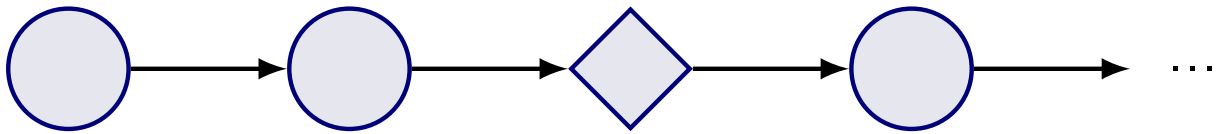
To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



change the node,

Non-Nesting Locks

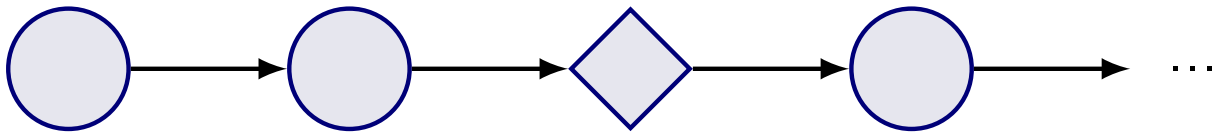
To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



change the node, and release the lock.

Non-Nesting Locks

To let others use the list, while avoiding any interference, we can instead lock nodes two at a time:



change the node, and release the lock.

This is *hand-over-hand* or *chain* locking.

Non-Nesting Locks

Hand-over-hand locking is a useful concurrency idiom:

- ✓ It allows multiple threads to modify a list concurrently, without overtaking or interfering.
- ✓ It works for trees and more complex datastructures.
- ✗ It requires locks whose acquire/release is not nested.
- ✗ In particular, the built-in locks of Java/C# are not enough...
- ✗ ... so we need to use the extended concurrency libraries.

As before, this works, but it doesn't come for free: it has to be properly specified and correctly implemented for the situation at hand.

Outline

Thread Priorities

Where multiple threads are competing for time-slices, it becomes important to control the behaviour of the scheduler that decides which runs when.

One way to do this is to assign each thread a *priority*, and use that to inform scheduler decisions. Typically, higher priority threads may pre-empt lower priority threads, but not vice versa; and if both are waiting to run, the higher priority thread will go first.

Java and C# both attach scheduler priorities to threads, as do almost all concurrent systems. Exactly how the priority affects scheduling decisions will differ from one system to another.

Getting Priorities Wrong

Consider the following concurrent system:

- A shared resource: say, a data bus with an access lock
- A low priority thread: collect some boring data, briefly write to bus.
- A medium priority thread: do an interesting thing for a while.
- A high priority thread: manage the bus; intermittent but vital.

Getting Priorities Wrong

Consider the following concurrent system:

- A shared resource: say, a data bus with an access lock
- A low priority thread: collect some boring data, briefly write to bus.
- A medium priority thread: do an interesting thing for a while.
- A high priority thread: manage the bus; intermittent but vital.

Consider the following timeline of activities:

- Low thread seizes the lock, prepares to use the bus.

Getting Priorities Wrong

Consider the following concurrent system:

- A shared resource: say, a data bus with an access lock
- A low priority thread: collect some boring data, briefly write to bus.
- A medium priority thread: do an interesting thing for a while.
- A high priority thread: manage the bus; intermittent but vital.

Consider the following timeline of activities:

- Low thread seizes the lock, prepares to use the bus.
- High thread pre-empts low, runs, cannot get lock, sleeps.

Getting Priorities Wrong

Consider the following concurrent system:

- A shared resource: say, a data bus with an access lock
- A low priority thread: collect some boring data, briefly write to bus.
- A medium priority thread: do an interesting thing for a while.
- A high priority thread: manage the bus; intermittent but vital.

Consider the following timeline of activities:

- Low thread seizes the lock, prepares to use the bus.
- High thread pre-empts low, runs, cannot get lock, sleeps.
- Medium thread runs, does not need lock, continues to run.

Getting Priorities Wrong

Consider the following concurrent system:

- A shared resource: say, a data bus with an access lock
- A low priority thread: collect some boring data, briefly write to bus.
- A medium priority thread: do an interesting thing for a while.
- A high priority thread: manage the bus; intermittent but vital.

Consider the following timeline of activities:

- Low thread seizes the lock, prepares to use the bus.
- High thread pre-empts low, runs, cannot get lock, sleeps.
- Medium thread runs, does not need lock, continues to run.

The low thread never gets a chance to run, so does not release the lock, and the high thread is blocked indefinitely by lower-rated tasks.

Priority Inversion

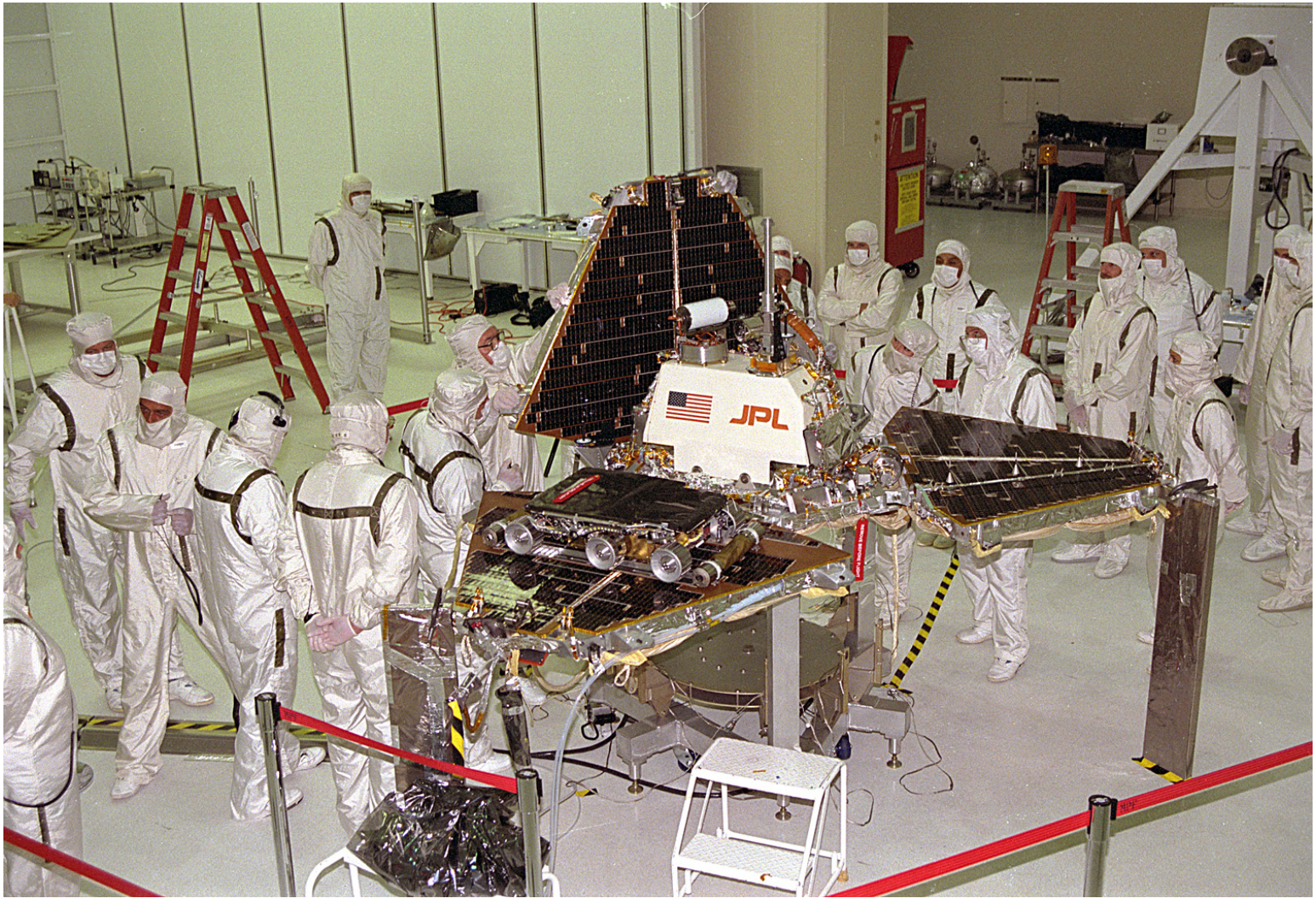
This situation is known as *priority inversion*, where a low-priority thread, or combination of threads, manages to block a higher-priority activity.

There is a standard solution: *priority inheritance*, in which a thread holding a lock is temporarily promoted to the priority level of any thread waiting on that lock.

However, this adds complexity, has runtime cost, and makes understanding scheduling yet more complex. Java does not include priority inheritance, .NET might, and the .NET compact framework (embedded systems) certainly does.

But honestly, that all sounds rather contrived and unlikely, doesn't it? The sort of thing that requires planets to align first — does it ever happen?

Mars Pathfinder



Mars Pathfinder Lander preparations, February 1996

Mars Pathfinder

Mars Pathfinder Lander viewed from the rover Sojourner, November 2003

Mars Pathfinder

Pathfinder has a memory bus architecture joining the various system components. Code running on the system included:

- A low-priority weather observation process that occasionally posted data to the bus.
- Medium-priority long-running communications tasks that didn't use the bus.
- A high-priority process to regularly check that all is well on the bus.

Safety feature: If the high-priority process was prevented from running for a long time, then the system stopped all activities, reset, and shut down for the rest of the day.

Mars Pathfinder

Pathfinder has a memory bus architecture joining the various system components. Code running on the system included:

- A low-priority weather observation process that occasionally posted data to the bus.
- Medium-priority long-running communications tasks that didn't use the bus.
- A high-priority process to regularly check that all is well on the bus.

Safety feature: If the high-priority process was prevented from running for a long time, then the system stopped all activities, reset, and shut down for the rest of the day. A "long time" means more than 1/8s.

Mars Pathfinder

Pathfinder has a memory bus architecture joining the various system components. Code running on the system included:

- A low-priority weather observation process that occasionally posted data to the bus.
- Medium-priority long-running communications tasks that didn't use the bus.
- A high-priority process to regularly check that all is well on the bus.

Safety feature: If the high-priority process was prevented from running for a long time, then the system stopped all activities, reset, and shut down for the rest of the day. A "long time" means more than 1/8s.

On Mars, the mission was so successful that data levels rose far above those anticipated. And the lander reset.

Mars Pathfinder

Pathfinder has a memory bus architecture joining the various system components. Code running on the system included:

- A low-priority weather observation process that occasionally posted data to the bus.
- Medium-priority long-running communications tasks that didn't use the bus.
- A high-priority process to regularly check that all is well on the bus.

Safety feature: If the high-priority process was prevented from running for a long time, then the system stopped all activities, reset, and shut down for the rest of the day. A "long time" means more than 1/8s.

On Mars, the mission was so successful that data levels rose far above those anticipated. And the lander reset. And again the next day...

Mars Pathfinder

The cause of the Pathfinder resets was indeed priority inversion, complicated by the fact that different threads used different mechanisms to access the bus: the weather observation didn't directly take a lock, it used a pipe that used a call to the file system that ...

NASA fixed it: they ran a duplicate on earth, recreated the reset, dumped a trace and read it carefully. The fix was tricky: it patched the compiled binaries on the probe to modify global flags controlling priority inheritance; inevitably, this affected other code but was reckoned to be safe.

NASA are proud to "test what they fly and fly what they test" — all the monitoring and control code remains on the actual probe. This includes tracing and logging code, and even a command shell. Although the lightspeed lag might make it tricky to use interactively.

See: *What really happened on Mars?*
http://research.microsoft.com/~mbj/Mars_Pathfinder/

Outline

Memory Models

In a concurrent environment, with multiple threads executing over shared memory, a *memory model* describes how reads and writes in one thread are seen by others.

This is nontrivial in an architecture where memory is accessed by one or more processors through layers of caches.

In a *strong* memory model, every thread sees reads and writes in exactly the same order; everything is always flushed out to "real" memory before anything else happens.

In a *relaxed* memory model, this may not hold. Different threads may see things in a different order; they might even see different things.

The Java Memory Model

Shared-memory concurrency is built in to Java, and the details of its memory model are a part of the language definition.

Java has a relaxed memory model, whose details have undergone significant changes over time.

James Gosling, Bill Joy, Guy Steele.
The Java Language Specification (First Edition), Chapter 17.
Addison-Wesley, 1996.

William Pugh (Specification Lead).
JSR 133: Java Memory Model and Thread Specification Revision,
September 2004.

Jaroslav Ševčík.
Program Transformations in Weak Memory Models. PhD Thesis,
University of Edinburgh, 2009.

JMM Audience

The Java memory model (JMM) is addressed to at least three audiences:

- Programmers: writing concurrent code, working out what synchronization is required.
- Compilers: working out appropriate optimized instruction sequences.
- Processors/VMs: executing threads and memory operations; with concurrent execution, instruction-level parallelism, caching, speculative execution,...

JMM content

The JMM is largely in the form of do's and don't's, with accompanying intuition and explanation.

- Programmers: which idioms work, which don't.
- Compilers: what instruction manipulation is allowed, what isn't.
- Processors/VMs: what caching, speculation, parallelism is acceptable, and what isn't.

The JMM does not have a formal semantics, although it does have some theorems. Some of the theorems have proofs. Some of the proofs are valid.

That sounds harsh, but in fact it's remarkable to have even this level of end-to-end certainty in a real-world concurrent language.

JMM Examples

Two simple threads

```
int x=0, y=0

int r1 = x;     |   int r2 = y;
    y = 2;      |       x = 1;
```

Here x and y are shared variables, initially both 0, while r1 and r2 are local variables. Both command sequences run together.

Question: After they have finished, if r1==1, can r2==2 ?

JMM Examples

Two simple threads

```
int x=0, y=0

int r1 = x;     |   int r2 = y;
    y = 2;      |       x = 1;
```

Here x and y are shared variables, initially both 0, while r1 and r2 are local variables. Both command sequences run together.

Question: After they have finished, if r1==1, can r2==2 ?

Answer: Yes, absolutely. Just reorder these independent instructions. Which might be done by the compiler, the processor, or the memory cache.

In Java this is legal behaviour whenever a *data race* is present.

JMM Guarantees

What has been lost here is *sequential consistency*: the idea that whatever really happens, the final outcome must be equivalent to some interleaving of operations which agrees with the order they appear in the program source.

The JMM guarantees sequential consistency in any program without data races.

Separately, it recognizes that data races might be intentional, and provides some sensible limits on behaviour in those cases.

Even so, the JMM is far from intuitive: it has no global time, and no global store. There are just actions, some causally ordered, some visible from particular threads.

Speculation

Two simple threads

```
int x=0, y=0

r1 = x;          |    r2 = y;
if (r1==0)       |     x = r2;
  { y=42; }      |
else             |
  { y=r1; }      |
```

Question: After the two threads have finished, can r1==r2==42 ?

Speculation

Two simple threads

```
int x=0, y=0

r1 = x;          |     r2 = y;
if (r1==0)       |      x = r2;
  { y=42; }      |
else             |
  { y=r1; }      |
```

Question: After the two threads have finished, can r1==r2==42 ?

Answer: Yes, absolutely. The left-hand thread may speculatively execute the first branch, then retract that, but not before the other thread has seen the value in y, assigned it to r2,...

Speculation

Two simple threads

```
int x=0, y=0

r1 = x;          |      r2 = y;
if (r1==0)       |       x = r2;
  { y=42; }      |
else             |
  { y=r1; }      |
```

Question: After the two threads have finished, can r1==r2==42 ?

Answer: Yes, absolutely.

This is explicitly legal under the JMM. However, the "out of thin air" constraint does place some limits on what can happen.

Summary

Locks: reentrant, may deadlock, hand-over-hand locking does not nest.

Programming concurrent locks is difficult.

Priority inversion: low-priority threads may block high-priority ones.

Programming concurrent priorities is difficult.

Relaxed memory model: races, sequential consistency, reordering, speculation.

Programming concurrent architectures is difficult.