CS 242

Concurrency

John Mitchell

Announcements

- Last homework due December 3
- Final exam – Monday, December 8
  - 8:30-11:30 AM
  - TCSEQ 200
  - *Refreshments??*

Concurrency

Two or more sequences of events occur in parallel

- Multiprogramming
  - A single computer runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another
- Multiprocessors
  - Two or more processors may be connected
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

Process: sequential program running on a processor

The promise of concurrency

- Speed
  - If a task takes time t on one processor, shouldn't it take time t/n on n processors?
- Availability
  - If one process is busy, another may be ready to help
- Distribution
  - Processors in different locations can collaborate to solve a problem or work together
- Humans do it so why can't computers?
  - Vision, cognition appear to be highly parallel activities

Challenges

- Concurrent programs are harder to get right
  - Folklore: Need an order of magnitude speedup (or more) to be worth the effort
- Some problems are inherently sequential
  - Theory – circuit evaluation is P-complete
  - Practice – many problems need coordination and communication among sub-problems
- Specific issues
  - Communication – send or receive information
  - Synchronization – wait for another process to act
  - Atomicity – do not stop in the middle and leave a mess

Why is concurrent programming hard?

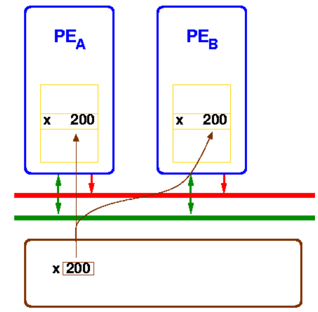
- Nondeterminism
  - *Deterministic*: two executions on the same input it always produce the same output
  - *Nondeterministic:* two executions on the same input may produce different output
- Why does this cause difficulty?
  - May be many possible executions of one system
  - Hard to think of all the possibilities
  - Hard to test program since some errors may occur infrequently

Example

- Cache coherence protocols in multiprocessors
  - A set of processors share memory
  - Access to memory is slow, can be bottleneck
  - Each processor maintains a memory cache
  - The job of the cache coherence protocol is to maintain the processor caches, and to guarantee that the values returned by every load/store sequence generated by the multiprocessor are consistent with the memory model.

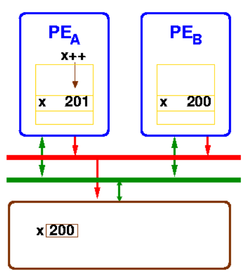
Cache filled by read

- PE_A reads loc x
  - Copy of x put in PE_A's cache.
- PE_B also reads x
  - Copy of x put in PE_B's cache too.

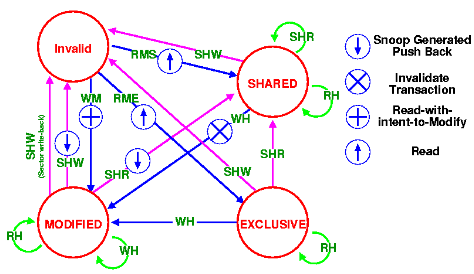


Cache modified by write

- PE_A adds 1 to x
  - x is in PE_A's cache, so there's a cache hit
- If PE_B reads x from cache, *may* be wrong
  - OK if program semantics allows PE_B read before PE_A write
- Need protocol to avoid using stale values



State diagram for cache protocol



- Necessary for multiprocessor; hard to get right.

Basic question for this course

- How can programming languages make concurrent and distributed programming easier?
  - Can do concurrent, distributed programming in C using system calls
  - Is there something better?

What could languages provide?

- Abstract model of system
  - abstract machine => abstract system
- Example high-level constructs
  - Process as the value of an expression
    - Pass processes to functions
    - Create processes at the result of function call
  - Communication abstractions
    - Synchronous communication
    - Buffered asynchronous channels that preserve msg order
  - Mutual exclusion, atomicity primitives
    - Most concurrent languages provide some form of locking
    - Atomicity is more complicated, less commonly provided

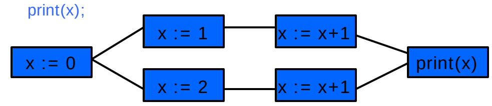
Basic issue: conflict between processes

- Critical section
  - Two processes may access shared resource
  - Inconsistent behavior if two actions are interleaved
  - Allow only one process in *critical section*
- Deadlock
  - Process may hold some locks while awaiting others
  - *Deadlock* occurs when no process can proceed

Cobegin/coend

- Limited concurrency primitive
- Example

```
x := 0;
cobegin
   begin x := 1; x := x+1 end;   } execute sequential
   begin x := 2; x := x+1 end;     blocks in parallel
coend;
print(x);
```



Atomicity at level of assignment statement

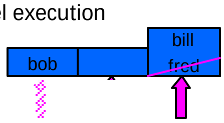
Mutual exclusion

- Sample action

```
procedure sign_up(person)
   begin
     number := number + 1;
     list[number] := person;
end;
```

- Problem with parallel execution

```
cobegin
   sign_up(fred);
   sign_up(bill);
end;
```



Locks and Waiting

```
<initialze concurrency control>
cobegin
   begin
      <wait>
      sign_up(fred);   // critical section
      <signal>
   end;
   begin
      <wait>
      sign_up(bill);   // critical section
      <signal>
   end;
end;
```

Need atomic operations to implement wait

Mutual exclusion primitives

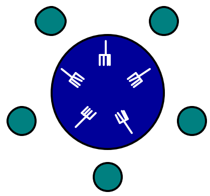
- Atomic test-and-set
  - Instruction atomically reads and writes some location
  - Common hardware instruction
  - Combine with busy -waiting loop to implement mutex
- Semaphore
  - Avoid busy -waiting loop
  - Keep queue of waiting processes
  - Scheduler has access to semaphore; process sleeps
  - Disable interrupts during semaphore operations
    - OK since operations are short

Monitor Brinch-Hansen, Dahl, Dijkstra, Hoare

- Synchronized access to private data. Combines:
  - private data
  - set of procedures (methods)
  - synchronization policy
    - At most one process may execute a monitor procedure at a time; this process is said to be *in* the monitor.
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed.
- Modern terminology: synchronized object

Deadlock

- Possible with any mutual exclusion primitive
- Example
  - Dining philosophers
  - Each needs two forks to eat a plate of pasta
  - Each picks up fork to left, all at same time
  - None one can eat



Some entertaining Java animations on web

Reality

- Concurrent programming is difficult
  - Race conditions, deadlock are pervasive in Java libraries, etc.
- Languages should be able to help
  - Capture useful paradigms, patterns, abstractions
- Other tools are needed
  - Testing is difficult for multi-threaded programs
  - Many race-condition detectors being built today

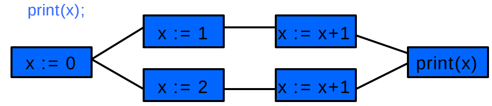
Concurrent language examples

- Language Examples
  - Cobegin/coend
  - Actors (C. Hewitt)
  - Concurrent ML
  - Java
- Main features to compare
  - Threads
  - Communication
  - Synchronization
  - Atomicity

Cobegin/coend

- Limited concurrency primitive
- Example

```
x := 0;
cobegin
   begin x := 1; x := x+1 end;   } execute sequential
   begin x := 2; x := x+1 end;   } blocks in parallel
coend;
print(x);
```



Atomicity at level of assignment statement

Properties of cobegin/coend

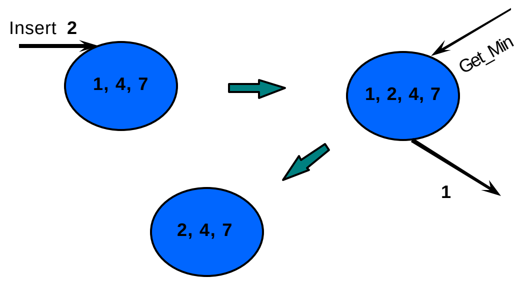
- Advantages
  - Create concurrent processes
  - Communication: shared variables
- Limitations
  - Mutual exclusion: none
  - Atomicity: none
  - Number of processes is fixed by program structure
  - Cannot abort processes
    - All must complete before parent process can go on

History: Concurrent Pascal, P. Brinch Hansen, Caltech, 1970's

Actors [Hewitt, Agha, Tokoro, Yonezawa, ...]

- Each actor (object) has a script
- In response to input, actor may atomically
  - create new actors
  - initiate communication
  - change internal state
- Communication is
  - Buffered, so no message is lost
  - Guaranteed to arrive, but not in sending order
    - Order-preserving communication is harder to implement
    - Programmer can build ordered primitive from unordered
    - Inefficient to have ordered communication when not needed

Example



Actor program

- Stack node

  parameters

  a stack_node with acquaintances content and link

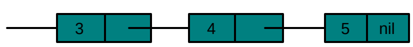
  if operation requested is a pop and content != nil then
  become forwarder to link
  send content to customer

  if operation requested is push(new_content) then
  let P=new stack_node with current acquaintances (a clone)
  become stack_node with acquaintances new_content and P

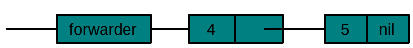
Hard to read but it does the "obvious" thing, except that the concept of *forwarder* is unusual....

Forwarder

- Stack before pop



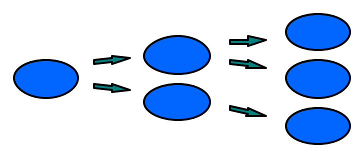
- Stack after pop



  - Node "disappears" by becoming a forwarder node. The system manages forwarded nodes in a way that makes them invisible to the program. (Exact mechanism doesn't really matter since we're not that interested in Actors.)

Concurrency and Distribution

- Several actors may operate concurrently



- Concurrency not forced by program
  - Depends on system scheduler
- Distribution not controlled by programmer

Attractive idealization, but too "loose" in practice.

Concurrent ML [Reppy, Gansner, ...]

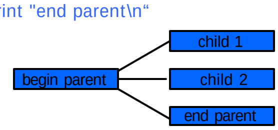
- Threads
  - New type of entity
- Communication
  - Synchronous channels
- Synchronization
  - Channels
  - Events
- Atomicity
  - No specific language support

Threads

- Thread creation
  - spawn : (unit → unit) → thread_id
- Example code

```
CIO.print  "begin parent\n";
     spawn (fn () => (CIO.print "child 1\n";));
     spawn (fn () => (CIO.print "child 2\n";));
CIO.print "end parent\n"
```

- Result



Channels

- Channel creation
  - channel : unit → 'a chan
- Communication
  - recv : 'a chan → 'a
  - send : ('a chan * 'a) → unit
- Example

```
ch = channel();
spawn (fn()=> ... <A> ... send(ch,0); ... <B> ...);
spawn (fn()=> ... <C> ... recv ch; ...     <D> ...);
```

- Result

CML programming

- Functions
  - Can write functions : channels → threads
  - Build concurrent system by declaring channels and "wiring together" sets of threads
- Events
  - Delayed action that can be used for synchronization
  - Powerful concept for concurrent programming
- Sample Application
  - eXene – concurrent uniprocessor window system

Sample CML programming

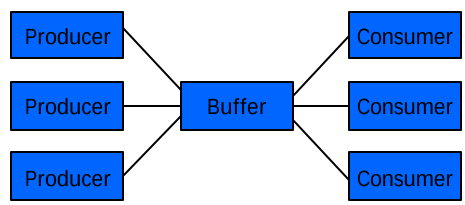
- Function to create squaring process

```
fun square (inCh, outCh) =
   forever () (fn () =>
     send (outCh, square(recv(inCh))));
```

- Put processes together

```
fun mkSquares () =
let
   val outCh = channel()
   and c1 = channel()
in
   numbers(c1);
   square(c1, outCh);
   outCh
end;
```

Problem: Producer-Consumer



- Easy with buffered asynchronous communication
- Requires buffer if synchronous communication

Synchronous consumer or buffer ???

- Code probably looks like this:

```
for i = 1 to n
    receive(... producer[i] ...)
```

- What's the problem?
  - Synchronous receive blocks waiting for sender
  - Deadlock if
    - Producer 1 is ready to send
    - Producer 2 is finished (nothing left to send)
    - Consumer or queue decides to receive from Producer 2
- How do we solve this problem?

Guarded Commands [Dijkstra]

- Select one available command; non-blocking test

```
do
   Condition ⇒ Command
   ...
   Condition ⇒ Command
od
```

- Outline of producer-consumer buffer

```
do
   Producer ready and queue not full ⇒
        Receive from waiting producer and store in queue
   Consumer ready and queue not empty ⇒
        Send to waiting consumer and remove from queue
od
```

Expressiveness of CML

- How do we write choice of guarded commands?
  - Events and "choose" function
- CML Event = "delayed" action
  - 'a event
    - the type of actions that return an 'a when executed
  - sync : 'a event → 'a
    - Function that synchronizes on an 'a event and returns an 'a
  - fun recv(ch) = sync (recvEvt (ch));
- Choice
  - choose : 'a event list → 'a event

Does not seem possible to do producer-consumer in CML without choose

CML from continuations

- Continuation primitives
  - callcc : ('a cont → 'a) → 'a
    Call function argument with current continuation
  - throw : 'a cont -> 'a -> 'b
  - Curried function to invoke continuation with arg
- Example

```
fun f(x,k) = throw k(x+3);
fun g(y,k) = f(y+2,k) + 10;
fun h(z) = z + callcc(fn k => g(z+1,k));
h(1);
```

A CML implementation (simplified)

- Use queues with side-effecting functions

```
datatype 'a queue = Q of {front: 'a list ref, rear: 'a list ref}
fun queueIns (Q(...))(...) = (* insert into queue *)
fun queueRem (Q(...)) = (* remove from queue *)
```

- And continuations

```
val enqueue = queueIns rdyQ
fun dispatch () = throw (queueRem rdyQ) ()
fun spawn f = callcc (fn parent_k =>
                  ( enqueue parent_k; f (); dispatch()))
```

Source: Appel, Reppy

Java Concurrency

- Threads
  - Create process by creating thread object
- Communication
  - shared variables
  - method calls
- Mutual exclusion and synchronization
  - Every object has a lock (inherited from class Object)
    - synchronized methods and blocks
  - Synchronization operations (inherited from class Object)
    - wait : pause current thread until another thread calls notify
    - notify : wake up waiting threads

Java Threads

- Thread
  - Set of instructions to be executed one at a time, in a specified order
- Java thread objects
  - Object of class Thread
  - Methods inherited from Thread:
    - start : method called to spawn a new thread of control; causes VM to call run method
    - suspend : freeze execution
    - interrupt : freeze execution and throw exception to thread
    - stop : forcibly cause thread to halt

Example subclass of Thread

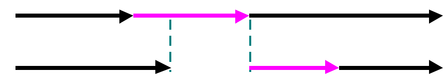
```
class PrintMany extends Thread {
    private String msg;
    public PrintMany (String m) {msg = m;}
    public void run() {
            try {  for (;;){ System.out.print(msg + " ");
                             sleep(10);
                    }
            } catch (InterruptedException e) {
                    return;
            }
    }                          (inherits start from Thread)
```

Interaction between threads

- Shared variables
  - Two threads may assign/read the same variable
  - Programmer responsibility
    - Avoid race conditions by explicit synchronization!!
- Method calls
  - Two threads may call methods on the same object
- Synchronization primitives
  - Each object has internal lock, inherited from Object
  - Synchronization primitives based on object locking

Synchronization example

- Objects may have *synchronized* methods
- Can be used for mutual exclusion
  - Two threads may share an object.
  - If one calls a synchronized method, this locks object.
  - If the other calls a synchronized method on same object, this thread blocks until object is unlocked.



Synchronized methods

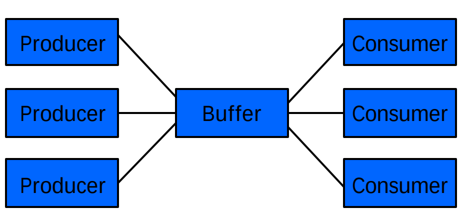
- Marked by keyword

  `public synchronized void commitTransaction(...) {...}`
- Provides mutual exclusion
  - At most one synchronized method can be active
  - Unsynchronized methods can still be called
    - Programmer must be careful
- Not part of method signature
  - sync method equivalent to unsync method with body consisting of a *synchronized block*
  - subclass may replace a synchronized method with unsynchronized method

Example [Lea]

```
class LinkedCell {            // Lisp-style cons cell containing
   protected double value;    // value and link to next cell
   protected LinkedCell next;
   public LinkedCell (double v, LinkedCell t) {
          value = v; next = t;
   }
   public synchronized double getValue() {
          return value;
   }
   public synchronized void setValue(double v) {
          value = v;   // assignment not atomic
   }
   public LinkedCell next() {   // no synch needed
          return next;
   }
```

Producer-Consumer?



- Method call is synchronous
- How do we do this in Java?

Join, another form of synchronization

- Wait for thread to terminate

```
class Future extends Thread {
    private int result;
    public void run() {  result = f(…); }
    public int getResult() { return result;}
}
…
Future t = new future;
t.start()                         // start new thread
…
t.join(); x = t.getResult(); // wait and get result
```

Priorities

- Each thread has a priority
  - Between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY
    - These are 1 and 10, respectively
    - Main has default priority Thread.NORM_PRIORITY (=5)
  - New thread has same priority as thread created it
  - Current priority accessed via method getPriority
  - Priority can be dynamically changed by setPriority
- Schedule gives preference to higher priority

ThreadGroup

- Every Thread is a member of a ThreadGroup
  - Default: same group as creating thread
  - ThreadGroups nest in a tree-like fashion
- ThreadGroup support security policies
  - Illegal to interrupt thread not in your group
  - Prevents applet from killing main screen display update thread
- ThreadGroups not normally used directly
  - collection classes (for example java.util.Vector) are better choices for tracking groups of Thread objects
- ThreadGroup provides method uncaughtException
  - invoked when thread terminates due to uncaught unchecked exception (for example a NullPointerException)

Aspects of Java Threads

- Portable since part of language
  - Easier to use in basic libraries than C system calls
  - Example: garbage collector is separate thread
- General difficulty combining serial/concur code
  - Serial to concurrent
    - Code for serial execution may not work in concurrent sys
  - Concurrent to serial
    - Code with synchronization may be inefficient in serial programs (10-20% unnecessary overhead)
- Abstract memory model
  - Shared variables can be problematic on some implementations

Concurrent garbage collector

- How much concurrency?
  - Need to stop thread while mark and sweep
  - Other GC: may not need to stop all program threads
- Problem
  - Program thread may change objects during collection
- Solution
  - Prevent read/write to memory area
  - Details are subtle; generational, copying GC
    - Modern GC distinguishes short-lived from long-lived objects
    - Copying allows read to old area if writes are blocked ...
    - Relatively efficient methods for read barrier, write barrier

Some rough spots

- Class may have synchronized, unsynch methods
  - no notion of a class which is a monitor
- Immutable objects (final fields)
- Fairness is not guaranteed
  - Chose arbitrarily among equal priority threads
- Condition rechecks essential
  - use loop – see next slide
- Wait set is not a FIFO queue
  - notifyAll notifies all waiting threads, not necessarily highest priority, one waiting longest, etc.

Condition rechecks

- Want to wait until condition is true

```
public synchronized void lock() throws InterruptedException {
        if ( isLocked ) wait();
        isLocked = true;
}
public synchronized void unLock() {
        isLocked = false;
        notify();
}
```

- But need loop since another process may run

```
public synchronized void lock() throws InterruptedException {
        while ( isLocked ) wait();
        isLocked = true;
}
```

Allen Holub, *Taming Java Threads*

Problem with language specification

- Java Lang Spec allows access to partial objects

```
class Broken {
   private long x;
   Broken() {
      new Thread() {
          public void run() { x = -1; }
      }.start();
      x = 0;
} }
```

Thread created within constructor can access the object not fully constructed

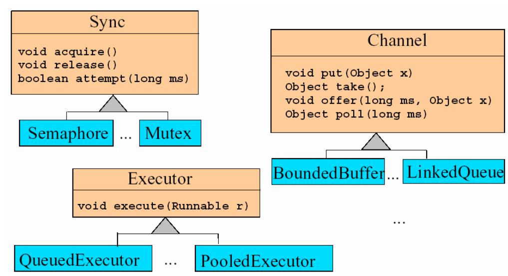
Nested Monitor Lockout Problem

```
class Stack {
   LinkedList list = new LinkedList();
   public synchronized void push(Object x) {
        synchronized(list) {
             list.addLast( x ); notify();
   } }
   public synchronized Object pop() {
        synchronized(list) {
             if( list.size() <= 0 ) wait();
             return list.removeLast();
   } }
}
```

Releases lock on Stack object but not lock on list;
a push from another thread will deadlock

Java progress: util.concurrent

- Doug Lea's utility classes, basis for JSR 166
  - A few general-purpose interfaces
  - Implementations tested over several years
- Principal interfaces and implementations
  - Sync: acquire/release protocols
  - Channel: put/take protocols
  - Executor: executing Runnable tasks



Sync

- Main interface for acquire/release protocols
  - Used for custom locks, resource management, other common synchronization idioms
  - Coarse-grained interface
    - Doesn't distinguish different lock semantics
- Implementations
  - Mutex, ReentrantLock, Latch, CountDown, Semaphore, WaiterPreferenceSemaphore, FIFOSemaphore, PrioritySemaphore
    - Also, utility implementations such as ObservableSync, LayeredSync that simplifycomposition and instrumentation

Channel

- Main interface for buffers, queues, etc.

- Implementations
  - LinkedQueue, BoundedLinkedQueue, BoundedBuffer, BoundedPriorityQueue, SynchronousChannel, Slot

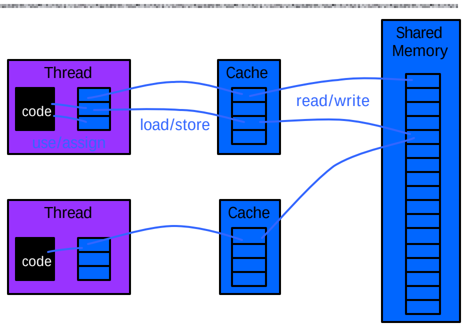
Executor

- Main interface for Thread-like classes
  - Pools
  - Lightweight execution frameworks
  - Custom scheduling
- Need only support execute(Runnable r)
  - Analogous to Thread.start
- Implementations
  - PooledExecutor, ThreadedExecutor, QueuedExecutor, FJTaskRunnerGroup
  - Related ThreadFactory class allows most Executors to use threads with custom attributes

Java memory model

- Main ideas
  - Threads have local memory (cache)
  - Threads fill/flush from main memory
- Interaction restricted by constraints on actions
  - Use/assign are local thread memory actions
  - Load/store fill or flush local memory
  - Read/write are main memory actions

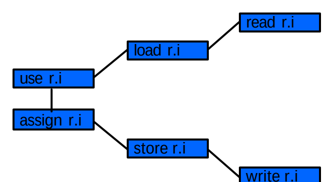
Memory Hierarchy



Example

- Program

  r.i = r.i+1



- The value of field i of object i is
  - *read* from main memory
  - *load*ed into the local cache of the thread
  - *use*d in the addition r.i+1
- Similar steps to place the value of r.i in shared memory

Java Memory Model [Java Lang Spec]

- Example constraints on use, assign, load, store:
  - *use* and *assign* actions by thread must occur in the order specified by the program
  - Thread is not permitted to lose its most recent assign
  - Thread is not permitted to write data from its working memory to main memory for no reason
  - New thread starts with an empty working memory
  - New variable created only in main memory, not thread working memory
- "Provided that all the constraints are obeyed, a *load* or *store* action may be issued at any time by any thread on any variable, at the whim of the implementation."

Access to Main Memory

- Constraints on load, store, read ,write
  - For every *load*, must be a preceding *read* action
  - For every *store*, must be a following *write* action
  - Actions on master copy of a variable are performed by the main memory in order requested by thread

Prescient stores

- Under certain conditions …
  - Store actions (from cache to shared memory) may occur earlier than you would otherwise expect
  - Purpose:
    - Allow optimizations that make properly synchronized programs run faster
    - These optimizations may allow out-of-order operations for programs that are not properly synchronized

Details are complicated. Main point: there's more to designing a good memory model than you might think!

Criticism [Pugh]

- Model is hard to interpret and poorly understood
- Constraints
  - prohibit common compiler optimizations
  - expensive to implement on existing hardware
- Not commonly followed
  - Java Programs
    - Sun Java Development Kit not guaranteed valid by the existing Java memory model
  - Implementations not compliant
    - Sun Classic Wintel JVM, Sun Hotspot Wintel JVM, IBM 1.1.7b Wintel JVM, Sun production Sparc Solaris JVM, Microsoft JVM

Prescient stores anomaly [Pugh]

- Program

```
x = 0; y = 0;
Thread 1:    a = x; y = 1;
Thread 2:    b = y; x = 1;
```

- Without prescient stores
  - Either a=b=0, or a=0 and b=1, or a=1 and b=0
- With prescient stores
  - Write actions for x,y may occur before either read
  - Threads can finish with a=b=1

Homework: draw out ordering on memory operations

Over-constrained actions [Pugh]

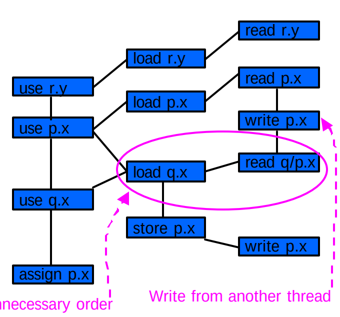
- Program

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x from another thread
k = q.x;
p.x = 42;
```

- Problem
  - Memory model is too constrained
    - Programmer will be happy if j, k get same value
    - Memory model *prevents* this

Constraints on memory actions

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write
k = q.x;
p.x = 42;
```



Summary

- Concurrency
  - Powerful computing idea
  - Requires time and effort to use effectively
- Actors
  - High-level object-oriented form of concurrency
- Concurrent ML
  - Threads and synchronous events
- Java concurrency
  - Combines thread and object-oriented approaches
  - Experience leads to programming methods, libraries

  Example: ConcurrentHashMap