CS 242 2012

Concurrency

Reading (two lectures)

Chapter 14, except section 14.3 and except pages 461-464

JSR 133 (Java Memory Model) FAQ

Note: book presentation of memory model is obsolete

Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

Concurrency

Two or more sequences of events occur in parallel

Note:

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

Concurrency on machines

- 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

Challenges

- Concurrent programs are harder to get right
  - Folklore: Need at least an order of magnitude in speedup for concurrent prog 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

Basic question for this course

- How can programming languages make concurrent programming easier?
- Which abstractions are most effective?
  - What are the advantages and disadvantages of various approaches?

Apart from basic concepts, this lecture covers past ideas (cobegin / coend, actor model) and current Java.

Next week we look at forward-looking research ideas.

Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

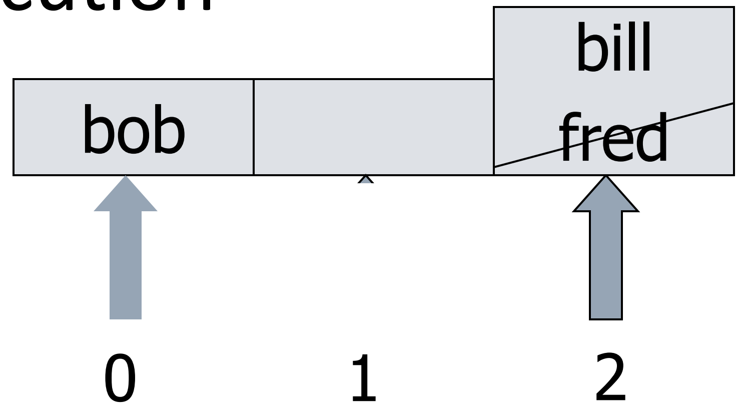
Basic issue: race conditions

- Sample action

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

- Problem with parallel execution

```
sign_up(fred) || sign_up(bill);
```



Resolving 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*

- Potential solution: Locks?
- Problem: Deadlock
  - Process may hold some locks while awaiting others
  - *Deadlock* occurs when no process can proceed

Locks and Waiting

```
<initialize concurrency control>

Thread 1:
        <wait>
        sign_up(fred);  // critical section
        <signal>

Thread 2:
        <wait>
        sign_up(bill);    // critical section
        <signal>
```

Need atomic operations to implement wait

Mutual exclusion primitives

- Atomic test-and-set
  - Instruction atomically reads and writes some location
  - Common hardware instruction
  - Used to implement a busy-waiting loop to get mutual exclusion
- 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

State of the art

- Concurrent programming is difficult
  - Race conditions, deadlock are pervasive
- 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
    - Static detection: conservative, may be too restrictive
    - Run-time detection: may be more practical for now

Outline

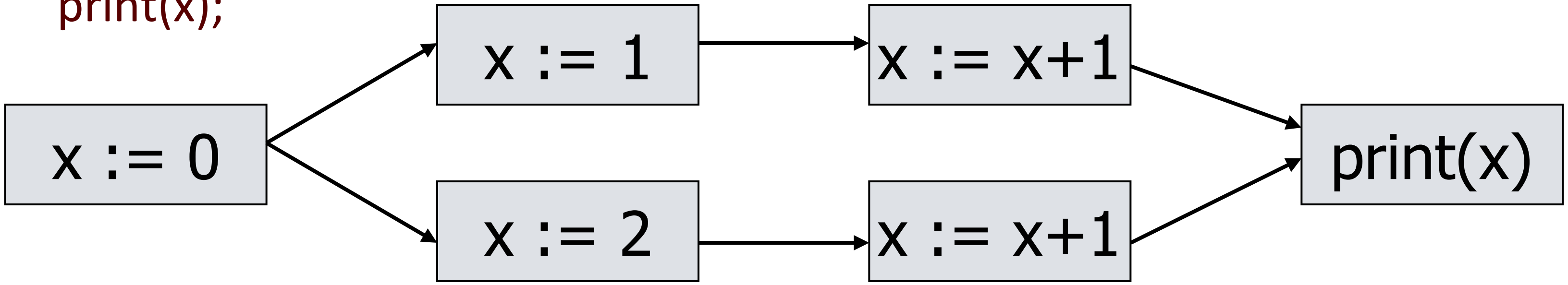
- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

Cobegin/coend

- Limited concurrency primitive
- Example

```
x := 0;
cobegin
    begin x := 1; x := x+1 end;
    begin x := 2; x := x+1 end;
coend;
print(x);
```

execute sequential blocks in parallel



Atomicity at level of assignment statement

Properties of cobegin/coend

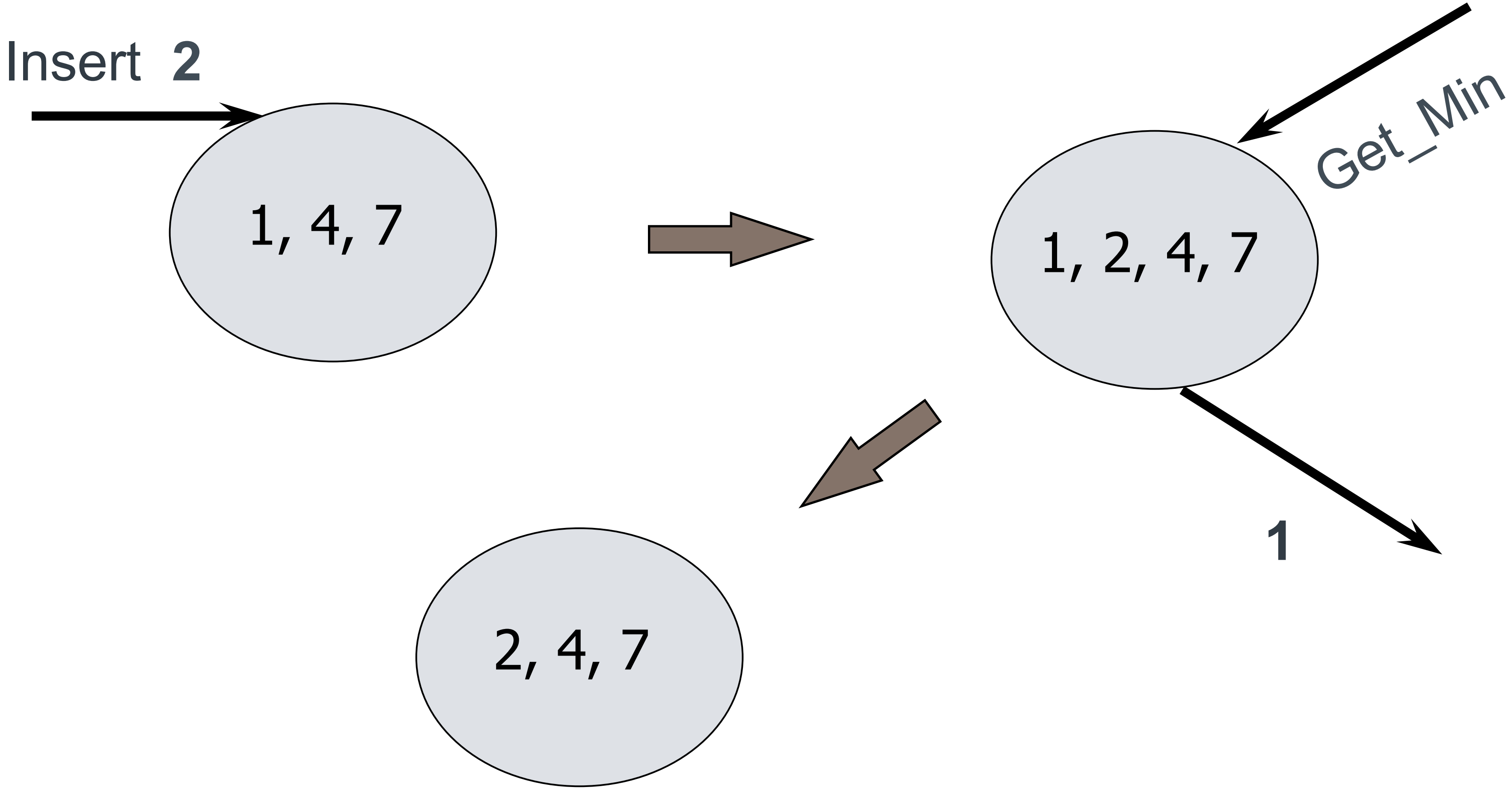
- Advantages
  - Create concurrent processes
  - Communication: Shared variables
- Limitations
  - Mutual exclusion: none
  - Atomicity: none
  - Number of processes is fixed by program structure
  - Cannot abort processes
    - All must complete before parent process can go on

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

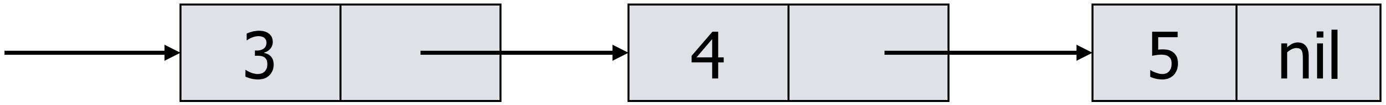
- Stack node

```
                  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

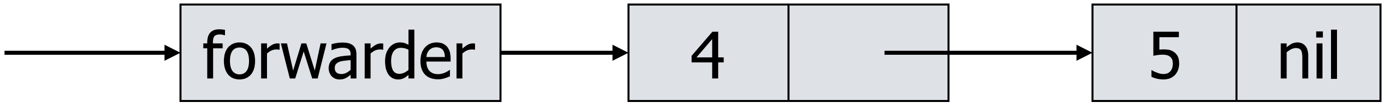


3

4

5 nil

- Stack after pop



forwarder

4

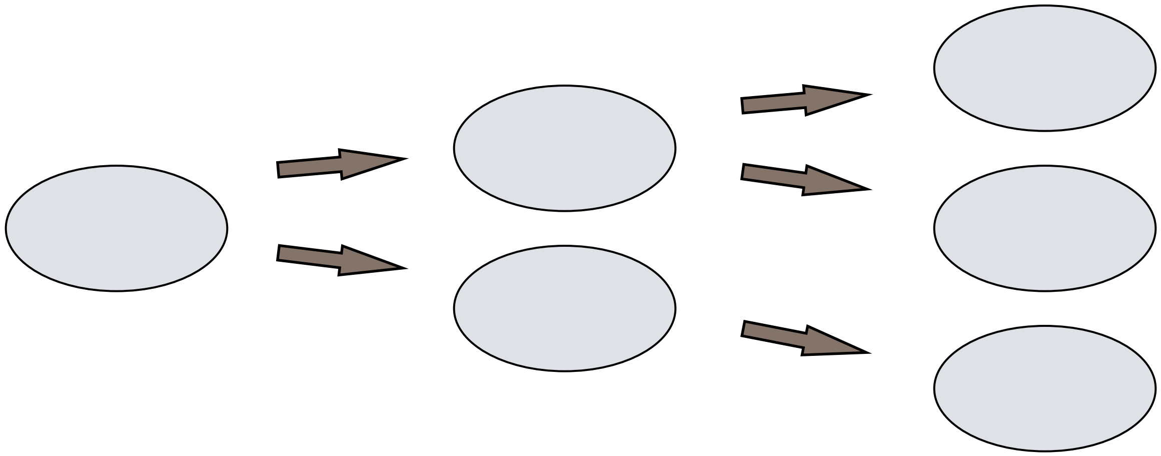
5 nil

- 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 matter ….)

Concurrency

- Several actors may operate concurrently



- Concurrency not controlled explicitly by program
  - Messages sent by one actor can be received and processed by others sequentially or concurrently

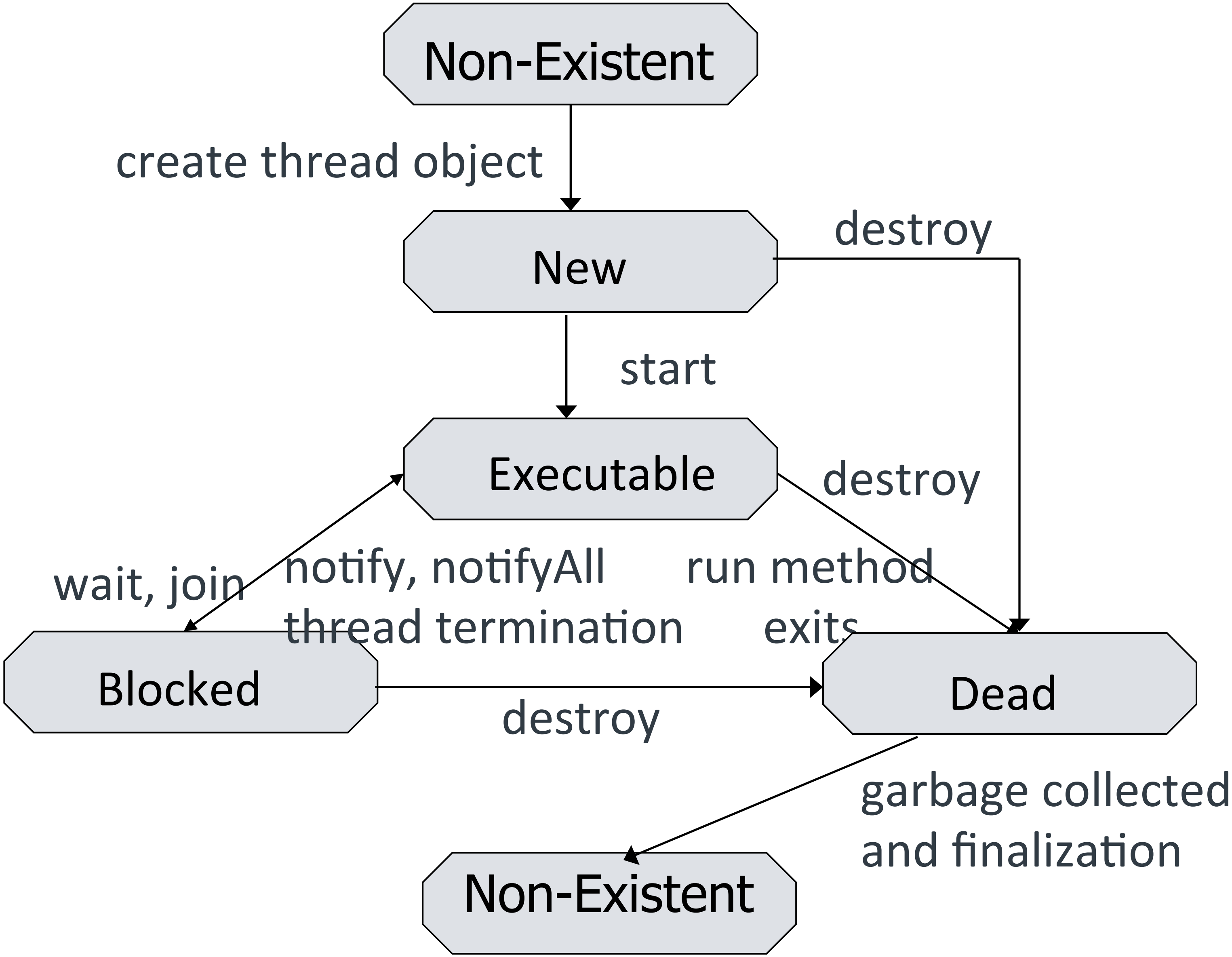
Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

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 Thread States



Non-Existent

create thread object

New

destroy

start

Executable

destroy

wait, join

notify, notifyAll
thread termination

run method
exits

Blocked

Dead

destroy

garbage collected
and finalization

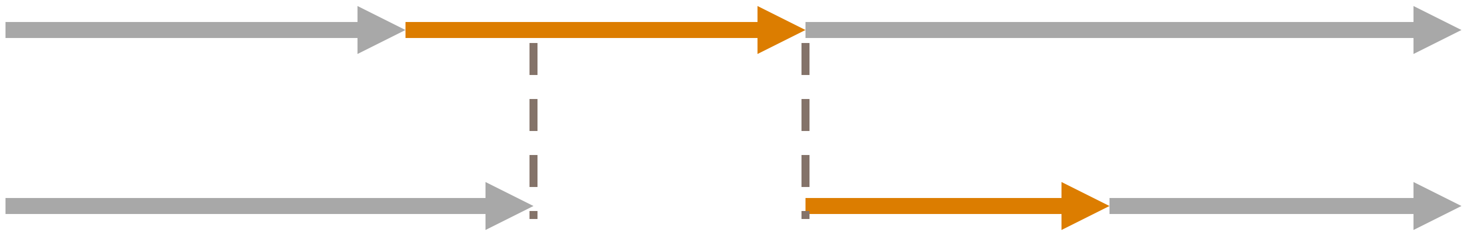
Non-Existent

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

- Provides mutual exclusion
  - Two threads may have access to some 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(…) {…}
```

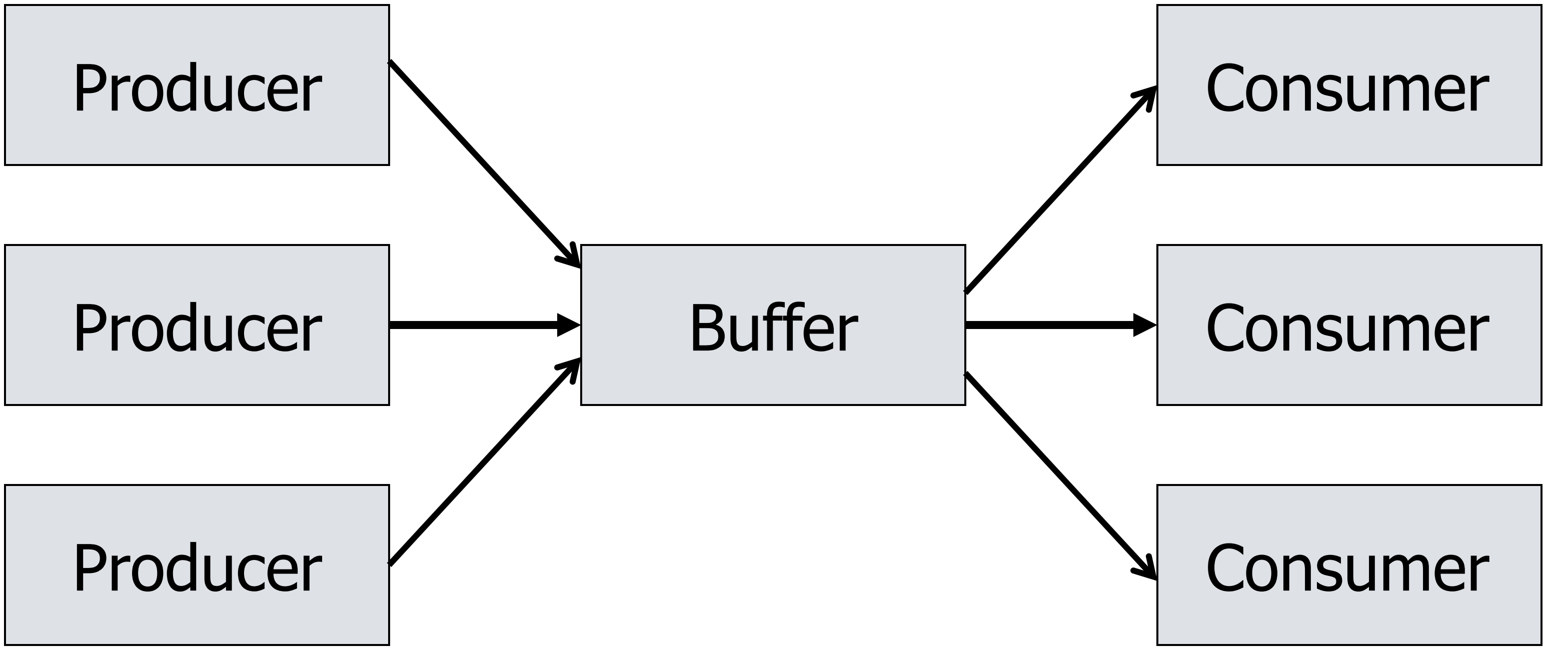
- Not part of method signature
  - sync method equivalent to unsync method with body consisting of a *synchronized block*
  - subclass may replace a synchronized method with unsynchronized method

Example

[Lea]

```
class LinkedCell {          // Lisp-style cons cell containing
    protected double value;  // value and link to next cell
    protected final 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



- How do we do this in Java?

Solution to producer-consumer

- Basic idea
  - Consumer must **wait** until something is in the buffer
  - Producer must **notify** waiting consumers when item available
- More details
  - Consumer waits
    - While waiting, must *sleep* – accomplished with the wait method
    - Need condition recheck loop
  - Producer notifies
    - Must *wake up* at least one consumer
    - This is accomplished with the notify method

Stack<T>: produce, consume methods

```
public synchronized void produce (T object) {
    stack.add(object);
    notify();
}

public synchronized T consume () {
    while (stack.isEmpty()) {
      try {
          wait();
      } catch (InterruptedException e) {
      }
    }
    Int lastElement = stack.size() - 1;
    T object = stack.get(lastElement);
    stack.remove(lastElement);
    return object;
}
```

http://www1.coe.neu.edu/~kokar/java/tut.html

Limitations of Java 1.4 primitives

- No way to back off from an attempt to acquire a lock
  - Cannot give up after waiting for a specified period of time
  - Cannot cancel a lock attempt after an interrupt
- No way to alter the semantics of a lock
  - Reentrancy, read versus write protection, fairness, …
- No access control for synchronization
  - Any method can perform synchronized(obj) for any object
- Synchronization is done within methods and blocks
  - Limited to block-structured locking
  - Cannot acquire a lock in one method and release it in another

See http://java.sun.com/developer/technicalArticles/J2SE/concurrency/

Concurrency references

- Thread-safe classes
  - B Venners, Designing for Thread Safety, JavaWorld, July 1998: http://www.artima.com/designtechniques/threadsafety.html
- Nested monitor lockout problem
  - http://www-128.ibm.com/developerworks/java/library/j-king.html?dwzone=java
- Inheritance anomaly
  - G Milicia, V Sassone: The Inheritance Anomaly: Ten Years After, SAC 2004: http://citeseer.ist.psu.edu/647054.html
- Java memory model
  - See http://www.cs.umd.edu/~jmanson/java.html
  - and http://www.cs.umd.edu/users/jmanson/java/journal.pdf
- Race conditions and correctness
  - See slides: lockset, vector-clock algorithms
- Atomicity and tools
  - See http://www.cs.uoregon.edu/activities/summerschool/summer06/

More detail in references than required by course

Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - ➡ Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

Thread safety

- Concept
  - The fields of an object or class always maintain a valid state, as observed by other objects and classes, even when used concurrently by multiple threads
- Why is this important?
  - Classes designed so each method preserves state invariants
    - Example: priority queues represented as sorted lists
  - Invariants hold on method entry and exit
    - If invariants fail in the middle of execution of a method, then concurrent execution of another method call will observe an inconsistent state (state where the invariant fails)
  - What's a "valid state"? Serializability …

Example (two slides)

```
public class RGBColor {
   private int r;    private int g;     private int b;
   public RGBColor(int r, int g, int b) {
      checkRGBVals(r, g, b);
      this.r = r;     this.g = g;     this.b = b;
   }

   …

 private static void checkRGBVals(int r, int g, int b) {
      if (r < 0 || r > 255 || g < 0 || g > 255 ||
         b < 0 || b > 255) {
         throw new IllegalArgumentException();
      }
   }
}
```

Example (continued)

```
public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r;     this.g = g;      this.b = b;
    }

public int[] getColor() {   //  returns array of three ints: R, G, and B
        int[] retVal = new int[3];
        retVal[0] = r;     retVal[1] = g;   retVal[2] = b;
        return retVal;
    }

    public void invert() {
        r = 255 - r;      g = 255 - g;       b = 255 - b;
    }
```

Question: what goes wrong with multi-threaded use of this class?

Some issues with RGB class

- Read/write conflicts
  - If one thread reads while another writes, the color that is read may not match the color before *or* after
- Write/write conflicts
  - If two threads try to write different colors, result may be a “mix” of R,G,B from two different colors

How to make classes thread-safe

1. Synchronize critical sections
   - Make fields private
   - Synchronize sections that should not run concurrently
2. Make objects immutable
   - State cannot be changed after object is created

     ```
     public RGBColor invert() {
       RGBColor retVal = new RGBColor(255 - r, 255 - g, 255 - b);
       return retVal;
     }
     ```

   - Use pure functional programming for concurrency
3. Use a thread-safe wrapper
   - See next slide …

How to make classes thread-safe: thread-safe wrapper

- Idea
  - New thread-safe class has objects of original class as fields
  - Wrapper class provides methods to access original class object

- Example

```
public synchronized void setColor(int r, int g, int b) {
   color.setColor(r, g, b);
}
public synchronized int[] getColor() {
   return color.getColor();
}
public synchronized void invert() {
   color.invert();
}
```

Comparison

- Synchronizing critical sections
  - Good default approach for building thread-safe classes
  - Only way to allow wait() and notify()
- Using immutable objects
  - Good if objects are small, simple abstract data type
  - Benefit: pass to methods without alias issues, unexpected side effects
  - Examples: Java String and primitive type wrappers Integer, Long, Float, etc.
- Using wrapper objects
  - Can give clients choice between thread-safe version and non-safe
  - Works with existing class that is not thread-safe
  - Example: Java 1.2 collections library – classes are not thread safe but some have class method to enclose objects in thread-safe wrapper

Performance issues

- Why not just synchronize everything?
  - Performance costs
  - Possible risks of deadlock from too much locking
- Performance in current Sun JVM
  - Synchronized method are 4 to 6 times slower than non-synchronized
- Performance in general
  - Unnecessary blocking and unblocking of threads can reduce concurrency
  - Immutable objects can be short-lived, increase garbage collector

Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

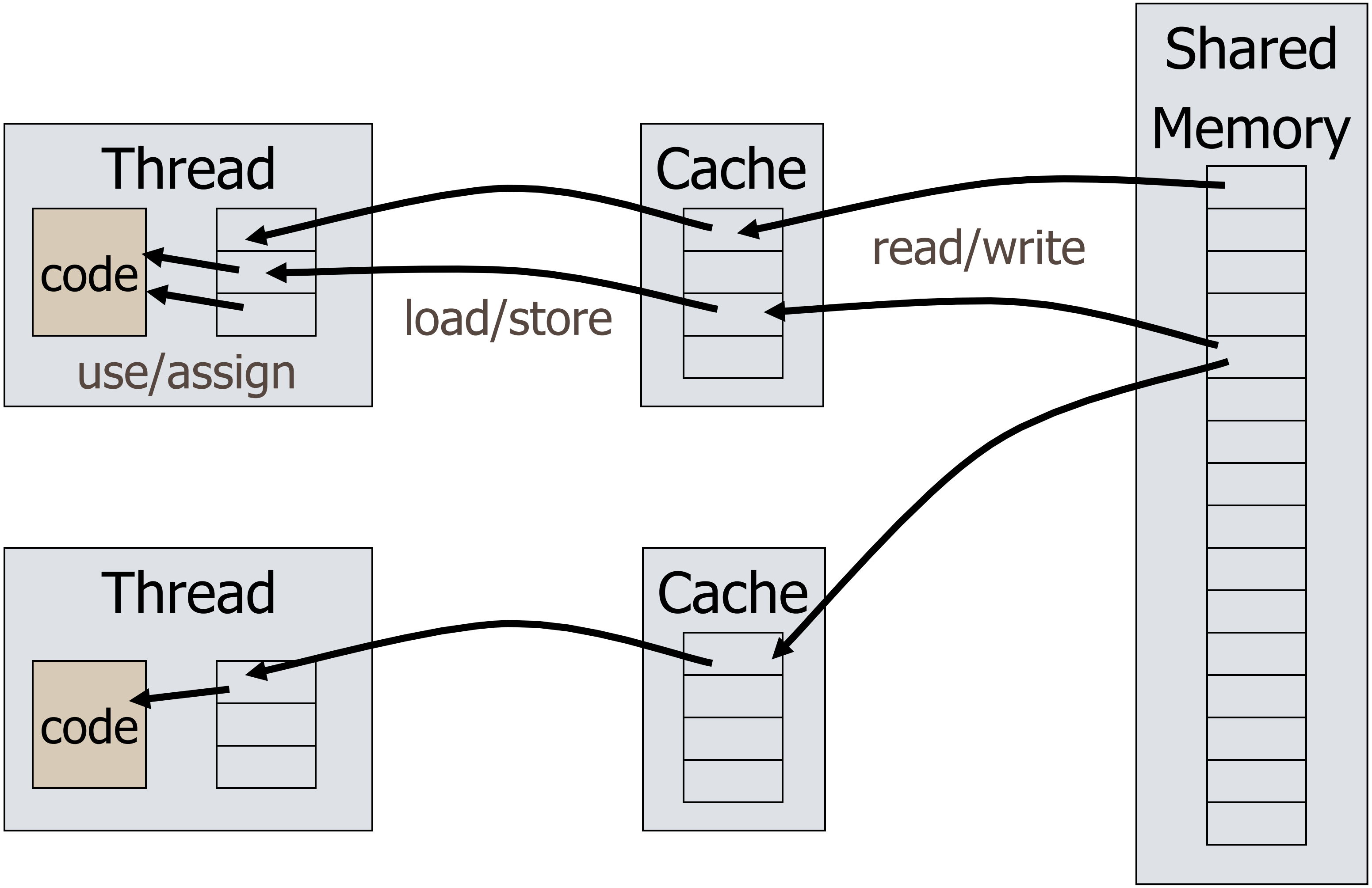
Java Memory Model

- Semantics of multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
  - Need semantics for incorrectly synchronized programs
- Determines
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs may occur on correct implementation
    - Should not be too generous

Reference:

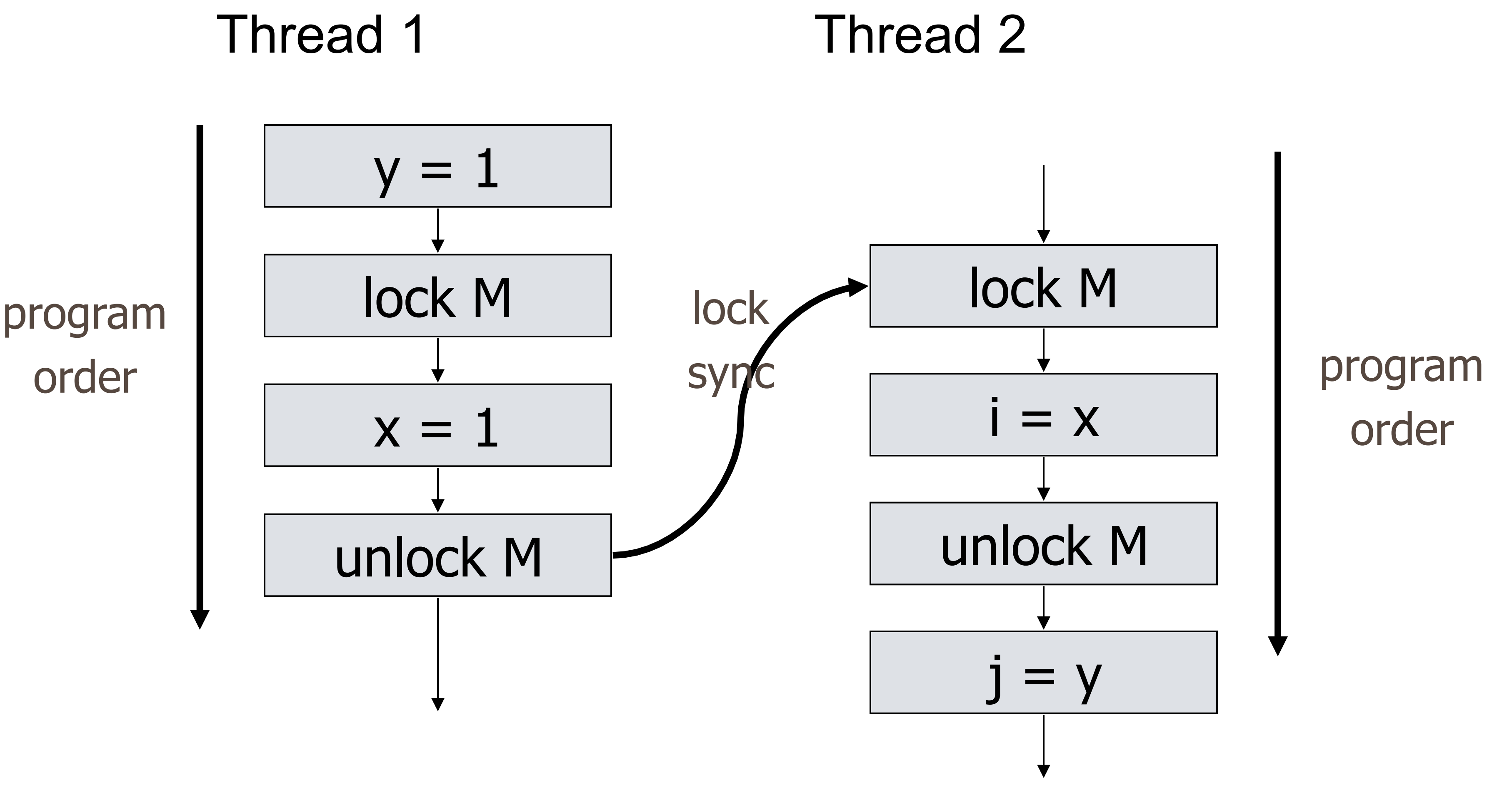
http://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html

Memory Hierarchy



Old memory model placed complex constraints on read, load, store, etc.

Program and locking order



[Manson, Pugh]

Race conditions

- “Happens-before” order
  - Transitive closure of program order and synchronizes-with order
- Conflict
  - An *access* is a read or a write
  - Two accesses *conflict* if at least one is a write
- Race condition
  - Two accesses form a *data race* if they are from different threads, they conflict, and they are not ordered by happens-before

Race cond

Subtle issue: program order as written, or as compiled and optimized ?

- "Happens-before" order
  - Transitive closure of program order and synchronizes-with order
- Conflict
  - An *access* is a read or a write
  - Two accesses *conflict* if at least one is a write
- Race condition
  - Two accesses form a *data race* if they are from different threads, they conflict, and they are not ordered by happens-before

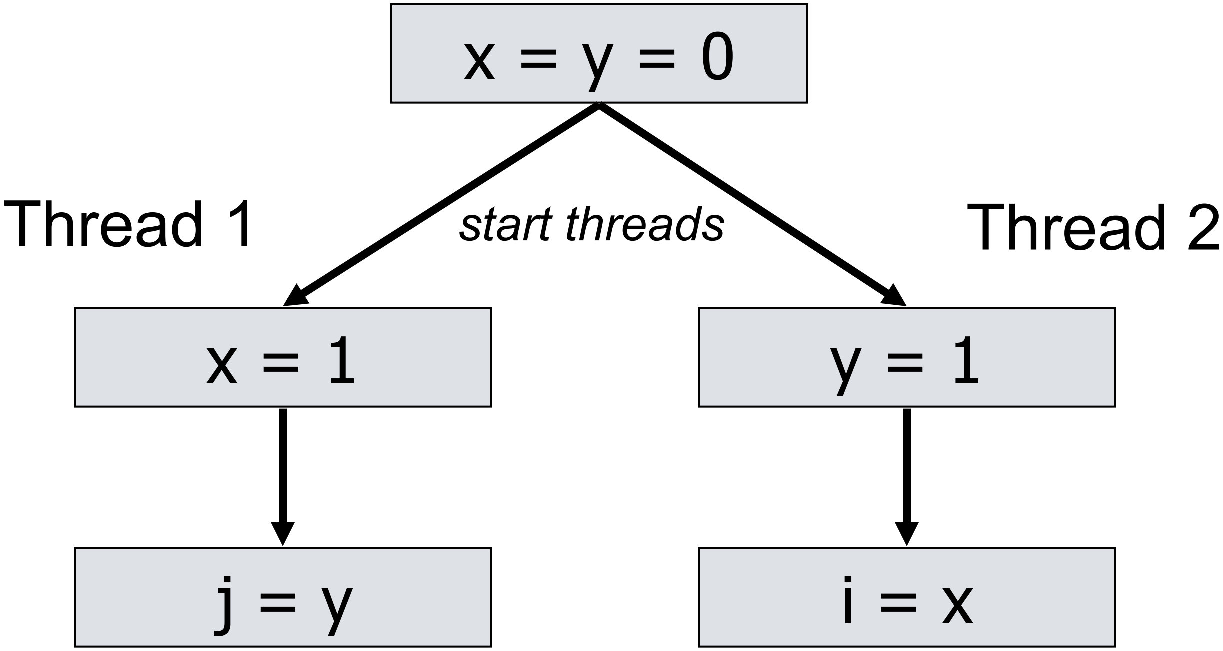
Memory Model Question

- How should the compiler and run-time system be allowed to schedule instructions?
- Possible partial answer
  - If instruction A occurs in Thread 1 before release of lock, and B occurs in Thread 2 after acquire of same lock, then A must be scheduled before B
- Does this solve the problem?
  - Too restrictive: if we prevent reordering in Thread 1,2
  - Too permissive: if arbitrary reordering in threads
  - Compromise: allow local thread reordering that would be OK for sequential programs

Instruction order and serializability

- Compilers can reorder instructions
  - If two instructions are independent, do in any order
  - Take advantage of registers, etc.
- Correctness for sequential programs
  - Observable behavior should be same as if program instructions were executed in the order written
- Sequential consistency for concurrent programs
  - If program P has no data races, then memory model should guarantee sequential consistency
  - Question: what about programs *with* races?
    - Much of complexity of memory model is for reasonable behavior for programs with races (need to test, debug, …)

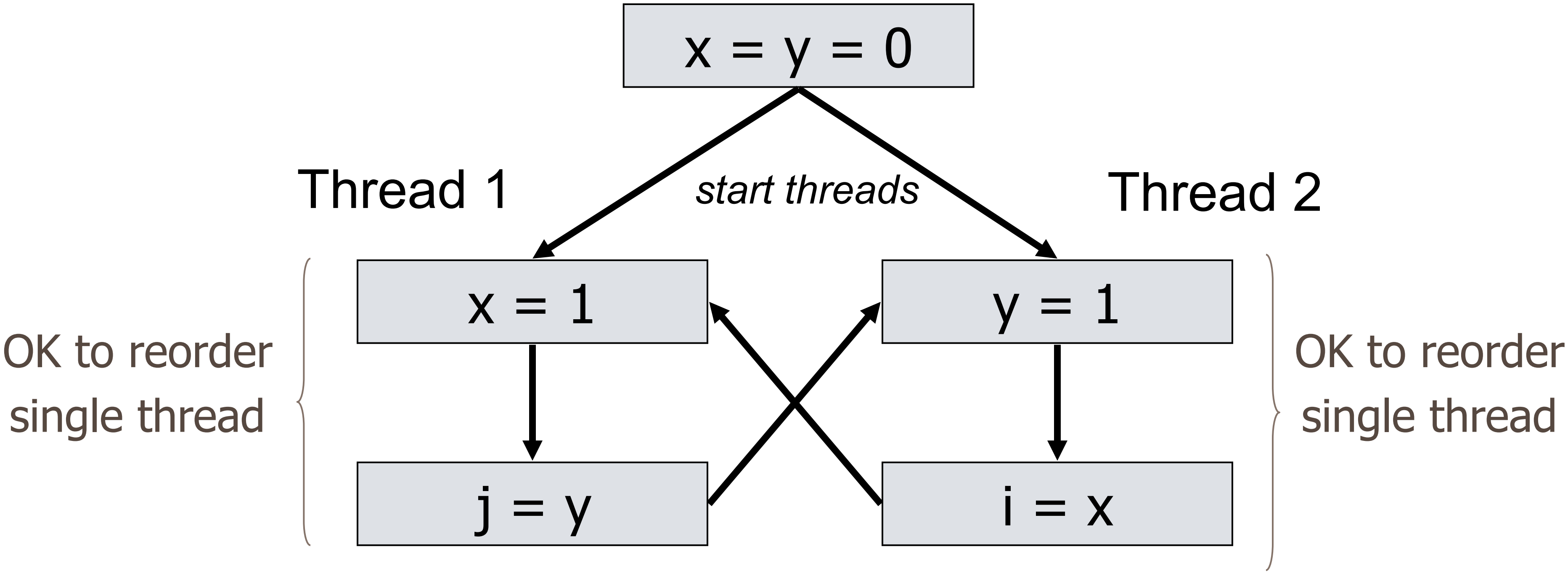
Example program with data race



Can we end up with i = 0 and j = 0?

[Manson, Pugh]

Sequential reordering + data race



How can i = 0 and j = 0?

Java definition considers this OK since there is a data race

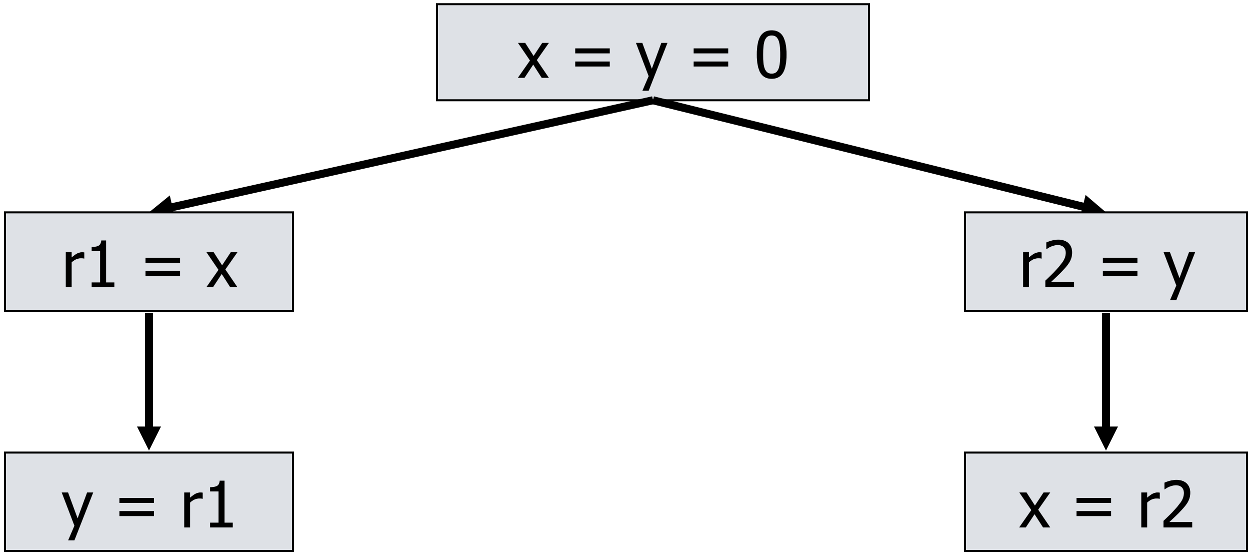
[Manson, Pugh]

Allowed sequential reordering

- “Roach motel” ordering
  - Compiler/processor can move accesses into synchronized blocks
  - Can only move them out under special circumstances, generally not observable
- Release only matters to a matching acquire
- Some special cases:
  - locks on thread local objects are a no-op
  - reentrant locks are a no-op
  - Java SE 6 (Mustang) does optimizations based on this

[Manson, Pugh]

Something to prevent …



- Must not result in r1 = r2 = 42
  - Imagine if 42 were a reference to an object!
- Value appears "out of thin air"
  - This is causality run amok
  - Legal under a simple "happens-before" model of possible behaviors

[Manson, Pugh]

Summary of memory model

- Strong guarantees for race-free programs
  - Equivalent to interleaved execution that respects synchronization actions
  - Thread reordering must preserve sequential semantics of thread
- Weaker guarantees for programs with races
  - Allows program transformation and optimization
  - No weird out-of-the-blue program results
- Form of actual memory model definition
  - Happens-before memory model (examples on next slide)
  - Additional condition: for every action that occurs, there must be identifiable cause in the program

Happens-Before orderings

- Starting a thread happens-before the run method of the thread
- The termination of a thread happens-before a join with the terminated thread
- Volatile fields
- Many util.concurrent methods set up happen-before orderings
  - placing an object into any concurrent collection happen-before the access or removal of that element from the collection

Outline

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

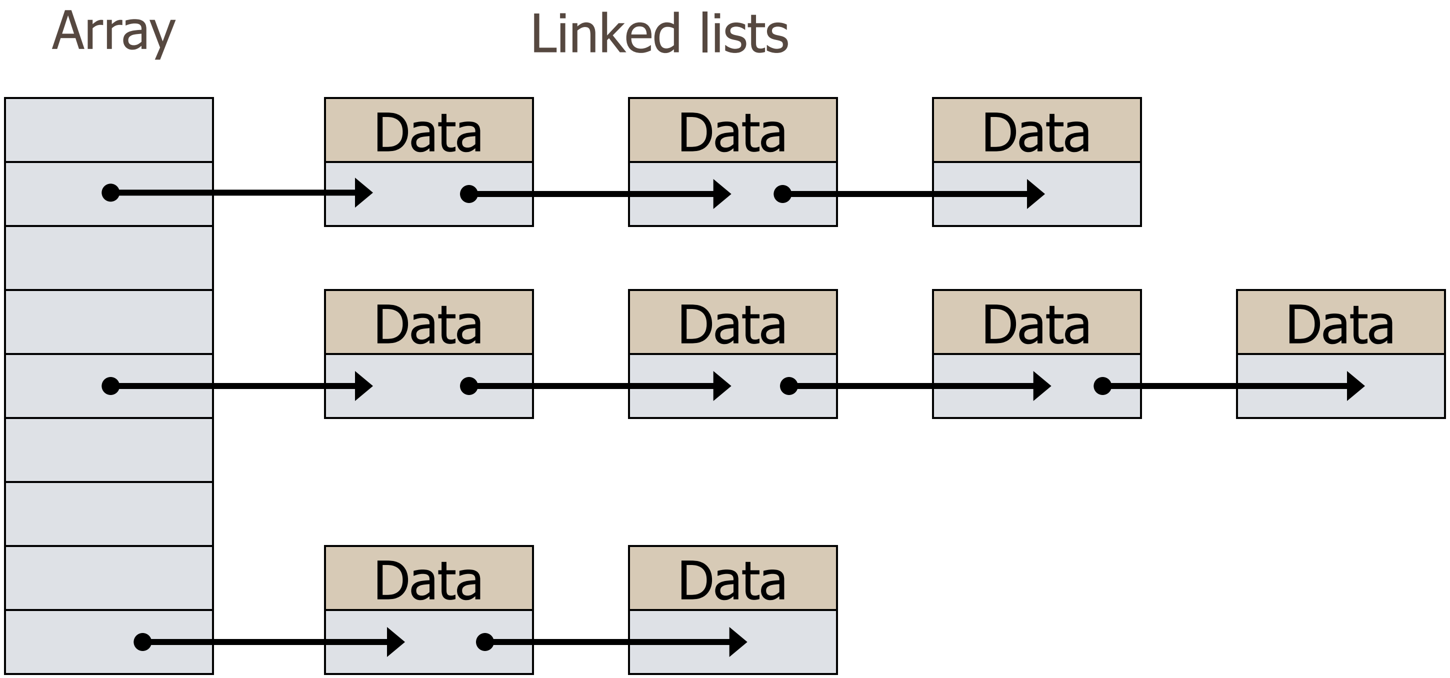
Example: Concurrent Hash Map

- Implements a hash table
  - Insert and retrieve data elements by key
  - Two items in same bucket placed in linked list
  - Allow read/write with minimal locking
- Tricky

  "ConcurrentHashMap is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the Java Memory Model (JMM) to achieve higher performance. … Use it, learn from it, enjoy it – but unless you're an expert on Java concurrency, you probably shouldn't try this on your own."

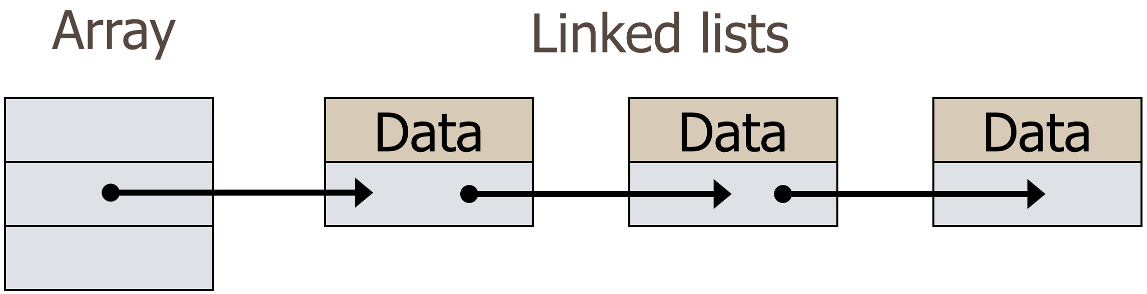
See http://www-106.ibm.com/developerworks/java/library/j-jtp08223

ConcurrentHashMap



- Concurrent operations
  - read: no problem
  - read/write: OK if different lists
  - read/write to same list: clever tricks sometimes avoid locking

ConcurrentHashMap Tricks



- Immutability
  - List cells are immutable, except for data field

    ⇒ read thread sees linked list, even if write in progress
- Add to list
  - Can cons to head of list, like Lisp lists
- Remove from list
  - Set data field to null, rebuild list to skip this cell
  - Unreachable cells eventually garbage collected

Summary

- What is concurrency?
- Basic issues in concurrency
  - Race conditions, locking, deadlock, mutual exclusion
- Simple language approaches (Past ideas)
  - Cobegin/Coend (Concurrent Pascal), Actor model
- Java Concurrency
  - Threads, synchronization, wait/notify
  - Methods for achieving thread safety
  - Java memory model
  - Concurrent hash map example

Happy Thanksgiving!

Two examples of problems in concurrent Java

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

- Background: wait and locking
  - *wait* and *notify* used within synchronized code
    - Purpose: make sure that no other thread has called method of same object
  - *wait* within synchronized code causes the thread to give up its lock and sleep until notified
    - Allow another thread to obtain lock and continue processing
- Problem
  - Calling a blocking method within a synchronized method can lead to deadlock

Nested Monitor Lockout Example

```
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

Preventing nested monitor deadlock

- Two programming suggestions
  - No blocking calls in synchronized methods, or
  - Provide some nonsynchronized method of the blocking object
- No simple solution that works for all programming situations