CS 242

Concurrency 2

John Mitchell

Reading: Chapter 15 + additional readings
Note: book presentation of memory model is obsolete

Outline

- General issues illustrated using Java
  - Thread safety
  - Nested monitor lockout problem
  - Inheritance anomaly
- Java Memory Model
  - Execution orders that virtual machine may follow
  - Example: concurrent hash map
- Beyond Java
  - Race condition detection
    - Memory model provides few guarantees for code with races
  - Atomicity

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, Goldilocks algorithms
- Atomicity and tools
  - See http://www.cs.uoregon.edu/activities/summerschool/summer06/

More detail in references than required by course

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

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

How to make classes thread-safe

- Synchronize critical sections
  - Make fields private
  - Synchronize sections that should not run concurrently
- 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;
}
```

  - Application of pure functional programming for concurrency
- Use a thread-safe wrapper
  - See next slide …

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 one that is not
  - 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

Nested monitor lockout problem (1)

- 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();
    } }
}
```

Could be blocking method of List class

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

“Inheritance anomaly”

- General idea
  - Inheritance and concurrency control do not mix well
- Ways this might occur
  - Concurrency control (synch, waiting, etc.) in derived class requires redefinitions of base class and parents
  - Modification of class requires modifications of seemingly unrelated features in parent classes
- History of inheritance anomaly
  - Identified in 1993, before Java
  - Arises in different languages, to different degrees, depending on concurrency primitives

Some forms of inher. anomaly

- Partitioning of acceptable states
  - Method can only be entered in certain states
  - New method in derived class changes set of states
  - Must redefine base class method to check new states
- History sensitiveness method entry
  - New method in derived class can only be called after other calls
  - Must modify existing methods to keep track of history

Java example (base class)

```
public class Buffer {
  protected Object[] buf;      protected int MAX;    protected int current = 0;
  Buffer(int max) {
     MAX = max;
     buf = new Object[MAX];
  }
  public synchronized Object get()  throws Exception {
    while (current<=0) { wait(); }
    current--;
    Object ret = buf[current];
    notifyAll();
    return ret;
  }
  public synchronized void put(Object v) throws Exception {
    while (current>=MAX) { wait(); }
    buf[current] = v;
    current++;
    notifyAll();
  }
}
```

Derived class: history-based protocol

```
public class HistoryBuffer extends Buffer {
  boolean afterGet = false;
  public HistoryBuffer(int max) { super(max); }

  public synchronized Object gget()  throws Exception {
    while ((current<=0)||(afterGet)) { wait(); }
    afterGet = false;
    return super.get();
  }
  public synchronized Object get()  throws Exception {
    Object o = super.get();
    afterGet = true;
    return o;
  }
  public synchronized void put(Object v) throws Exception {
    super.put(v);
    afterGet = false;
  }
}
```

New method, can be called only after get

Need to redefine to keep track of last method called

Need to redefine to keep track of last method called

Java progress: util.concurrent

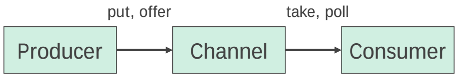
- Doug Lea's utility classes, basis for JSR 166
  - A few general-purpose interfaces
  - Implementations tested over several years
- Principal interfaces and implementations
  - Sync: acquire/release protocols
  - Channel: put/take protocols
  - Executor: executing Runnable tasks

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

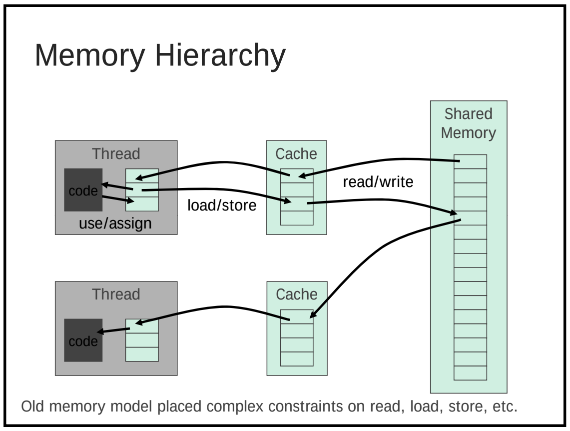
java.util.Collection

- Adapter-based scheme
  - Allow layered synchronization of collection classes
- Basic collection classes are unsynchronized
  - Example: java.util.ArrayList
  - Except for Vector and Hashtable
- Anonymous synchronized Adapter classes
  - constructed around the basic classes, e.g.,
    List l = Collections.synchronizedList(new ArrayList());

Java Memory Model

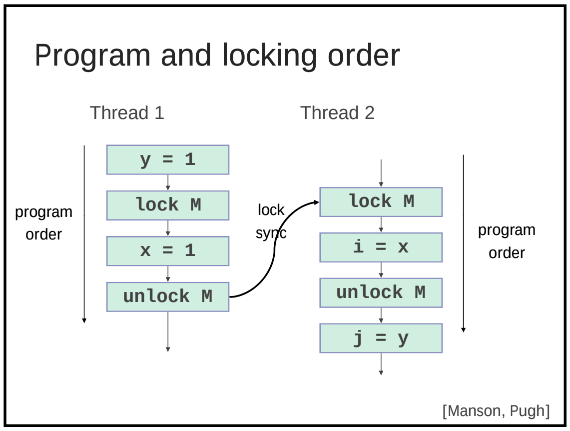
- Semantics of multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
  - Need semantics for incorrectly synchronized programs
- Determines
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs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

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



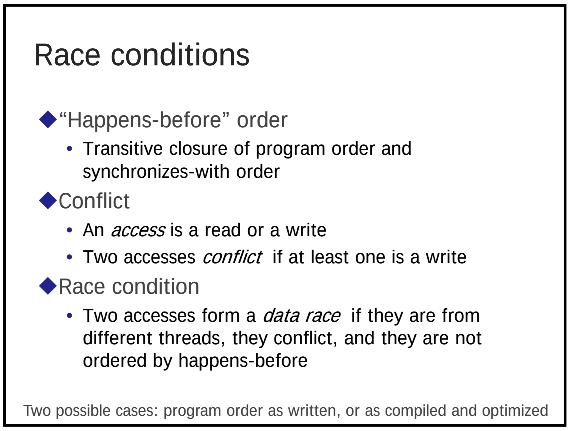
Memory Hierarchy

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



Program and locking order

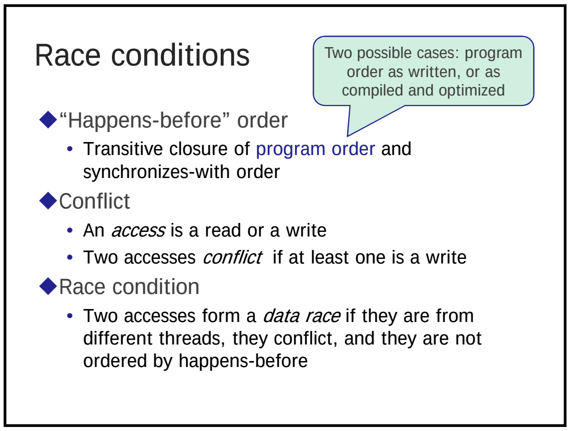
[Manson, Pugh]



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

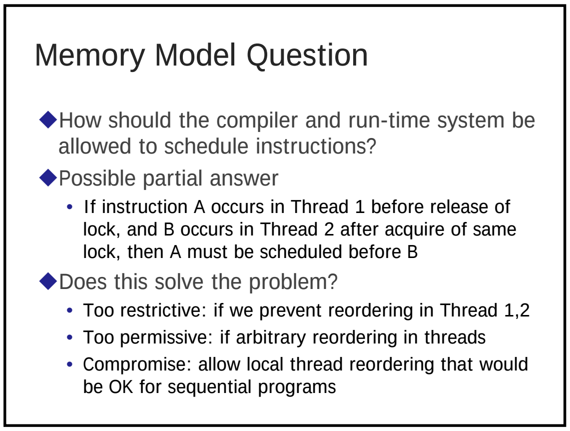
Two possible cases: program order as written, or as compiled and optimized



Race conditions

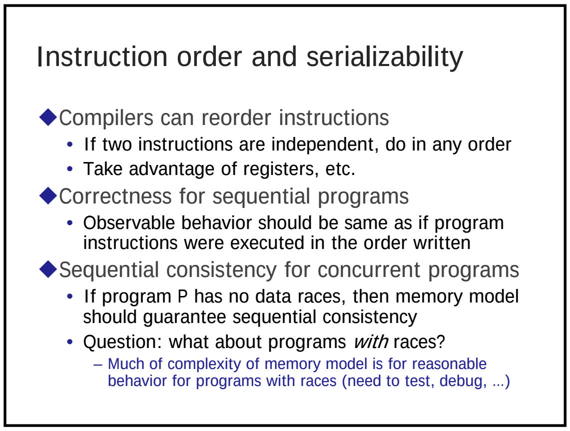
Two possible cases: program order as written, or as compiled and optimized

- ◆“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

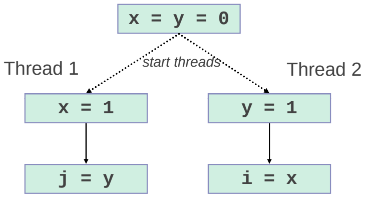
- ◆How should the compiler and run-time system be allowed to schedule instructions?
- ◆Possible partial answer
  - If instruction A occurs in Thread 1 before release of lock, and B occurs in Thread 2 after acquire of same lock, then A must be scheduled before B
- ◆Does this solve the problem?
  - Too 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


`x = y = 0`

start threads

Thread 1

`x = 1`

`j = y`

Thread 2

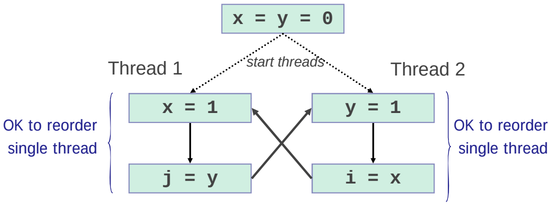
`y = 1`

`i = x`

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

[Manson, Pugh]

Sequential reordering + data race


`x = y = 0`

start threads

Thread 1

`x = 1`

`j = y`

Thread 2

`y = 1`

`i = x`

OK to reorder single thread

OK to reorder single thread

How can i = 0 and j = 0?

Java definition considers this OK since there is a data race

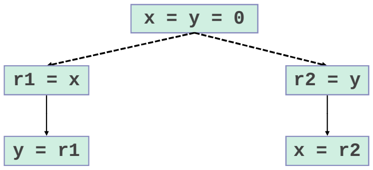
[Manson, Pugh]

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 …


`x = y = 0`

`r1 = x`

`y = r1`

`r2 = y`

`x = r2`

- 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
  - Additional condition: for every action that occurs, there must be identifiable cause in the program

Volatile fields

- If two accesses to a field conflict:
  - use synchronization to prevent race, or
  - make the field volatile
    - serves as documentation
    - gives essential JVM machine guarantees
- Consequences of volatile
  - reads and writes go directly to memory (not registers)
  - volatile longs and doubles are atomic
    - not true for non-volatile longs and doubles
  - volatile reads/writes cannot be reordered
    - reads/writes become acquire/release pairs

Volatile happens-before edges

- A volatile write happens-before all following reads of the same variable
  - A volatile write is similar to a unlock or monitor exit (for determining happens-before relation)
  - A volatile read is similar to a lock or monitor enter
- Volatile guarantees visibility
  - Volatile write is visible to happens-after reads
- Volatile guarantees ordering
  - Happens-before also constrains scheduling of other thread actions

Example (Manson, Pugh)

- `stop` *must* be declared volatile
  - Otherwise, compiler could keep in register

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
      try { Thread.sleep(100); } …;
  }
  private void oneStep() { /*...*/ }
}
```

Additional properties of volatile

- Incrementing a volatile is not atomic
  - if threads threads try to increment a volatile at the same time, an update might get lost
- volatile reads are very cheap
  - volatile writes cheaper than synchronization
- No way to make elements of an array be volatile
- Consider using util.concurrent.atomic package
  - Atomic objects work like volatile fields but support atomic operations such as increment and compare and swap

[Manson, Pugh]

Other 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
- 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

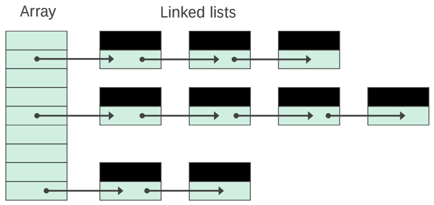
Example: Concurrent Hash Map

- 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."

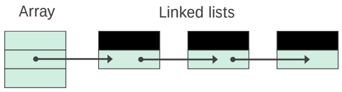
See http://www-106.ibm.com/developerworks/java/library/j-jtp08223

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

More info: see homework

Races in action

- Power outage in northeastern grid in 2003
- Affected millions of people
- Race in Alarm and Event Processing code
- “We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it.”-- GE Energy's Mike Unum

Race condition detection

- Weak Java memory model guarantees for races
  - If a program contains a data race, program behavior may be unintuitive, hard to test and debug
- Use language restriction, tools to identify races
  - Type-Based Race Prevention
    - Languages that cannot express “racy” programs
  - Dynamic Race Detectors
    - Using instrumented code to detect races
  - Model-Checkers
    - Searching for reachable race states

Type-Based Race Prevention

- Method
  - Encode locking discipline into language.
  - Relate shared state and the locks that protect them.
  - Use typing annotations.
  - Recall ownership types; this will seem familiar

Example: Race-Free Cyclone

- “This lock protects this variable”

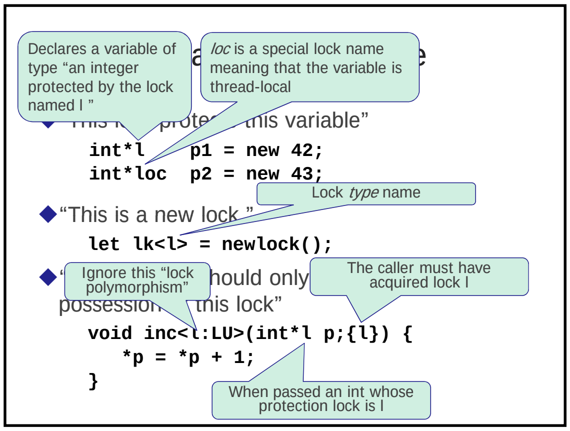
```
int*l    p1 = new 42;
int*loc  p2 = new 43;
```

- “This is a new lock.”

```
let lk<l> = newlock();
```

- “This function should only be called when in possession of this lock”

```
void inc<l:LU>(int*l p;{l}) {
    *p = *p + 1;
}
```



Declares a variable of type “an integer protected by the lock named l ”

loc is a special lock name meaning that the variable is thread-local

this variable”

```
int*l    p1 = new 42;
int*loc  p2 = new 43;
```

Lock *type* name

“This is a new lock.”

```
let lk<l> = newlock();
```

Ignore this “lock polymorphism”

hould only

The caller must have acquired lock l

possession

this lock”

```
void inc<l:LU>(int*l p;{l}) {
    *p = *p + 1;
}
```

When passed an int whose protection lock is l

Type-Based Race Prevention

- Positives
  - Soundness: Programs are race-free by construction
  - Familiarity: Locking discipline is a common paradigm
  - Relatively Expressive
    - Classes can be parameterized by different locks
    - Types can often be inferred
- Negatives:
  - Restrictive: Not all race-free programs are legal
    - Other synchronization? (wait/notify, etc.)
  - Annotation Burden: Lots of annotations to write

Dynamic Race Detectors

- Find race conditions by
  - Instrument source code / byte code
  - Run lockset and happens-before analyses
  - Report races detected *for that run*
- No guarantee that all races are found
  - Different program input may lead to different execution paths

Basic Lockset Analysis

- Monitor program execution
- Maintain set of locks held at each program point
  - When lock is acquired, add to the set of current locks
  - Remove lock from lockset when it is released
- Check variable access
  - The first time a variable is accessed, set its “candidate set” to be the set of held locks
  - The next time variable is accessed, take the intersection of the candidate set and the set of currently held lock
- If intersection empty, flag potential race condition

Happens-Before Analysis

- Maintain representation of happens-before as program executes
  - Can be done using “local clocks” and synchronization
- Check for races
  - When a variable access occurs that happens-for does not guarantee is ‘after’ the previous one, we have detected an actual race

Can combine lockset, happens-before

- Lockset analysis detects violation of locking discipline
  - False positives if strict locking discipline is not followed
- Happens-Before reports actual race conditions
  - No false positives, but false negatives can occur
  - High memory and CPU overhead
- Combined use
  - Use lockset, then switch to happens-before for variables where a race is detected

Goldilocks algorithm [FATES/RV ’06]

- Lockset-based characterication of the happens-before relation
  - Similar efficiency to other lockset algorithms
  - Similar precision to vector-clocks
  - Locksets contain locks, volatile variables, thread ids
- Theorem
  - When thread t accesses variable d, there is no race iff lockset of d at that point contains t
- Sound: Detects races that occur in execution
  - Race reported → Two accesses not ordered by happens-before

Atomicity

◆ Concept
- Mark block so that compiler and run-time system will execute block without interaction from other threads

◆ Advantages
- Stronger property than race freedom
- Enables sequential reasoning
- Simple, powerful correctness property

Next slides: Cormac Flanagan

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
    }
    synchronized (this) {
      i = t+1;
    }
  }
  ...
}
```

Ref.inc()
- ◆ race-free
- ◆ behaves incorrectly in a multithreaded context

Race freedom *does not* prevent errors due to unexpected interactions between threads

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }

  void read() { return i; }
  ...
}
```

Ref.read()
- ◆ has a race condition
- ◆ behaves correctly in a multithreaded context

Race freedom *is not necessary* to prevent errors due to unexpected interactions between threads

Atomic

An easier-to-use and harder-to-implement primitive:

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
lock acquire/release

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
(behave as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)

AtomJava [Grossman]

Novel prototype recently completed …

◆ Source-to-source translation for Java
- Run on any JVM (so parallel)
- At VM's mercy for low-level optimizations

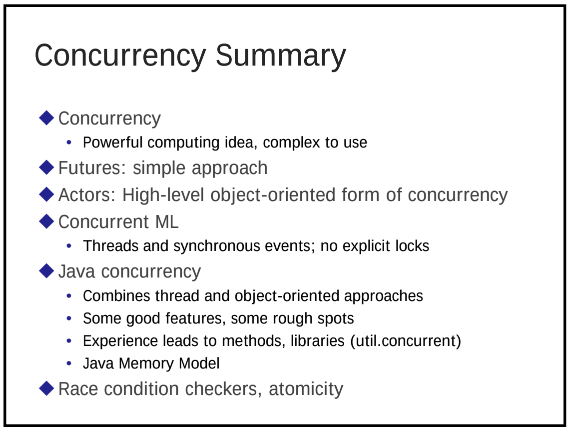
◆ Atomicity via locking (object ownership)
- Poll for contention and rollback
- No support for parallel readers yet ☹

Hope: whole-program optimization can get "strong for near the price of weak"

Implementing atomic

◆ Key pieces:
- Execution of an atomic block logs writes
- If scheduler pre-empts a thread in atomic, rollback the thread
- Duplicate code so non-atomic code is not slowed by logging
- Smooth interaction with GC

[Grossman]



Concurrency Summary

- Concurrency
  - Powerful computing idea, complex to use
- Futures: simple approach
- Actors: High-level object-oriented form of concurrency
- Concurrent ML
  - Threads and synchronous events; no explicit locks
- Java concurrency
  - Combines thread and object-oriented approaches
  - Some good features, some rough spots
  - Experience leads to methods, libraries (util.concurrent)
  - Java Memory Model
- Race condition checkers, atomicity