Lecture 7b

Atomizer: A Dynamic Atomicity Checker
public atomic StringBuffer {
  private int count guarded_by this;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  public synchronized void append(StringBuffer sb) {
    int len = sb.length();
    ... 
    ... 
    sb.getChars(..., len, ...);
    ... 
  }
}

append(...) is not atomic

sb.length() acquires the lock on sb, gets the length, and releases lock
other threads can change sb
use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)
The Atomizer

• Annotate program with atomicity requirements
  
  /*# atomic */ synchronized StringBuffer append(...) ...  
  /*# atomic */ synchronized int length() ...  
  /*# atomic */ synchronized int getChars(...) ...  

• Atomizer checks for violations at run time

• Core analysis
  
  - Reduction [Lipton 76]  
  - Eraser [Savage et al. 97]

• Applied to 150,000+ lines of Java code
Atomizer: Instrumentation Architecture

Atomizer

Instrumented Source Code

/**# atomic */
void append(...)
{ ... }

javac +JVM

runtime
• Lockset
• Reduction

event stream
T1: begin_atomic
T2: acquire(lock3)
T2: read(x,5)
T1: write(y,3)
T1: end_atomic
T2: release(lock3)

Warning: method "append" may not be atomic at line 43

27 Jan 2004
Lockset Algorithm

• Tracks lockset for each field
  - lockset = set of locks held on all accesses
  - empty lockset indicates possible race condition

• Algorithm:
  - On first access to o.f:
    ▪ LockSet(o.f) = Held(curThread)
  - On all subsequent accesses to o.f:
    ▪ LockSet(o.f) = LockSet(o.f) \cap Held(curThread)
Lockset Example

Thread 1

synchronized(x) {
    o.f = 3;
}

Thread 2

synchronized(y) {
    o.f = 2;
}

• LockSet(o.f) = {x}
Lockset Example

Thread 1
synchronized(x) {
    o.f = 3;
}

Thread 2
synchronized(y) {
    o.f = 2;
}

• LockSet(o.f) = \{ x \} \cap \{ y \} = \{ \}
Problems

• This doesn’t quite work

• We need to deal with
  - Uninitialized data
  - Read-Shared Data
Uninitialized Data

- Data often initialized by one owner
- No need to lock at this time
- How do we know when initialization is done?
  - Answer: We don’t
  - But, we can tell when the value is accessed by a second thread
Lockset

any thread
r/w

Shared-read/write
Track lockset
Extending Lockset (Thread Local Data)

- Shared-read/write Track lockset
- Thread Local
- first thread r/w
- second thread r/w
- any thread r/w
Read Shared

- Once created, some data is only read
- No need to lock read-only data
Extending Lockset (Read Shared Data)

- First thread: Read/Write
- Thread Local
- Second thread: Read
- Second thread read
- Any thread: Write
- Shared-read/write
- Track lockset
- Any thread: Read/Write
- Read Shared
- Any thread: Read

27 Jan 2004
Reduction [Lipton 75]

\[ \text{acq(this)} \quad X \quad j=\text{bal} \quad Y \quad \text{bal}=j+n \quad Z \quad \text{rel(this)} \]

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \]
Reduction [Lipton 75]
Reduction [Lipton 75]

\[ \text{acq(this)} \quad X \quad j=\text{bal} \quad Y \quad \text{bal}=j+n \quad Z \quad \text{rel(this)} \]

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \quad S_7 \]

27 Jan 2004
Reduction [Lipton 75]

\[
\begin{align*}
S_0 & \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \\
X & \quad j=\text{bal} \quad Y & \quad \text{bal}=j+n & \quad Z & \quad \text{rel(this)}
\end{align*}
\]
**Movers**

- **R**: right-mover
  - lock acquire
- **L**: left-mover
  - lock release
- **B**: both-mover
  - race-free field access
- **N**: non-mover
  - access to "racy" fields

• Reducible blocks have form \((R|B)^* [N] (L|B)^*\)
Dynamic Reduction

- Automata to check \((R|B)^* [N] (L|B)^*\)
public class StringBuffer {
    private int count;
    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
    /*# atomic */
    public synchronized void append(StringBuffer sb){
        int len = sb.length();
        ...
        ...
        ...
        sb.getChars(...,len,...);
        ...
    }
}
Summary

- Race conditions are a *heuristic* for detecting errors
- Atomicity is a deeper, more semantic property
  - sequential reasoning ok on atomic methods
- Can verify atomicity
  - statically (we saw a type system for this)
  - dynamically (by inspecting an event trace)