How to avoid concurrency defects with static and dynamic analysis

Ben Chelf
CTO
Coverity, Inc.
Agenda

- Increasing Complexity in Software Development
- Common Concurrency Defects
- Using Static Analysis to detect Race Conditions
- Using Dynamic Analysis to detect concurrency problems
Exploding complexity

TODAY’S PREMIUM AUTOMOBILE
10 MILLION LINES OF CODE

Diagram of Onboard Network of a Premium Car
(Source: Manfred Boy, ICSE'06)

WINDOWS VISTA
50 MILLION LINES OF CODE

Windows OS Code Growth - MLOC
(Source: Wikipedia)

<table>
<thead>
<tr>
<th>Year</th>
<th>Code (MLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT 3.1 ('93)</td>
<td>10</td>
</tr>
<tr>
<td>NT 3.5 ('94)</td>
<td>15</td>
</tr>
<tr>
<td>NT 4.0 ('95)</td>
<td>20</td>
</tr>
<tr>
<td>2000</td>
<td>30</td>
</tr>
<tr>
<td>XP ('01)</td>
<td>40</td>
</tr>
<tr>
<td>Vista ('05)</td>
<td>50</td>
</tr>
</tbody>
</table>
Mulit-threaded Development

- New requirement for developers that need to:
  - Take full advantage of multi-core hardware capabilities
  - Remain competitive

- Challenges of multi-threaded development
  - New discipline: developers may be unprepared
  - Many migration challenges from single-threaded development
  - Testing complexity increased exponentially with code size

“The number of cores per processor chip will double every 18 to 24 months”  Gartner
A New Class of Code Complexity

```java
void get_input() {
    int x = input_from_user();
    if (x < 100) {
        printf("x is less than 100!\n");
    } else {
        printf("x is at least 100 or greater!\n");
    }
}
```
Single-threaded Complexity

4 different execution possibilities

x < 100

x >= 100
Multi-threaded Complexity

\[ x < 100 \quad x \geq 100 \]
Defect Type: Deadlock

```
public class Deadlock {
    static Object a;
    static Object b;

    public static void lock1() {
        synchronized(a) {
            ...
            synchronized(b) {
                ...
            }
        }
    }

    public static void lock2() {
        synchronized(b) {
            ...
            synchronized(a) {
                ...
            }
        }
    }
}
```

**Deadlock**: Two or more threads wait for locks in a circular chain such that the locks can never be acquired.
Defect Type: Thread Block

Thread Block: A thread calls a long-running operation while holding a lock, preventing the progress of other threads.

```c
/* drivers/net/pmcmba/wavelan_cs.c */
spin_lock_irqsave(&lp->lock, flags); /* 1889 */
switch(cmd)
...
    case SIOCGIWPRIV: /* 2304 */
        if(copy_to_user(wrq->u.data(pointer, ...)) /* 2305 */
```
Defect Type: Thread Block

A

extern int my_balance;
extern lock bal_lock;

void thread_one() {
    lock(bal_lock);
    my_balance += 100;
    unlock(bal_lock);
}

B

extern int my_balance;
extern lock bal_lock;

void thread_two() {
    my_balance += 200;
}

JOINT BANK ACCOUNT: Balance $782

PERSON A ➔ Deposit $100 ➔ LOCK ➔ Add $100 ➔ $882 ➔ Balance $882
(A OVERWRITES B DEPOSIT)

PERSON B ➔ Deposit $200 ➔ Add $200 ➔ $982

Desired sum
Balance $1082
Finding Concurrency Errors

- **Debuggers**
  - Must wait for problem to occur to debug it
  - Deadlock and Race not easy to reproduce

- **Profilers**
  - Heavy burden in terms of memory and execution time
  - Rely on a particular timing of events to reproduce problem

- **Dynamic and Static Source Code Analysis**
  - Effective for some classes of concurrency defects
Static Race Condition Methodology

Combination of *inter-procedural* and *statistical* analysis

- **Inter-procedural analysis**
  - Automatically determines lock/variable pairings

- **Statistical analysis**
  - Automatically determines which functions acquire which locks
  - Analyzes every function for *unprotected* variables

- **New interface**
  - Specifically designed to present race conditions in easy to diagnose format
Problem: What are the rules (code behavior) that must be true in a software system?

- 100s-1000s of rules in 100s-1000s of subsystems
- To check you must answer questions including
  - Must a() follow b()?
  - Can foo() fail?
  - Does bar(p) free p?
  - Does lock l protect x?

Advice: Finding rules through manual inspection of code is difficult
- Don’t try it!

Proposed Methodology:
- Infer correct behavior of code through analysis, then check for contradictions
Unguarded Access

The variable below was accessed without holding a guarding lock. In a multi-threaded environment, this can lead to a race condition. Prevent inferred that a lock was required to access this variable based on the examples shown below:

```
private final void run()
{
    Packet packet;

    while ((isRunning) || (queue.isEmpty()))
    {
        synchronized(queue)
    }
```

Guarded Access

Field `nu.fw.jeti.backend.Output.queue` guarded by lock `nu.fw.jeti.backend.Output.queue`

```
public synchronized void setAuthenticated()
```

Guarded Access

Field `nu.fw.jeti.backend.Output.queue` guarded by lock `nu.fw.jeti.backend.Output.queue`

```
if (isRunning)
{
    synchronized(queue)
}
```

Guarded Access

Field `nu.fw.jeti.backend.Output.queue` guarded by lock `nu.fw.jeti.backend.Output.queue`

```
queue.addLast(p);
queue.notifyAll();
```
Dynamic Analysis

- Dynamically instruments locks and variables of interest
- Detection algorithms for race and deadlocks
  - Detects location of potential errors
- Tracks all locks created during execution
- Uses lockset-like algorithm
  - State machine
display.setValue(counter);
}

void increment() {
    counter:  //read[v]
    int val = counter;
    val++;
    //write[v+1]
    display.setValue(counter);
}
Defects Identified in Defect pane during execution

<table>
<thead>
<tr>
<th>ID</th>
<th>Detector</th>
<th>File</th>
<th>Line</th>
<th>Owner Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>84</td>
<td>garden.Counter</td>
</tr>
<tr>
<td>2</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>85</td>
<td>garden.Counter</td>
</tr>
<tr>
<td>3</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>82</td>
<td>garden.Counter</td>
</tr>
</tbody>
</table>

Java Stack Trace Console

[ERROR] [Thread-EAST] Possible race condition on read: no lock consistently protects field garden.Counter
locks held by current thread: {}
The previous thread's state at the time of this access is: Runnable

stack trace:
  at garden.Counter.increment(Garden.java:82)
  at garden.Turnstile.run(Garden.java:115)
**Standard Java Stack Trace delivered to Console**

```java
void increment() {
    int temp = counter;  //read[v]
    Simulate.HWinterrupt();
    counter = temp + 1;  //write[v+1]
    display.setValue(counter);
}
```

缺陷列表:

<table>
<thead>
<tr>
<th>ID</th>
<th>运行时条件</th>
<th>文件</th>
<th>方法</th>
<th>行</th>
<th>类</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>increment</td>
<td>84</td>
<td>garden.Counter</td>
</tr>
<tr>
<td>2</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>increment</td>
<td>85</td>
<td>garden.Counter</td>
</tr>
<tr>
<td>3</td>
<td>RACE_CONDITION</td>
<td>Garden.java</td>
<td>increment</td>
<td>82</td>
<td>garden.Counter</td>
</tr>
</tbody>
</table>

控制台日志:

```
[ERROR] [Thread-EAST] Possible race condition on read: no lock consistently protects field garden.Counter.previous locks used to protect field: {} locks held by current thread: {} The previous thread's state at the time of this access is: RUNNABLE

stack trace:
at garden.Counter.increment(Garden.java:82)
at garden.Turnstile.run(Garden.java:115)
```
```
   display.setValue(counter);

   void increment() {
       int temp = counter;  //read[v]
       Simulate.HWInterrupt();
       counter = temp + 1;  //write[v+1]
       display.setValue(counter);
   }
```

Hot Links in Stack trace back into code:

```
[ERROR] [Thread-EAST] Possible race condition on read: no lock consistently protects field garden.Counter
previous locks used to protect field: ()
locks held by current thread: ()
The previous thread's state at the time of this access is: RUNNABLE
stack trace:
at garden.Counter.increment(Garden.java:82)
at garden.Turnstile.run(Garden.java:115)
```
Deadlock detected before actual deadlock occurred.
The combination of Static and Dynamic analysis can be more effective than either tool alone

Static Analysis can use information gathered during a dynamic run

- Static can use the dynamic results to aid its inference of which locks are associated with which variables

Dynamic analysis can use the results of static analysis to reduce the overhead of instrumenting code

- Java ‘Final’ fields detected are omitted from instrumentation since they cannot have races on them
Conclusion

- Multi-thread application development is more complex
- Existing techniques for debugging multi-threaded applications are not sufficient
- Static and Dynamic analysis enable high quality multi-threaded application development
- Coverity Prevent and Thread Analyzer are industry leading static and dynamic analysis products
Thanks for your attention!