Deadlock Bug Detection Techniques

Prof. Moonzoo Kim
CS KAIST
Bug Detection Techniques for Concurrent Programs

Model checking techniques
+ High precision
+ Comprehensive error detection
- Scalability (state explosion problem)
- Verification expertise is required

Testing techniques
+ High precision
+ Friendly to developers
- Difficult to generate test cases and thread schedules

Bug detection techniques
+ Fast and convenient (no need to generate many executions)
- False alarms

Verifications
- Precision
- False alarm

100~1,000 LOC
Scalability
1,000,000 LOC <

- SPIN
- jCute
- Fusion
- Java Pathfinder
- CHESS
- KISS
- CalFuzzer
- ConTest
- rtest
- Atomizer
- Eraser
- RacerX
- MetaL

CS492B Analysis of Concurrent Programs, Prof. Moonzoo Kim
Deadlock Bugs Frequently Occur in Real World

<table>
<thead>
<tr>
<th>Application</th>
<th>What it does</th>
<th>Non-Deadlock</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>Database Server</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Apache</td>
<td>Web Server</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Web Browser</td>
<td>41</td>
<td>16</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>Office Suite</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>74</strong></td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

- In a survey on 105 real-world concurrency bugs in open-source applications, **31 out of 105 bugs are deadlock bugs** [Lu et al., ASPLOS 08]
Deadlock Bugs Frequently Occur in Real World

- According to Apache bug tracking systems, there have been 200 deadlock related issues since 2014

<table>
<thead>
<tr>
<th>Key</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPID-5344</td>
<td>[AMQP 1.0 JMS] Thread deadlock related to session creation.</td>
</tr>
<tr>
<td>QPID-5439</td>
<td>[AMQP 1.0 JMS client] Client hangs during connection setup.</td>
</tr>
<tr>
<td>QPID-5294</td>
<td>[AMQP 1.0 JMS] Thread deadlock due to JVM bug JDI.</td>
</tr>
<tr>
<td>SOLR-5615</td>
<td>Deadlock while trying to recover after a ZK session expiration.</td>
</tr>
<tr>
<td>MESOS-930</td>
<td>Provide slave&lt;-&gt;executor protocol.</td>
</tr>
<tr>
<td>FELIX-4384</td>
<td>Difference between inner class and normal class service methods.</td>
</tr>
<tr>
<td>HBASE-10452</td>
<td>Fix potential bugs in exception handlers.</td>
</tr>
<tr>
<td>DRILL-333</td>
<td>Throw exception when trying to send message within a transaction.</td>
</tr>
<tr>
<td>LUCENE 5420</td>
<td>Long text matching email local part rule in HAY045Li.</td>
</tr>
</tbody>
</table>
Deadlock

- A deadlock occurs when each of a set of threads is blocked, waiting for another thread in the set to satisfy a certain condition. To resolve the deadlock:
  - Release shared resource
  - Raise event
Resource Deadlock

• Ex. Dining philosopher problem

[Ex. Dining philosopher problem diagram]

1. Think
2. If left fork is available, pick it up
3. If right fork is available, pick it up
4. Eat
5. Put the right folk down
6. Put the left folk down
Resource Deadlock in Concurrent Programs

- ABBA deadlock

Thread1() {
    lock(X)
    x = ...
    lock(Y)
    y = ...
    unlock(Y)
    unlock(X)
}

Thread2() {
    lock(Y)
    y = ...
    lock(X)
    x = ...
    unlock(X)
    unlock(Y)
}
Non-blocking Algorithm

• An algorithm is called **non-blocking** if failure or suspension of any thread cannot cause failure or suspension of another thread
  – a non-blocking algorithm is lock-free if there is guaranteed system-wide progress, and wait-free if there is also guaranteed per-thread progress.

• Blocking a thread is undesirable for many reasons while non-blocking algorithms do not suffer from these downsides
  – while the thread is blocked, it cannot accomplish anything
  – certain interactions between locks can lead to error conditions such as deadlock, livelock, and priority inversion.
  – using locks involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead and is more prone to bugs.
Communication Deadlock

• Lost notify

Thread1() {
1: ...
2: for(i=0;i<10;i++){
3: wait(m);}
}

Thread2() {
11: ...
12: for(j=0;j<10;j++){
13: notify(m);}
}

\[t_1: \text{Thread 1}\]
3: wait(m) //i==0
... 
3: wait(m) //i==9

\[t_2: \text{Thread 2}\]
13: notify(m) //j==0
... 
13: notify(m) //j==9 (terminate)
public final void wait() {

• Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

• The current thread must own this object's monitor.
  – The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

• Interrupts and spurious wakeups are possible, and this method should always be used in a loop:

  synchronized (obj) {
    while (<condition does not hold>)
      obj.wait();
    ... // Perform action appropriate to condition
  }

See the following stackoverflow discussion: http://stackoverflow.com/questions/1050592/do-spurious-wakeups-actually-happen
public final void notify()

• Wakes up a single thread that is waiting on this object's monitor.
  – If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

• The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  – The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

• This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:
  – By executing a synchronized instance method of that object.
  – By executing the body of a synchronized statement that synchronizes on the object.
  – For objects of type Class, by executing a synchronized static method of that class.
Finding Deadlock Bugs is Difficult

• A deadlock bug induces deadlock situations only under certain thread schedules

• Systems software creates a massive number of locks for fine-grained concurrency controls

• Function caller-callee relation complicates the reasoning about possible nested lockings
Bug Detection Approach

Resource deadlock
• Basic potential deadlock detection algorithm
• GoodLock algorithm

Communication deadlock
• CHECKMATE: a trace program model-checking technique for deadlock detection
Basic Potential Deadlock Detection

• Extend the cyclic deadlock monitoring algorithm

• Cyclic deadlock monitoring algorithm (e.g. LockDep)
  – Monitor lock acquires and releases in runtime
  – Lock graph \((N, E_N)\)
    • Create a node \(n_X\) when a thread acquires lock \(X\)
    • Create an edge \((n_X, n_Y)\) when a thread acquires lock \(Y\)
      while holding lock \(X\)
    • Remove \(n_X, (n_X, *)\) and \((*, n_X)\) when a thread releases \(X\)

\(\rightarrow\) Report deadlock when the graph has any cycle
Cyclic Deadlock Detection Example (1/2)

Thread1() { Thread2() {
    1: lock(X) 11: lock(Y)
    2: a = ... ; 12: b = ... ;
    3: lock(Y) 13: lock(X)
    4: b = ... ; 14: a = ... ;
    5: unlock(Y) 15: unlock(X)
    6: unlock(X) 16: unlock(Y)
}
}

Deadlock detected!
Cyclic Deadlock Detection Example (2/2)

Thread1() {
    lock(X);
    a = ...;
    lock(Y);
    b = ...;
    unlock(Y);
    unlock(X);
}

Thread2() {
    lock(Y);
    b = ...;
    lock(X);
    a = ...;
    unlock(X);
    unlock(Y);
}

No problem
Basic Deadlock Prediction Technique

• Potential cyclic deadlock detection algorithm [Harrow, SPIN 00]
  – Lock graph \((N, E_N)\)
    • Create a node \(n_X\) when a thread acquires lock \(X\)
    • Create an edge \((n_X, n_Y)\) when a thread acquires lock \(Y\)
      while holding lock \(X\)
    • Remove \(n_X\), \((n_X, *)\) and \((*, n_X)\) when a thread releases \(X\)
  \(\rightarrow\) Report potential deadlocks if the resulted graph at the end of an execution has a cycle

[Harrow, SPIN 00] J. J. Harrow, Jr.: Runtime checking of multithreaded applications with Visual Threads, SPIN Workshop 2000
Potential Cyclic Deadlock Detection Example

Thread1() {
1: lock(X)
2: a = ...
3: lock(Y)
4: b = ...
5: unlock(Y)
6: unlock(X)
}

Thread2() {
11: lock(Y)
12: b = ...
13: lock(X)
14: a = ...
15: unlock(X)
16: unlock(Y)
}

Cycle \( \rightarrow \) Potential deadlock
Basic Deadlock Prediction Technique

• The algorithm is commercialized as a SW tool VisualThreads (HP)

• Empirical results show that the algorithm is very effective to discover hidden deadlock bugs

• Challenge: generate many false positive
False Positive Example#1 – Single Thread Cycle

Thread1() { Thread2() {
1: lock(X); 11: lock(X);
2: lock(Y); 12: unlock(X);
3: unlock(Y); 13: lock(Y);
4: unlock(X); 14: unlock(Y); }
5: lock(Y); 2
6: lock(X); 5
7: unlock(X); The lock graph has a cycle, but no deadlock
8: unlock(Y); }

A cycle that consists of edges created by one thread is a false positive
False Positive Example #2: Gate Lock

Thread1() {
1: lock(X);
2: lock(Y);
3: lock(Z);
4: unlock(Z);
5: unlock(Y);
6: unlock(X);
}

Thread2() {
11: lock(X);
12: lock(Z);
13: lock(Y);
14: unlock(Y);
15: unlock(Z);
16: unlock(X);
}

Cycle, but no deadlock
False Positive Example#3: Thread Creation

main(){
    start(f1);
}

f1(){
    lock(X);
    lock(Y);
    unlock(Y);
    unlock(X);
    start(f2);
}

f2(){
    lock(Y);
    lock(X);
    unlock(X);
    unlock(Y);
}

Thread segment#1

Thread segment#2

Cycle, but no deadlock
GoodLock Algorithm [Agarwal, IBM 10]

- Extend the lock graph in the basic potential deadlock detection algorithm to consider thread, gate lock, and thread segment

- Thread segment graph \((S, E_S)\)
  - When the main thread \(t_0\) starts:
    - Create a thread segment node \(s_0\);
    - map \(t_0\) to \(s_0\) \((M(t_0) = s_0)\);
    - \(\text{n} = 1\).
  - When a thread \(t_i\) starts a new thread \(t_j\)
    - Create two thread segment nodes \(s_n\) and \(s_{n+1}\);
    - Create two edges \((M(t_i), s_n)\) and \((M(t_i), s_{n+1})\);
    - \(M(t_i) = s_n\); \(M(t_j) = s_{n+1}\);
    - \(\text{n} = n + 2\);

Thread Segment Graph Example

```
f1()
1: lock(X);
2: lock(Y);
3: start(f2);
4: unlock(Y);
5: unlock(X);
}
f2()
11: lock(Y);
12: lock(X);
13: unlock(X);
14: unlock(Y);
```
Extended Lock Graph

• Lock graph \((N, E_N)\)
  
  – Create a node \(n_X\) when a thread acquires lock \(X\)
  
  – Create an edge \((n_X, L, n_Y)\) when a thread acquires lock \(Y\) while holding lock \(X\), where \(L = (s_1, t, G, s_2)\)
    
    • \(s_1\): the thread segment \((s_1 \in S)\) where lock \(X\) was acquired
    • \(t\): the thread that acquires lock \(Y\)
    • \(G\): the set of locks that \(t\) holds when it acquires \(Y\)
    • \(s_2\): the thread segment where lock \(Y\) was acquired
Potential Deadlock Detection

• A cycle is valid (i.e., true positive) when every pair of edges
  \((m_{11}, (s_{11}, t_1, G_1, s_{12}), m_{12}), \text{ and } (m_{21}, (s_{21}, t_2, G_2, s_{22}), m_{22})\) in the cycle satisfies:
  • \(t_1 \neq t_2\), and
  • \(G_1 \cap G_2 = \emptyset\), and
  • \(\neg(s_{12} < s_{21})\)
    – The happens-before relation \(<\) is the transitive closure of the relation \(R\) such that \((s_1, s_2) \in R\) if there exists the edge from \(s_1\) to \(s_2\) in the thread segment graph
Thread Creation Example Revisit

main()

0: ...
1: `start(f1);`

f1()

1: `lock(X);`
2: `lock(Y);`
3: `start(f2);`
4: `unlock(Y);`
5: `unlock(X);`

f2()

11: `lock(Y);`
12: `lock(X);`
13: `unlock(X);`
14: `unlock(Y);`

\[ e_1 : (n_X, (s_2, t_1, \{X\}, s_2), n_Y) \]

\[ e_2 : (n_Y, (s_4, t_2, \{Y\}, s_4), n_X) \]
Revising Single Thread Cycle Example

main() {
  1: `start(Thread1);`
  2: `start(Thread2);`
}

Thread1() {
  11: `lock(X);`
  12: `lock(Y);`
  13: `unlock(Y);`
  14: `unlock(X);`
  15: `lock(Y);`
  16: `lock(X);`
  17: `unlock(X);`
  18: `unlock(Y);`
}

Thread2() {
  21: `lock(X);`
  22: `unlock(X);`
  23: `lock(Y);`
  24: `unlock(Y);`
}

\[ e_1: (n_X, (s_2, t_1, \{X\}, s_2), n_Y) \]

\[ e_2: (n_Y, (s_2, t_1, \{Y\}, s_2), n_X) \]
Revising Gate Lock Example

main() {
    start(Thread1);
    start(Thread2);
}

Thread1() {
    lock(X);
    lock(Y);
    lock(Z);
    unlock(Z);
    unlock(Y);
    unlock(X);
}

Thread2() {
    lock(X);
    lock(Z);
    lock(Y);
    unlock(Y);
    unlock(Z);
    unlock(X);
}

\[
e_1: (n_Y, (s_2, t_1, \{X, Y\}, s_2), n_Z)
\]

\[
e_2: (n_Z, (s_4, t_2, \{X, Z\}, s_4), n_Z)
\]
Detecting Potential Deadlock with Wait/Notify, Semaphore, etc*

```java
class BlockedBuffer {
    List buf = new ArrayList();
    int cursize = 0;
    int maxsize;

    BlockedBuffer(int max){
        maxsize = max;
    }

    sync boolean isFull(){
        return (cursize>=maxsize);
    }

    sync boolean isEmpty(){
        return (cursize == 0);
    }

    sync void resize(int m){
        maxsize = m;
    }

    sync void put(Object e){
        while (isFull())
            wait();
        buf.add(e);
        cursize++;
        notify();
    }

    Object get(){
        Object e;
        sync(this){
            while (isEmpty())
                wait();
            e = buf.remove(0);
            if (isFull()){
                cursize--;
                notify();
            } else
                cursize--;
            return e;
        }
    }
}

*P. Joshi et al., An Effective Dynamic Analysis for Detecting Generalized Deadlocks, FSE 2010*
```
main() {
    BoundedBuffer bf =
        new BoundedBuffer(1);
    (new Thread1(bf)).start();
    (new Thread2(bf)).start();
    (new Thread3(bf)).start();
}

Thread1(BoundedBuffer bf){
    bf.put(0);
    bf.put(1);
}

Thread2(BoundedBuffer bf){
    bf.resize(10);
}

Thread3(BoundedBuffer bf){
    bf.get();
}
Another Correct Execution Scenario

```java
main() {
    BoundedBuffer bf =
        new BoundedBuffer(1);
    (new Thread1(bf)).start();
    (new Thread2(bf)).start();
    (new Thread3(bf)).start();
}

Thread1(BoundedBuffer bf){
    bf.put(0);
    bf.put(1);}

Thread2(BoundedBuffer bf){
    bf.resize(10);}

Thread3(BoundedBuffer bf){
    bf.get();}
```

Thread1
```
bf.put(0)
bf.put(1)
```

Thread2
```
while(isFull())
    wait();
if(isFull()) {...
    notify();
}
```

Thread3
```
bf.get()
```

bf.put(1)

bf.put(1)

bf.resize(10)
main() {
    BoundedBuffer bf =
        new BoundedBuffer(1);
    (new Thread1(bf)).start();
    (new Thread2(bf)).start();
    (new Thread3(bf)).start();
}

Thread1(BoundedBuffer bf){
    bf.put(0);
    bf.put(1);
}

Thread2(BoundedBuffer bf){
    bf.resize(10);
}

Thread3(BoundedBuffer bf){
    bf.get();
}
CHECKMATE: Trace Program Model Checking

• Observe a multi-threaded program execution
• Retain only the synchronization operations observed during execution
  – Throw away all other operations like memory update and method calls
• Create a program from the retained operations (trace program)
• Model checking trace program
  – Check partial behaviors
Trace Program Example

main() {
    bf = Lock();
    isFull=false;
    start(t1);
    start(t2);
    start(t3);
}

t2() {
    bf.resize();
    lock(bf);
    isFull=false;
    unlock(bf);
}

t1() {
    bf.put(0)
    lock(bf);
    if(isFull)
        wait(bf);
    isFull=true;
    notify(bf);
    unlock(bf);
}

t3() {
    bf.get()
    lock(bf);
    if(isFull)
        notify(bf);
    unlock(bf);
}

Thread1    Thread2    Thread3

bf.put(0)
bf.put(1)
bf.resize(10)
bf.get()

bf.put(0)
bf.put(1)
bf.get()