Concurrency Bugs

Haibo Chen

Slides adapted from author’s
Writing concurrent program is hard

Concurrent programs are prone to concurrency bugs

Concurrency bugs have notorious characteristics
   Non-deterministic
   Hard to test and diagnose

But in multi-core era, we need to write concurrent program to harness the power of multi-core
I am convinced more than ever that this type of work is **very difficult**, and that every effort to do it with other than **the best people** is doomed to either **failure** or moderate success at **enormous cost**.

Edsger Dijkstra

*The Structure of the “THE” – Multiprogramming System*

1968
Approaches to addressing the problem

Concurrency bug detection
  Race detection
  Atomicity violation detection
  Deadlock bug detection

Concurrent program testing
  Exhaustive testing
  Different coverage criteria proposed

Concurrent programming language/model design
  Transactional memory
Outline

Data Race

Concurrency Bug Characteristics

Data race detection
What is Race?
Data Race

An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but the operations must be done in the proper sequence in order to be done correctly

- multithread
- distributed Programs

Key reason

- separate processes or threads of execution depends on same shared state
Data Race

\[ v = 0 \]

[thread-1]
\[ v += 1; \]

[thread-2]
\[ v += 1; \]

\[ v = ? \]
Data Race

[Thread-1]
%reg = v;
%reg++; v = %reg;

[Thread-2]
%reg = v;
%reg++; v = %reg;

v = 0

T1 reads 0 to %reg
T1 increments %reg to 1
T1 stores %reg to v (v = 1)
T2 reads 1 to %reg
T2 increments %reg to 2
T2 stores %reg to v (v = 2)

v = 2; (memory)
Data Race

[thread-1]
%reg = v;
%reg++;  
v = %reg;

[thread-2]
%reg = v;
%reg++;  
v = %reg;

v = 0 (memory)
T1 reads 0 to %reg
T2 reads 0 to %reg
T1 increments %reg to 1
T2 increments %reg to 1
T1 stores %reg to v (v = 1)
T2 stores %reg to v (v = 1)

v = 1; (memory)
The definition of data race:

1. Two concurrent threads access a shared variable.
2. At least one access is a write.
3. The threads use no explicit mechanism to prevent the accesses from being simultaneous.
Remove Race Condition

Critical Sections

v = 0

[thread-1]
%reg = v;
%reg++; v = %reg;

[thread-2]
%reg = v;
%reg++; v = %reg;

v = ?
Remove Race Condition

- **Mutually-exclusive**
  - lock & unlock
  - cli & sti

```plaintext
v = 0

[thread-1]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)

[v = 2]

[thread-2]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)

v = 2
```
Data Race Bug

Thread-1
OpenInputStream()
{
    PostEvent();
    ...
    m_inputStream = ...
    ...
}

Thread-2
ProcessCurrentURL()
{
    WaitEvent();
    ...
    if (m_inputStream){
        AsyncRead(m_inputStream);
    }
    ...
}

file: nsSocketTransport.cc  file: nsImapProtocol.cpp

*Data race bug in Mozilla*
CONCURRENCY BUG
CHARACTERISTICS
Goal: Improve state-of-the-art

Concurrency bug detection
- What types of bugs are un-solved?
- How helpful is it for bug fixing

Need to know the characteristics of real-world concurrency bugs!

Concurrent program language design
- How many mistakes can be avoided
- What other support do we need?

Real-world bug pattern
Real-world bug fixing

Real world bug patterns and fixing strategies
Previous bug characteristic study

Characteristic study on general software bugs
- Provide a lot of inspiration
- Not enough guidance for concurrency bugs

Characteristic study on concurrency bugs
- Some preliminary study
  - Lack real-world bug samples
- Hard to draw conclusions on bug characteristics
Contributions

105 real-world concurrency bugs from 4 large open source programs

Study from 4 dimensions

- Bug patterns
- Manifestation condition
- Diagnosing strategy
- Fixing methods

Implications for:

- Bug detection
- Software testing
- PL design
Outline

Methodology

Findings and implications
  Bug pattern
  Bug manifestation
  Bug fixing

Conclusions
## Application sources

<table>
<thead>
<tr>
<th>Software Type</th>
<th>MySQL</th>
<th>Apache</th>
<th>Mozilla</th>
<th>OpenOffice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C++/C</td>
<td>Mainly C</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>LOC (M line)</td>
<td>2</td>
<td>0.3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Bug DB history</td>
<td>6 years</td>
<td>7 years</td>
<td>10 years</td>
<td>8 years</td>
</tr>
</tbody>
</table>

Different types of real world applications
## Bug sources

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>Apache</th>
<th>Mozilla</th>
<th>OpenOffice</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-deadlock</td>
<td>14</td>
<td>13</td>
<td>41</td>
<td>6</td>
<td>74</td>
</tr>
<tr>
<td>Deadlock</td>
<td>9</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>31</td>
</tr>
</tbody>
</table>

## Limitations

- No scientific computing applications
- No JAVA programs
- Never-enough bug samples
Non-Deadlock Bug Pattern

Classified based on root causes

Categories:
Atomicity violation
Desired atomicity of certain code region is violated

Order violation
The desired order between two (sets of) accesses is flipped

Others
Atomicity violation

The desired atomicity of certain code region is violated.
Example of atomicity violation

Thread 1

```c
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...)
    ...
}
```

Thread 2

```c
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp
Example of atomicity violation

```c
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...)
    ...
}
```

Thread 1 should be atomically executed

```
Thread 2
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp
Example of atomicity violation

if (thd->proc_info) {
    ... 
    fputs(thd->proc_info, ...)
    ...
}

thd->proc_info = NULL;

MySQL ha_innodb.hpp
Order violation

• The desired order between two (sets of) accesses is flipped
Example of order violation

```cpp
thread 1

void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    ...
}

thread 2

void MgmtSrvr::status(…)
{
    *myid = node.m_state.dynamicid;
    ...
}

MySQL NodeState.hpp
```
Example of order violation

```cpp
void NodeState::setDynamicId(int id) {
    // initialized here
    dynamicid = id;
    ...
}

void MgmtSrvr::status(...) {
    *myid = node.m_state.dynamicid;
    ...
}
```

MySQL NodeState.hpp
### Example of order violation

```
void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    ...
}
```

```
void MgmtSrvr::status(…)
{
    *myid = node.m_state.dynamicid;
    ...
}
```

The code snippet shows a situation where the `dynamicid` variable is initialized in one thread (Thread 1) and then accessed by another thread (Thread 2) in a different order, which can lead to a data race condition. The correct order should be to initialize the variable first and then access it, as shown by the `buggy order` arrow.
Non-deadlock bug pattern

Implications

We should focus on atomicity violation and order violation
Non-deadlock bug pattern

Implications

We should focus on atomicity violation and order violation

Bug detection tools for order violation bugs are desired
How to trigger a bug

Bug manifestation condition
A specific execution order among a smallest set of memory accesses
The bug is guaranteed to manifest, as long as the condition is satisfied

How many threads are involved?
How many variables are involved?
How many accesses are involved?
Single Variable vs. Multiple Variable

Findings

1 Variable
- MySQL: 49 (66%)
- Apache: 25 (34%)

> 1 Variables
- Mozilla
- OpenOffice

1 Variable
- MySQL
- Apache
- Mozilla
- OpenOffice
Single Variable vs. Multiple Variable

Single variables are more common
The widely-used simplification is reasonable

Multi-variable concurrency bugs are non-negligible
Techniques to detect multi-variable concurrency bugs are needed
Multi-Variable Concurrency Bug Example

Thread 1

```
js_FlushPropertyCache ( ... ) {
    memset ( cache->table, 0, SIZE);
    ...
    cache->empty = TRUE;
}
```

Thread 2

```
js_PropertyCacheFill ( ... ) {
    cache->table[indx] = obj;
    ...
    cache->empty = FALSE;
}
```

Control the order among accesses to any one variable can *not* guarantee the bug manifestation.
Non-deadlock bugs
Number of Accesses

MySQL  Apache  Mozilla  OpenOffice

1 acc.  2 acc.  3 acc.  4 acc.  >4 acc.

7(9%)
Deadlock bugs
Number of Accesses

MySQgL  Apache  Mozilla  OpenOffice

1 acc.  2 acc.  3 acc.

4 acc.  >4 acc.

1 (3%)
Implications

Only a few percentage bugs need more than 4 access to trigger

Concurrent program testing can focus on small groups of accesses
  The testing target shrinks from exponential to polynomial
Number Threads Involved

101 out of 105 (96%) bugs involve at most two threads

Most bugs can be reliably disclosed if we check all possible interleaving between each pair of threads

Few bugs cannot

Example: Intensive resource competition among many threads causes unexpected delay
How Were Non-Deadlock Bugs Fixed?

Adding/changing locks 20 (27%)
Condition check 19 (26%)
Data-structure change 19 (26%)
Code switch 10 (13%)
Other 6 (8%)

Implications
No silver bullet for fixing concurrency bugs.
Lock usage information is not enough to fix bugs.
How Were Non-Deadlock Bugs Fixed?

- Give up resource acquisition: 19 (61%)
- Change resource acquisition order: 7 (23%)
- Split the resource to smaller ones: 1 (3%)
- Others: 4 (13%)

We need to pay attention to the correctness of \``fixed\'' deadlock bugs.
Can transactional memory be helpful?

What is transactional memory?

Allow a group of load and store instructions to execute in an atomic way

Analogous to database transactions for controlling access to shared memory in concurrent computing
Can transactional memory be helpful?

Only focus on basic atomicity and isolation properties of TM

How many bugs can be easily avoided by TM

What are the concerns for using TM
  - Transaction length
  - Rollback

How many bugs can not be avoided by TM
Findings about TM

- MySQL: 41 (39%)
- Apache: 8 (8%)
- Mozilla: 30 (28%)
- OpenOffice: 20 (19%)

Help:
- MySQL: 8 (8%)
- Apache: 8 (8%)
- Mozilla: 30 (28%)
- OpenOffice: 6 (6%)

Rollback Concern:
- MySQL: 41 (39%)
- Apache: 8 (8%)
- Mozilla: 30 (28%)
- OpenOffice: 20 (19%)

Concern:
- MySQL: 41 (39%)
- Apache: 8 (8%)
- Mozilla: 30 (28%)
- OpenOffice: 20 (19%)

No help:
- MySQL: 41 (39%)
- Apache: 8 (8%)
- Mozilla: 30 (28%)
- OpenOffice: 20 (19%)
Other findings

Impact of concurrency bugs
~ 70% leads to program crash or hang

Reproducing bugs are critical to diagnosis

Programmers lack diagnosis tools
Most are diagnosed via code review
Reproduce bugs are extremely hard and directly determines the diagnosing time

60% 1st-time patches contain concurrency bugs (old or new)
Summary

• Bug detection needs to look at order-violation bugs and multi-variable concurrency bugs

• Testing can target at more realistic interleaving coverage goals

• Fixing concurrency bugs is not trivial and not easy to get right
  – Support from automated tools is needed
Summary

Diagnosing tools are needed, especially bug reproducing tools
  Deterministic replay can be a helpful tool

Current bug detection tools are not widely used
  LLVM now has a subproject ThreadSanitizer

TM is promising, if some concerns can be addressed
Questions

Bug pattern, percentage, fixing strategies table
   Intended for you to get a better feel of concurrency bugs

Transactional memory summary
Language designed for concurrency summary

Why concurrent programming is hard?
   Welcome to share your experience
DATA RACE DETECTION
Data Race Detectors

Two major categories:
- Happens-before based
- Lockset based
HAPPENS-BEFORE BASED
The *happens-before* relation is a means of ordering events based on the causal relationship of pairs of events in a concurrent system. 

- denoted:
- Formulated by Leslie Lamport
- strict partial order on events
- without using physical clocks

*Times, Clocks, and Ordering of Events in a Distributed System*
Rules in HB Relation

For the events A and B,

**HB1**: On the same sequential thread,
\[ A \rightarrow B \text{ if } A \text{ executes before } B. \]

**HB2**: On the different threads,
\[ A \rightarrow B \text{ if there is a synchronization that dictates } A \text{ precedes } B. \]

**HB3**: If \[ A \rightarrow B \text{ and } B \rightarrow C \], then \[ A \rightarrow C \].
HB based Detectors

The definition of data race:
  1. a pair of accesses to the same memory location
  2. at least one access is a write
  3. neither one happens-before the other
Example

without HB relation !!
Example

```
Example

v = 0

[thread-1]
%reg = v;
%reg++;        (Red-marked)
v = %reg;

[thread-2]
%reg = v;
%reg++;        (Red-marked)
v = %reg;

v = ?
```

without HB relation !!
Example

```
[v = 0]

[thread-1]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)

[thread-2]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)

[v = 2]
```

HB relation!
Pros and Cons

Pros:
Detect true data race

Cons:
Difficult to implement efficiently
• Each thread, shared-memory location, and concurrent access
Depend on the interleaving produced by the scheduler
• Miss data race
LOCKSET BASED
The definition of lock:

1. a synchronization object used for mutual exclusion
2. a lock is either available or owned by a thread.
3. the operations on a lock m are lock(m) and unlock(m)
Lockset based Detectors

The definition of data race:

1. A pair of accesses to the same memory location
2. At least one access is a write
3. No lock protects all accesses to the same data
Lockset

Locking Discipline

– A programming policy that ensures the absence of data races
  • e.g. “every variable shared between threads is protected by a mutual exclusion lock”

Principle

– Check all shared memory accesses follow a consistent lock discipline
  • monitors all reads and writes, and infer the protection relation from the execution history
Algorithm of Lockset

The summary:

1. Let $\text{locks\_held}(t)$ be the set of locks held by thread $t$
2. For each $v$, initialize $C(v)$ to the set of all locks
3. On each access to $v$ by thread $t$, set $C(v) := C(v) \cap \text{locks\_held}(t)$ if $C(v) = \{\}$, then issue a warning
### Example

<table>
<thead>
<tr>
<th>Programs</th>
<th>locks_held(t)</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(mu1);</td>
<td>{ }</td>
<td>{mu1, mu2}</td>
</tr>
<tr>
<td>lock(mu2);</td>
<td>{mu1}</td>
<td></td>
</tr>
<tr>
<td>v = v + 1;</td>
<td>{mu1, mu2}</td>
<td>{mu1, mu2}</td>
</tr>
<tr>
<td>unlock(mu2);</td>
<td>{mu1}</td>
<td></td>
</tr>
<tr>
<td>v = v + 2;</td>
<td>{mu1}</td>
<td>{mu1}</td>
</tr>
<tr>
<td>unlock(mu1);</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>lock(mu2);</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>v = v + 1;</td>
<td>{mu2}</td>
<td>{} Warning!!</td>
</tr>
<tr>
<td>unlock(mu2);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Three Challenges

#1 Initialization
  – Shared variables are frequently initialized without holding a lock

#2 Read-only Shared Variable
  – write once and read all the time without lock

#3 Read-Write Lock
Solution to #1 and #2

Delay the refinement until the shared variable has been initialized

– No easy way to know when initialization is done
– Heuristic way: the first time to be accessed by the 2nd thread
Solution to #1 and #2

1. Virgin
   - The variable is **NEW** and has not yet been referenced by any thread.

2. Exclusive
   - The variable has been accessed by **ONE** thread only. Not update \( C(v) \)

3. Shared
   - \( C(v) \) is updated, but data races are **NOT** reported even if \( C(v) \) is empty.

4. Shared-Modified
   - \( C(v) \) is updated, and data races are reported if \( C(v) \) is empty.
Solution to #3

Modify the algorithm:

1. Let `locks_held(t)` and `write_locks_held(t)` be the set of locks held in any mode and write mode by thread t.

2. For each v, initialize `C(v)` to the set of all locks.

3. On each read to v by thread t, set `C(v) := C(v) ∩ locks_held(t)`
   if `C(v)={ }`, then issue a warning.

4. On each write to v by thread t, set `C(v) := C(v) ∩ write_locks_held(t)`
   if `C(v)={ }`, then issue a warning.
Pros and Cons

Pros:
More efficient way to detect data race
Predict data race that have not manifest

Cons:
Exists report false positive
  • Memory reuse
Limits the synchronization method to lock