Detecting and Fixing Concurrency Bugs

Shan Lu
University of Chicago
About course assignment

• Study 5 bugs in an open-source application’s Bugzilla
  – Pick the keyword you like
  – Pick the application you like (or use cbs ...)
  – Write the following for each bug
    • What is the bug root cause (fault)
    • What errors might be caused by the bug
    • What is the failure symptom of this bug
    • What is the fix strategy of developers
    • Can this bug be automatically detected? Exposed during testing? Automatically diagnosed or fixed?

• You can work in group
Types of bugs

Fighting approaches

constraints
Different types of bugs

- Memory bugs
  - Memory leaks
  - Buffer overflow
  - Null-ptr dereference
  - Uninitialized read

- Semantic bugs

- Concurrency bugs

- Performance/energy bugs

Faults in Linux: Ten years later. ASPLOS’11
Bug Characteristics in Open Source Software. EMSE’13
Don’t hesitate to ask me questions!
Background

Thread
Concurrency Bugs
Thread vs. Process

• Process – resource management unit
  – Nothing is shared among processes, except ...
  – Parent & child share initial image

• Thread – execution/scheduling unit
  – The address space is completely* shared among threads under the same process

See example code
Sources of non-determinism

• race.c

• On single-core machines
  – System event non-determinism

• On multi-core machines
  – System event non-determinism
  – (Parallel) hardware on-determinism
Thread synchronization (I)

• Lock
  – Enforce mutual exclusion

• Condition variable
  – Enforce pair-wise ordering

• What is needed to synchronize ...?

(1) Thread 1 \(x++\); Thread 2 \(x++\);

(2) Thread 1 \(p=\text{malloc}(10)\); Thread 2 \(*p=10\);
Thread synchronization (II)

• Semaphore
  – A counter (can be initialized with any positive value)
  – P (acquire one piece of resource)
  – V (release one piece of resource)

• What is needed to synchronize ...?
  (1) \textit{Thread 1 } x++; \quad \textit{Thread 2 } x++;;
  (2) \textit{Thread 1 } p=\text{malloc}(10); \quad \textit{Thread 2 } *p=10;
Thread APIs

- pthread_create
- pthread_join
- pthread_mutex_lock
- pthread_mutex_unlock
- pthread_cond_wait
- pthread_cond_signal
- ...

Other way of parallel execution

• Shared memory vs. message passing
What are concurrency bugs?

• Untimely accesses among threads (buggy interleavings)

MySQL

Thread 1

if (proc){
    tmp=*proc;
}

proc = NULL;

Thread 2

Mozilla

Thread 1 (child)

mThd=CreateThd();

_thread = mThd->state;

Thread 2 (parent)
It is important to fight con. bugs
Different aspects of fighting bugs

- In-house bug avoidance
- In-house detection & testing
- In-house verification

- In-field bug prevention
- In-field failure recovery
- In-field failure diagnosis

Low overhead

High accuracy
Outline

• What are concurrency bugs
• Concurrency bug detection
• Concurrency bug exposing
• Concurrency bug fixing
• Others
• Conclusion
Outline

- What are concurrency bugs
- Concurrency bug detection
  - Concurrency bug exposing
  - Concurrency bug fixing
- Others
- Conclusion
The key challenges

- What type of interleaving is buggy?
- Large state space
- False positives
- False negatives
- Overhead
Data race

• Definition

• Does this pattern match our examples?

• How to get rid of a data race?

Dinning’90, Netzer’91, Choi’91, Savage’97, Larus’98, Choi’02, O’Callahan’03, Yu’05, Aiken’06
How to detect data races?

- How do I know the execution of two accesses are concurrent?

- What does basic run-time monitoring tell us?

```c
count ++; <thread 1>
... //millions of instructions in between
count ++; <thread 2>
```
Physical time vs. logical time

- From Leslie Lamport
- What ordering do we know for sure in a distributed environment?

- Logical time based on causality/happens-before relationship
  - Vector timestamp
  - Scalar timestamp
What ordering is guaranteed?
Logical time

• Operations within one thread are (happens-before) ordered following program semantics
• Message sending is (happens-before) ordered before message receiving
• Ordering is transitive
  – A→B, B→C → A→C
How to represent logical time?
(scalar) logical clock

• Design a clock that can reflect the happens-before order
  – Increment within one process
  – Increment when receiving a message
Vector clock
How to use logical time in race det?

- What is the causality relationship here?
- Example 1
  
  \[
  \begin{align*}
  &Thread 1 &Thread 2 \\
  &tmp=x; &tmp=x; \\
  &x = tmp+1; &x=tmp+1;
  \end{align*}
  \]
- Example 2
  
  \[
  \begin{align*}
  &Thread 1 &Thread 2 (child) \\
  &p=malloc(10); &*p=10; \\
  &pthread_create(...) \\
  \end{align*}
  \]
- Example 3 (lock)
How to detect data races?

- Happen-before algorithm
  - Use logic time-stamps to find concurrent accesses

Thread 1
lock (L); <0,1>
ptr = NULL; <0,2>
unlock (L); <0,3>

ptr = malloc(10);
lock (L);
ptr[0] = 'a';
unlock (L);
How to detect data races?

• Happen-before algorithm
  – Use logic time-stamps to find concurrent accesses

Thread 1     Thread 2

ptr=NULL;     <,>
barrier(&b);  <,>

<,> barrier(&b);
<,> ptr = malloc(10);
<,> ptr[0] = ‘a’;
How to detect data races?

• Happen-before algorithm
  – Use logic time-stamps to find concurrent accesses

```
Thread 1
<1,0> ptr = malloc(10);
<,> lock (L);
<,> ptr[0] = 'a';
<4,0> unlock(L);

Thread 2
lock (L); <4,1>
ptr=NULL; <4,2>
unlock(L); <,>
```
Happen-before algorithm summary

• Strength
  – Work for different types of synchronization
  – Few false positives in race detection

• Weakness
  – False negatives in race detection
How to detect data races?

• Lock-set algorithm
  – A common lock should protect all conflicting accesses to a shared variable

Thread 1
lock (L);
ptr=malloc(10);
ptr[0]=’a’;
unlock(L);

Thread 2
lock (L);
ptr=’a’;
unlock(L);

Thread 1
lock (L);
ptr=malloc(10);
ptr[0]=’a’;
unlock(L);

Thread 2
lock (L);
ptr=’a’;
unlock(L);
Lock-set algorithm summary

• Strength
  – Fewer false negatives
    • Interleaving in-sensitive

• Weakness
  – More false positives
  – Cannot handle non-lock synchronization

• How to solve the false positive problem?
  – H-B & Lockset hybrid race detection
Are we done?

- **Performance**
  - Huge problem
  - Solution?

- **False positives**
  - Huge problem
  - 90% of data races do not lead to visible failures* [PLDI’07]
  - Solution?

- **False negatives**

  Thread 1
  
  ```c
  ptr = malloc(10);
  lock (L);
  ptr[0] = 'a';
  unlock(L);
  ```

  Thread 2
  
  ```c
  while (!flag) {};
  flag = TRUE;
  ```
How to speed-up?

• Hardware support
  – Non-existing
  – Existing

• Sampling
How to do better?

Let’s find a more accurate root-cause pattern for concurrency bugs!
Root-cause patterns

• A study of 105 real-world concurrency bugs
40

What do programmers want?

Concurrency
Bugs

Atomicity
Violation
Bugs

Order
Violation
Bugs

70%

30%

Root-cause patterns

Thread 1 (child) Thread 2 (parent)

mThd=CreateThd();

_state = mThd->state;

MySQL

Thread 1

Thread 2

if (proc){
    tmp=*proc;
}

proc = NULL;

Mozilla

atomicity violation bugs

order violation bugs
Root-cause patterns

- Concurrency
- Bugs
- Atomicity Violation
- Bugs

Single Variable

Multiple Variable

70%

30%
Why did I do this study?
How to detect atomicity-violations?

• Problem 1
  – Know which code region should maintain atomicity

• Problem 2
  – Judge whether a code region’s atomicity is violated
How to detect atomicity-violations?

- Problem 1
  - Know which code region should maintain atomicity

- Problem 2
  - Judge whether a code region’s atomicity is violated
Solution to problem 2

• Atomicity violation = unserializable interleaving

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS'06
Associating synchronization constraints with data in an object-oriented language, POPL'06
Solution to problem 2

- Atomicity violation = unserializable interleaving

Thread 1
- access x
- access x

Thread 2
- Read x
- Read x
- Read x
- Write x
- Read x
- Write x
- Read x
- Write x
- Write x

- 4 out of 8 cases are violations

- Inconsistent views
- Too early overwritten
- Leaking intermediate value
- Using stale value

Both hardware and software solutions exist
Solution to problem 1

• Which code regions are expected to be atomic?
  – Manual annotation
  – ??

```c
if (proc){
    tmp=*proc;
}
```

MySQL

```c
while (!flag) {}
flag=TRUE;
```

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS’06
Inference based bug detection
Infer likely program invariants

- What is the typical value of x?
- What is the ...?
- How to use it to detect general semantic bugs?
- How to use it to detect memory bugs?
- How to use it to detect concurrency bugs?
Solution to problem 1

• Which code regions are expected to be atomic?
  – Manual annotation
  – Training/Learning
  – Testing validation

Thread 1

if (proc){
  tmp=*proc;
}

proc = NULL;

MySQL

Thread 2

while (!flag) {};
flag=TRUE;

Thread 1

Thread 2

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS’06
What are order violations?

- Expected order between two operations are flipped
- Can it be detected by atom. vio. detectors?
- Can it be detected by race detectors?

```
mThd = CreateThd();
_state = mThd->state;
```

Thread 1 (child)  Thread 2 (parent)

Mozilla
How to detect order violation?

- Problem 1
  - How to judge which is the correct order?
- Problem 2
  - How to detect the order violation?

Thread 1 (child)  Thread 2 (parent)

mThd = CreateThd();
_state = mThd->state;

Mozilla
Solutions

• How to judge which is the correct order?
  – Learning based techniques [Micro’09, OOPSLA’10]
  – Semantic guided techniques [ASPLOS’11]

• How to detect the order violation
  – Easy

```c
Thread 1 (child)   Thread 2 (parent)

mThd = CreateThd();
_state = mThd->state;

Mozilla
```
What are multi-var conc. bugs?

• Multi-variable bugs
  – Untimely accesses to correlated variables
• Can it be detected by race detectors?
• Can it be detected by AVIO?

```c
if(InProgress)
  if(isBusy)
    if(URL == NULL)
      __assert_fail(),
      ...
```

Thread 1

Thread 2

InProgress=FALSE;
URL = NULL;

if(isBusy) {
  if(URL == NULL)
    __assert_fail(),
    ...
}
How to detect multi-variable bugs?

• Problem 1
  – How to judge which variables are correlated?

• Problem 2
  – How to detect untimely accesses

```c
Thread 1

InProgress = FALSE;
URL = NULL;

Thread 2

if(InProgress)
  isBusy = TRUE;

if(isBusy) {
  if(URL == NULL)
    __assert_fail(), ...
}

Mozilla
```
Solutions

• Which variables are correlated?
  – Variables that are frequently accessed together

• How to detect the violation?
  – Extend existing single-variable bug detectors

```
struct JSCache {
  ...
  JSEntry table[SIZE];
  bool empty;
  ...
}
Mozilla

struct JSRuntime {
  ...
  int totalString;
  double lengthSum;
  ...
}
Mozilla

struct fb_var_screeninfo {
  ...
  int red_msb;
  int blue_msb;
  int green_msb;
  int transp_msb;
}
Linux
```
Solutions

• Which variables are correlated?
  – Variables that are frequently accessed together
• How to detect the violation?
  – Extend existing single-variable bug detectors
Are we done?

• Are these “learning”-based techniques perfect?
Are we done?

• False positives
  – Still a problem!

• False negatives
  – Still a problem!
Break
How to do better?

```c
if (proc){
    tmp=*proc;
    proc = NULL;
}

MySQL
```
How to do better?

If we cannot find a more accurate root-cause pattern, let’s look at the effect patterns of concurrency bugs!
The lifecycle of bugs

Fault ➔ Error ➔ Failure

trigger ➔ propagate

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure

Fault ➔ Error ➔ Failure
The lifecycle of \((\text{most})\) concurrency bugs

*based on 70 real-world bugs*

Thread 2

Thread 1

![Diagram showing concurrency issues]

Data races
Atomicity violations
  - single variable
  - multiple variables
Order violations
...
The lifecycle of (most) concurrency bugs

based on 70 real-world bugs

Thread 2

Thread 1

Critical Read

Error

propagate

Memory errors
- NULL ptr
- Dangling ptr
- Uninitialized read
- Buffer overflow
- Semantic errors
The lifecycle of (most) concurrency bugs

based on 70 real-world bugs

Thread 2

Thread 1

Fault → Error → Failure

trigger

propagate

short

single-threaded

Crash @ invalid memory
Crash @ assertion
Infinite loops
Incorrect outputs
Error messages
Examples

Thread 1

if (proc){
    tmp=* proc;
}

proc = NULL;

MySQL

Thread 2

mThd=CreateThd();
-state = mThd->state;

Mozilla

atomicity violation ➔ null-ptr deref. ➔ crash

Thread 1

order violation ➔ uninitialized read ➔ crash

Thread 2
Summary of effect characteristics

- Simple error/failure patterns
- Single-threaded error propagation
- Short error propagation
Cause-oriented approach

Thread 2

Thread 1

Interleavings that match certain cause patterns are buggy

race detectors

Thread 1
read x
Thread 2
write x
Thread 1
read x

atom. vio. detectors

Thread 2
read x
write x

• Limitations
  – False positives
  – False negatives
Effect-oriented approach

Thread 2
Thread 1

Interleavings that lead to certain effect patterns are buggy

- Step 1: *Statically* identify potential failure/error site
- Step 2: *Statically* look for critical reads
- Step 3: *Dynamically* identify buggy interleaving

Fewer false positive
Fewer* false negative
i like the mapping in paper: cause maps to xxx effects; effect map back to xxx.

if i refer to interleaving here, we need to define interleaving earlier
Our tools

Thread 2

Thread 1

Fault → Error → Failure

- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow

- Semantic errors

- Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS’10
Our tools

Thread 2

Thread 1

Fault → Error → Failure

Memory errors
- NULL ptr
- Dangling ptr
- Uninitialized read
- Buffer overflow

Semantic errors

Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS’11
Our tools

Thread 2

Thread 1

Fault → Error → Failure

- Memory errors
- NULL ptr
- Dangling ptr
- Uninitialized read
- Buffer overflow
- Semantic errors
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS’11
ConSeq bug example

Thread 1

```
InProgress = FALSE;
URL = NULL;
```

Thread 2

```
if(InProgress)
    isBusy = TRUE;

if(isBusy) {
    if(URL == NULL)
        __assert_fail(),
    ...
}
```

Mozilla

---

MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. 
Shan Lu, et. al., SOSP’09 74
the sosp, muvi reference should be put earlier
Shan Lu, 2014-1-8
Step 1: Identify potential failure sites

Statically look for places where failures could happen

<table>
<thead>
<tr>
<th>Failure Type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion Failure</td>
<td></td>
</tr>
<tr>
<td>Error Message</td>
<td></td>
</tr>
<tr>
<td>Incorrect output</td>
<td></td>
</tr>
<tr>
<td>Infinite loop</td>
<td></td>
</tr>
</tbody>
</table>

Number of failure sites in MySQL: ~1000
Step 1: Identify potential failure sites

Statically look for places where failures could happen

```c
if(InProgress)
    isBusy=TRUE;
if(isBusy){
    if(URL == NULL){
        __assert_fail();
    }
}
```
Step 2: Look for critical reads

Statically find shared mem. reads that impact failure sites

```c
if(InProgress)
    isBusy=TRUE;
if(isBusy){
    if(URL == NULL){
        _assert_fail();
    }
}
```
Stage 3: Look for buggy interleavings

Dynamic analysis looks for interleavings that provide critical reads with bad values

Thread 1

... 

InProgress=FALSE;

URL = NULL;

Thread 2

if(InProgress)
    isBusy=TRUE;
if(isBusy) {
    if(URL == NULL){
        __assert_fail(),
    }
}
Is the alternative data dependence feasible in future runs?
Dependence feasibility analysis

• Can synchronization prevent a data dependence?
Dependence feasibility analysis

- Locks could make a data-dependence infeasible

- Barriers could make a data-dependence infeasible
Put everything together

Identify failure sites → Identify Critical Reads → Identify Suspect Interleavings → Suspect Interleaving Testing → Bug reports

Thread 1

...  
InProgress = FALSE;

Thread 2

if(InProgress)
  isBusy = TRUE;
if(isBusy){
  if(URL == NULL){
    __assert_fail();
  }
}

URL = NULL;
ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS’10
ConMem bug example

• What are the errors?
• How to detect them using dynamic analysis?

Thread 1

```
if (proc){
    tmp=* proc;
}
```

MySQL

Thread 2

```
proc = NULL;
```

Thread 1

```
mThd=CreateThd();
_state = mThd->state;
```

Mozilla

Thread 2
5-min Break?
Summary of conc. bug detection

• How to detect them?
  – Find patterns
    • Cause patterns
    • Effect patterns

• What are the remaining challenges?
  – Performance
  – False negatives
  – False positives
    • Customized synchronization

• The state of practice
  – Race detection; Atom. detection; ...
Outline

• What are concurrency bugs
• Concurrency bug detection
• Concurrency bug exposing
  • Concurrency bug failure recovery
  • Concurrency bug fixing
• Others
• Conclusion
Exposing Concurrency Bugs
Background --- Software Testing

• Testing space

• Coverage criteria
  – Testing property

• Test suite

• Software testing is extremely important!
The challenges

• Huge state space

• What is the coverage criteria?

• How to cover a testing property?
Background in testing

• Coverage criteria
  – Examples
  – Complexity vs. Capability

• Test input design
What are the coverage criteria?

- Total-order [TSE92]
- ALL-DU [ICSM92, ISSTA98]
- Synchronization [PPoPP05]
- Function [SoQua07]

- Bug-pattern based
  [Chess, RaceFuzzer, CTrigger...]

---

Thread 1

```java
if (proc){
  tmp=*proc;
}
```

Thread 2

```java
proc = NULL;
```

**MySQL**

---

Thread 1 (child)  Thread 2 (parent)

```java
_thread = _state = mThd->state;
```

```java
mThd = CreateThd();
```

**Mozilla**
How to cover a testing property?

How can I make A execute before B?

- Ad-hoc solution
  - Single-core based
  - Multi-core based
- Constraint-solving based solution [Madhu Viswanathan, NEC]

- How many properties can be covered in one run? [Madan Musuvathi]
Summary of exposing con. bugs

• Key challenges

• Key solutions

• What are the remaining challenges?
  – Better coverage criteria
  – Input generation
  – Regression testing
  – Unit testing
Summary of the day

• Concurrency bug detection
  – Cause based detection
    • Data race; atomicity violation; order violation; single variable; multi-variable
  – Effect based detection
  – Bug exposing (testing)

• Detection mechanisms
  – Run-time analysis
  – Static analysis
  – Learning-based technique