第十五讲：死锁和并发错误检测

第 5 节：并发错误检测

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 4 月 12 日

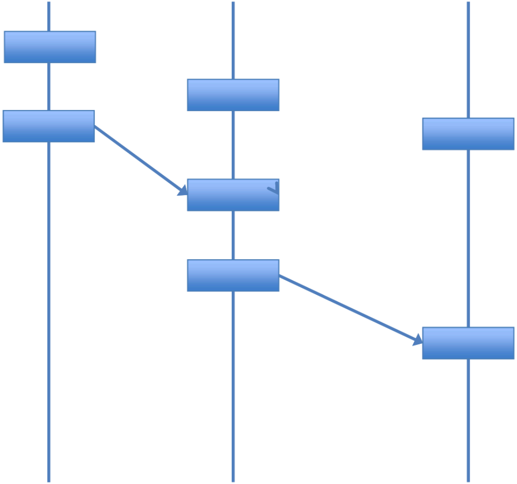
提纲

1. 第 5 节：并发错误检测
   - Concurrency Bug
   - Concurrency Bug Detection
   - AVIO
   - ConSeq & ConMem

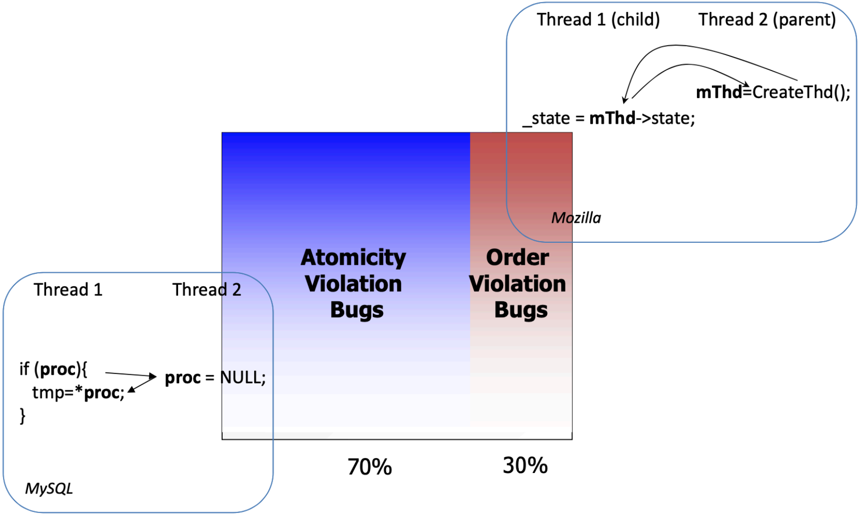
Ref: Shan Lu, Detecting and Fixing Concurrency Bugs, University of Chicago

Concurrency bug

What ordering is guaranteed?



Concurrency bug: Voilation



Thread 1 (child) Thread 2 (parent)

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

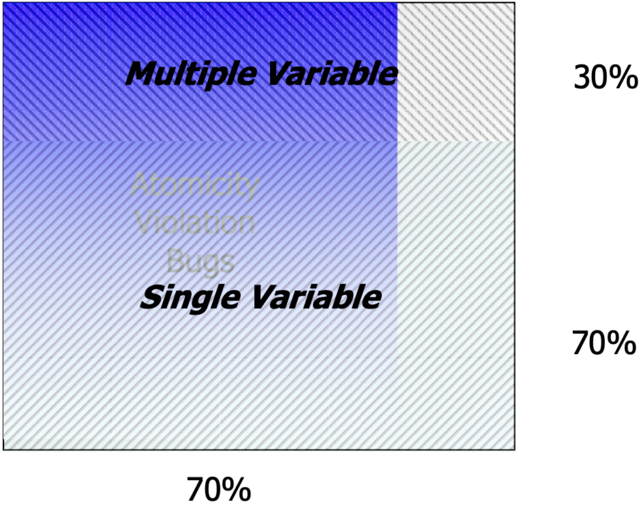
Mozilla

Thread 1 Thread 2

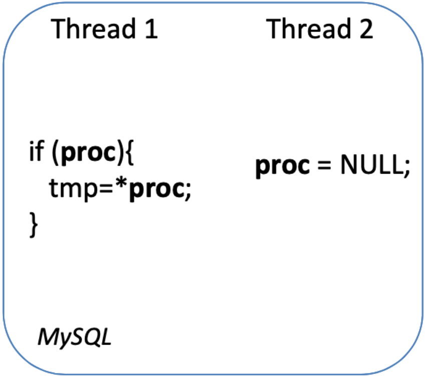
```
if (proc){                proc = NULL;
  tmp=*proc;
}
```

MySQL

Concurrency bug: Variable



Atomicity Violations



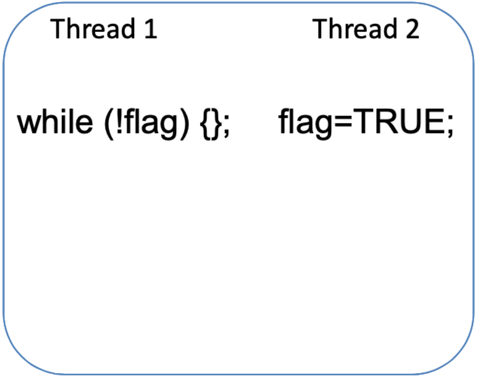
Thread 1

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

Thread 2

```
proc = NULL;
```

MySQL



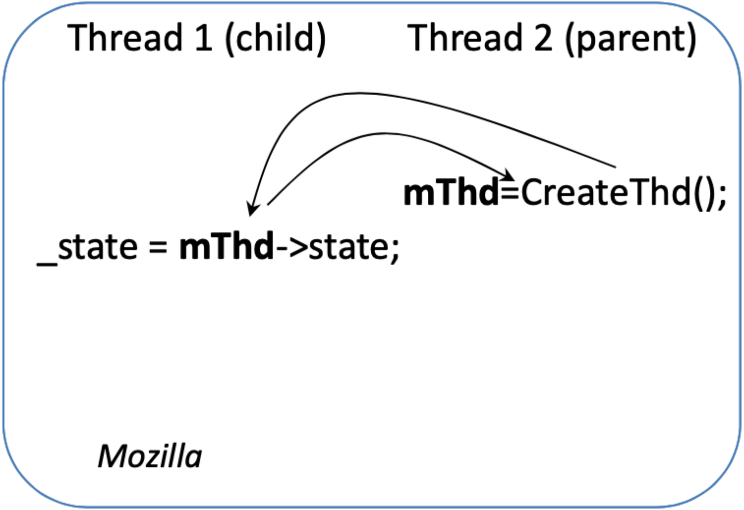
Thread 1

```
while (!flag) {};
```

Thread 2

```
flag=TRUE;
```

Order Violations



Thread 1 (child)

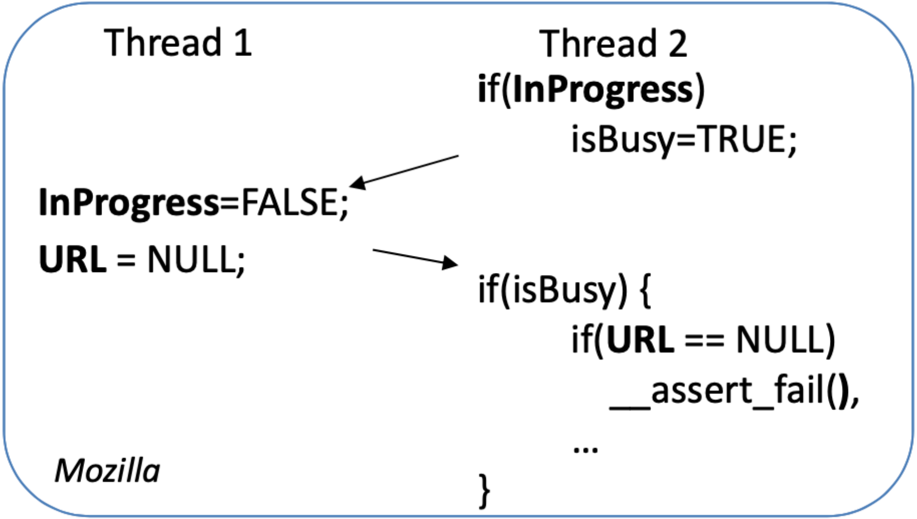
Thread 2 (parent)

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

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

Mozilla

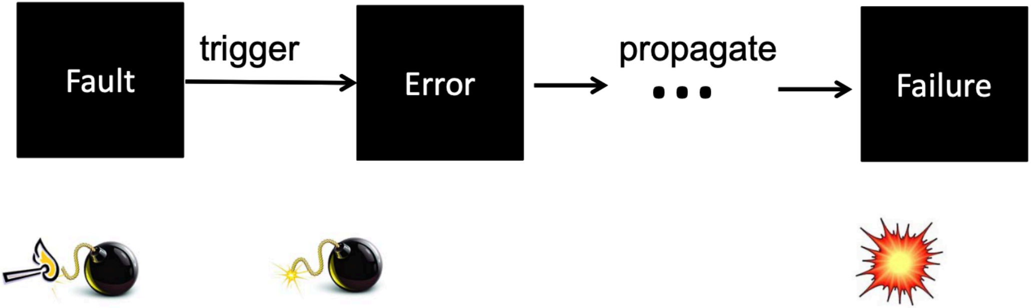
Multi-var Order Violations



```
Thread 1                          Thread 2
                                  if(InProgress)
                                          isBusy=TRUE;

InProgress=FALSE;
URL = NULL;

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

Mozilla

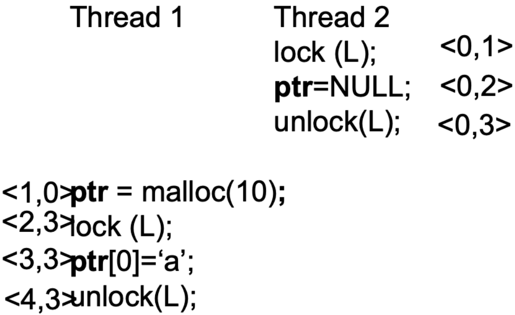
The lifecycle of bugs



Fault → trigger → Error → propagate … → Failure

logical clock algorithm

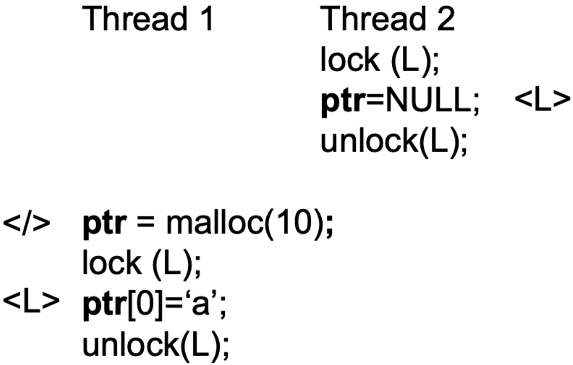
Use logic time-stamps to find concurrent accesses



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

<1,0>ptr = malloc(10);
<2,3>lock (L);
<3,3>ptr[0]='a';
<4,3>unlock(L);
```

Lock-set algorithm

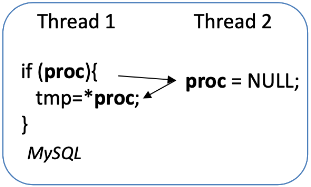
A common lock should protect all conflicting accesses to a shared variable



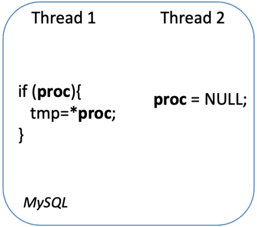
```
       Thread 1               Thread 2
                              lock (L);
                              ptr=NULL;   <L>
                              unlock(L);

</>    ptr = malloc(10);
       lock (L);
<L>    ptr[0]=‘a’;
       unlock(L);
```

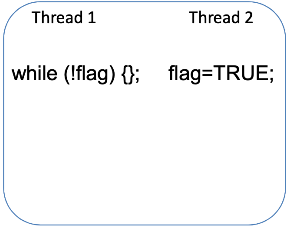
How to detect atomicity-violations?

Know which code region should maintain atomicity



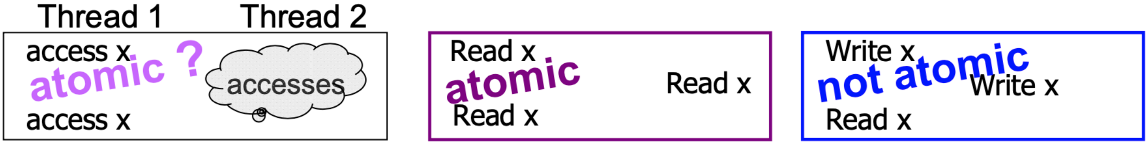
Judge whether a code region’ s atomicity is violated





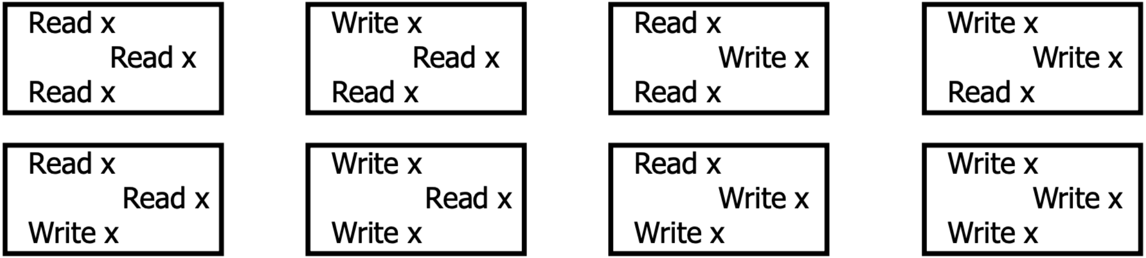
AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants (ASPLOS’ 06)

Atomicity violation = unserializable interleaving



AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants (ASPLOS’ 06)

Totally 8 cases of interleaving



Read x
Read x
Read x

Write x
Read x
Read x

Read x
Write x
Read x

Write x
Write x
Read x

Read x
Read x
Write x

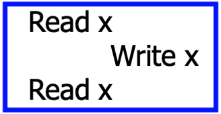
Write x
Read x
Write x

Read x
Write x
Write x

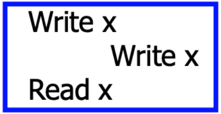
Write x
Write x
Write x

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants (ASPLOS’ 06)

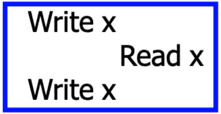
4 out of 8 cases are interleaving violations


Read x
Write x
Read x

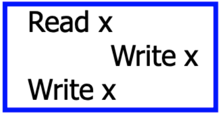
Inconsistent views


Write x
Write x
Read x

Too early overwritten


Write x
Read x
Write x

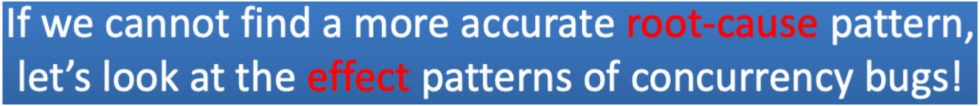
Leaking intermediate value


Read x
Write x
Write x

Using stale value

Both hardware and software solutions exist

ConSeq & ConMem

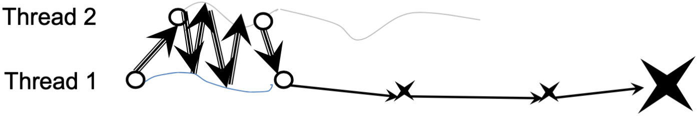


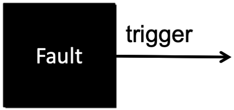
If we cannot find a more accurate root-cause pattern, let's look at the effect patterns of concurrency bugs!

- ConMem
  - Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS' 10
- ConSeq
  - Detecting Concurrency Bugs through Sequential Errors, ASPLOS' 11

The lifecycle of concurrency bugs: Fault

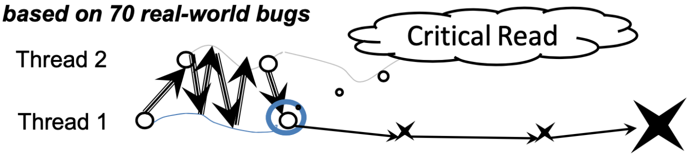
based on 70 real-world bugs



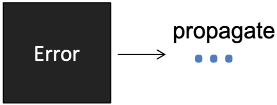


Data races
Atomicity violations
    single variable
    multiple variables
Order violations
...

The lifecycle of concurrency bugs: Error

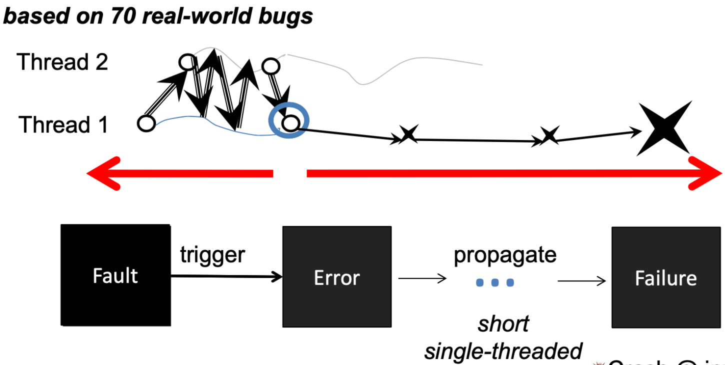


based on 70 real-world bugs



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

The lifecycle of concurrency bugs: Failure



based on 70 real-world bugs

Thread 2

Thread 1

Fault

trigger

Error

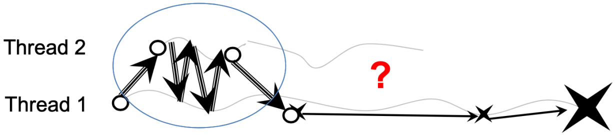
propagate

short
single-threaded

Failure

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

Cause-oriented approach



Interleavings that
match certain **cause** patterns
are buggy

race detectors

Thread 1	Thread 2
	write x
read x	

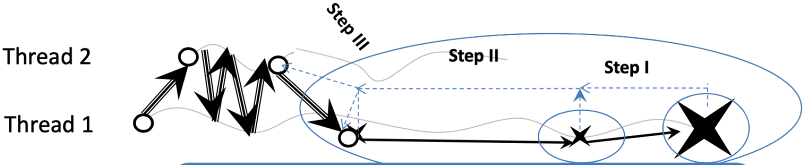
atom. vio. detectors

Thread 1	Thread 2
read x	
	write x
read x	

Limitations

- False positives
- False negatives

Effect-oriented approach

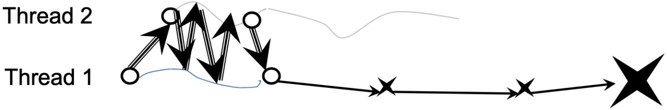


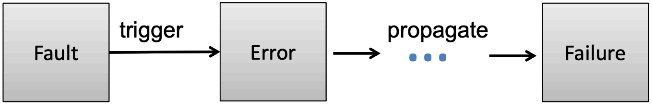
Interleavings that
lead to certain **effect** patterns
are buggy

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

ConMem

Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS’ 10





Fault

trigger

Error

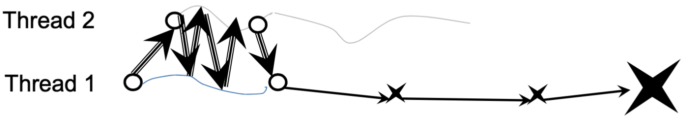
propagate

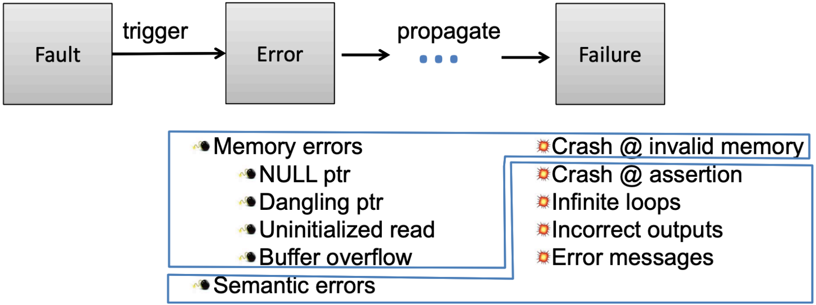
Failure

- 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





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

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