Program Analysis – Lecture 14
Analyzing Concurrent Programs
(Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout (() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2(5);
console.log(z);
```

https://ilias3.uni-stuttgart.de/vote/KN2I
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout (() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2(5);
console.log(z);
```

Result: NaN
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout(() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2(5);
console.log(z);
```

Result: NaN

Executed afterwards, i.e., no effect on return value.
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout (() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2(5);
console.log(z);
```

Result: NaN
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout (() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2(5);
console.log(z);
```

Calling doesn’t wait for promise, i.e., it’s still unresolved

Result: NaN
What does the following JavaScript code print?

async function f1(x) {
    setTimeout (() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

function f2(y, cb) {
    y = f1(y);
    return y * 2;
}

z = f2 (5);
console.log(z);

Result: NaN
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout(() => {
        x = x + 3;
    }, 10);
    return x + 1;
}
async function f2(y, cb) {
    y = await f1(y);
    return y * 2;
}
f2(5).then((z) => {
    console.log(z);
});
```

```javascript
async function f1(x) {
    setTimeout(() => {
        x = x + 3;
    }, 10);
    return x + 1;
}
async function f2(y, cb) {
    y = await f1(y);
    return y * 2;
}
f2(5).then((z) => {
    console.log(z);
});
```
What does the following JavaScript code print?

```javascript
async function f1(x) {
    setTimeout(() => {
        x = x + 3;
    }, 10);
    return x + 1;
}

async function f2(y, cb) {
    y = await f1(y);
    return y * 2;
}

f2(5).then((z) => {
    console.log(z);
});
```

Result: 12
Outline

1. Exploring Interleavings
2. Active Testing to Find Deadlocks

Mostly based on these papers:

- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
- *A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks*, Joshi et al. PLDI 2009
Scheduling Non-Determinism

- A single program executed with a single input may have many different interleavings.
- Scheduler decides interleavings non-deterministically.
- Some interleavings may expose bugs, others execute correctly ("Heisenbugs").
- Challenge: How to explore different interleavings? How to detect buggy interleavings?
CHESS in a Nutshell

- **A user mode scheduler that controls all scheduling non-determinism**

- **Guarantees:**
  - Every program run takes a new thread interleaving
  - Can reproduce the interleaving for every run

- **Systematic but non-exhaustive exploration of the set of possible interleavings**
Tree of Interleavings

- **Search space** of possible interleavings: Represent as a **tree**
  - Node = points of **scheduling decision**
  - Edge = decisions taken
  - Each **path** = one **possible schedule**
Example

```
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```
State Space Explosion

Number of interleavings: $O(n^n \cdot k)$

- Exponential in both $n$ and $k$
- Typically: $n < 10$, $k > 100$

Exploring all interleavings does not scale to large programs (i.e., large $k$)
Preemption Bounding

- Limit exploration to schedules with a small number $c$ of preemptions
  - Preemption = Context switches forced by the scheduler

- Number of schedules: $\mathcal{O}((n^2 \cdot k)^c \cdot n!)$
  - Exponential in $c$ and $n$, but not in $k$

- Based on empirical observation: Most concurrency bugs can be triggered with few (< 2) preemptions
Implementation and Results

- Implemented via binary instrumentation
- Applied to eight mid-size and large systems (up to 175K lines of code),
- Found a total of 27 bugs
- Major benefit over stress testing: Once a failure is detected, can easily reproduce and debug it
Outline

1. Exploring Interleavings
2. Active Testing to Find Deadlocks

Mostly based on these papers:

- Finding and Reproducing Heisenbugs in Concurrent Programs, Musuvathi et al., USENIX 2008
- A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks, Joshi et al. PLDI 2009
Deadlocks

- **Liveness failure** in shared-memory multi-threaded software
- **Set of threads** block forever
  - Each thread is *waiting to acquire a lock held by another thread*
- Relatively easy to detect once reached (in Java)
- **Challenge:** Check if deadlocks may occur
Why Do Deadlocks Exist?

- **Could avoid deadlocks: Follow lock order discipline**
  - Always acquire locks in specific order
  - But: Complex software written by many people
    - Third-party components: Unaware of lock order discipline

- **Fixing data races**
  - Additional synchronization: Increased chance for deadlock
Example

// initialize locks and other state

1 synchronized (L1) {
2 synchronized (L2) {
3 ...
4 }
5 }

6 synchronized (L2) {
7 synchronized (L1) {
8 ...
9 }
10 }
Example

// initialize locks and other state

Thread 1

Thread 2

1 synchronized (L1) {
2 synchronized (L2) {
3 ...
4 }
5 }

6 synchronized (L2) {
7 synchronized (L1) {
8 ...
9 }
10 }

In practice, lock acquisitions may be spread across multiple methods, classes, etc.
Lock graph

nodes = locks

edge = thread wants to acquire target lock while holding source lock

cycle = deadlock
Two-stage dynamic analysis to find deadlocks ("active testing")

- **First stage:** Imprecise dynamic analysis to find potential deadlocks
  - No guarantee that deadlocks really exist

- **Second stage:** Biased random thread scheduler
  - Trigger potential deadlocks with high probability
  - If triggered, guaranteed to exist
Model of Concurrent Programs

- Finite set of threads
  - Each thread: Sequence of labeled statements
- Execution: Sequence of states \( s_0, s_1, \ldots \)
- Events during execution:
  - \( c: \) Acquire \( (c) \) ... acquire lock \( L \) at location \( c \)
  - \( c: \) Release \( (c) \) ... release lock \( L \) at location \( c \)
  - \( c: \) \( o = \text{new} \ (O^*, T) \) ... object \( o \) of type \( T \) allocated
    in method where this resolves to \( o^* \)
Finding Potential Deadlocks

Lock dependency relation of execution:

Set of tuples (t, L, t, C)

- New tuple created whenever some state, thread t acquires lock l while holding all locks in L
- C is sequence of labels of acquire statements
Find lock dependency chain

\((t_1, L_1, L_1, C_1), \ldots, (t_m, L_m, L_m, C_m)\)

where

- \(L_m \in L_n\) \quad \leftarrow\text{potential deadlock cycle}
- \text{threads} \; t_1, \ldots, t_m \text{ are distinct objects}
- \text{locks} \; C_1, \ldots, C_m \text{ are distinct objects}
- \; L_i \in L_{i+1} \text{ for all } i \in \{1, m-1\}
- \; L_i \cap L_j = \emptyset \text{ for all } i, j \in \{1, m\} \text{ and } i \neq j

Each thread could potentially wait to acquire a lock held by the next thread.
Example

- \((t_1, \{3\}, L_1, (\text{line 1}))\)
- \((t_1, \{2, 3\}, L_2, (\text{line 1, line 2}))\)
- \((t_2, \{3\}, L_2, (\text{line 6}))\)
- \((t_2, \{1, 2, 3\}, L_1, (\text{line 6, line 7}))\)

relevant tuples
Computing Lock Dependency Relation

1. Initialize LockSet[t] and Context[t] is empty stacks
   - locks held by thread t
   - labels where currently held locks were acquired

2. Initialize relation D to empty set

3. When thread t executes c: Acquire(l)
   - Push c to Context[t]
   - Add (t, LockSet[t], l, Context[t]) to D
   - Push l to LockSet[t]

4. When thread t executes c: Release(l)
   - Pop from Context[t]
   - Pop from LockSet[t]
Biased Random Scheduler

- Normal scheduler of Java VM: Low chance to trigger a deadlock
- Idea: Control schedule and bias it toward triggering potential deadlocks
  - Focus on specific potential deadlock cycle
  - When thread $t$ is about to acquire a lock $l$ in context $C$, and $t, l, C$ occur in a tuple in the potential deadlock cycle
    - Pause thread before it acquires the lock
// initialize

sync (L1) { 1 ✓
  sync (L2) }
  ↑
  pause 2

sync (L2) { 3 ✓
  sync (L1) }
  ↑
  ↓
  deadlock 4
Challenge: Identifying Objects

- How to identify threads and locks across executions?
  - Addresses of objects don’t match

- Naive approach: Code location where object created
  - But: A single location may create many objects

- Better: Include additional context
  - E.g., sequence of locations \((c_1, \ldots, c_k)\) where object \(c_{i+1}\) is created by object \(c_i\)
Experimental Results

- Implemented for **Java**
- Evaluated on 8 larger programs and 8 collection classes from JDK (600k LoC)
  - 40+ deadlocks found (all true positives, by design)
  - E.g., for Jigsaw program
    - First stage: 283 potential deadlocks
    - Second stage: 29 confirmed deadlocks
Summary

- **Two approaches to control schedule of concurrent programs**
  - **CHESS**: Systematically explore interleavings
  - **Active testing**: Find potential bugs and then bias scheduler toward confirming them
    - Also implemented for data races and atomicity violations