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

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Warm-up Quiz

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);
```

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

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

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

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);
```

Executed afterwards, i.e., no effect on return value

Result: NaN

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

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);
```

`async` function returns a promise

Result: NaN

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

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);
```

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

Result: NaN

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

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);
```

Unresolved promise times two yields NaN

Result: NaN

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz (Fixed)

What does the following JavaScript code print?

```
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);
});
```

Warm-up Quiz (Fixed)

What does the following JavaScript code print?

```
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**

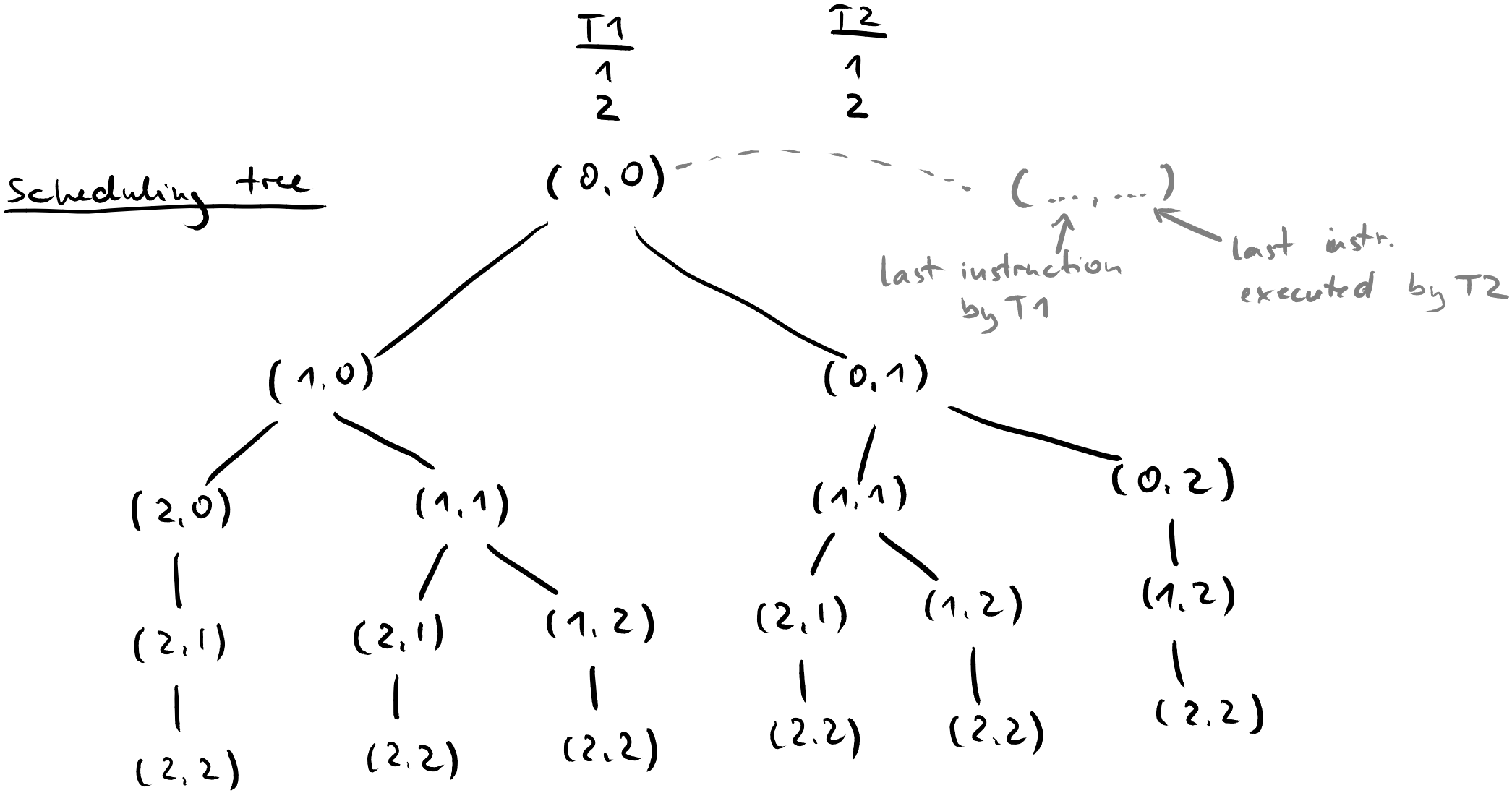
Example

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

Thread 1 Thread 2

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

```
// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```


Scheduling tree
T1
1
2
T2
1
2
(0,0)
(...,...)
last instruction by T1
last instr. executed by T2
(1,0)
(0,1)
(2,0)
(1,1)
(1,1)
(0,2)
(2,1)
(2,1)
(1,2)
(2,1)
(1,2)
(1,2)
(2,2)
(2,2)
(2,2)
(2,2)
(2,2)
(2,2)

State Space Explosion

Thread 1:	Thread 2:
instr. 1	**instr. 1**
instr. 2	**instr. 2**
...	**...**
instr. k	**instr. k**

n threads

k instructions

- Number of interleavings: $\mathcal{O}(n^{n \cdot k})$
- 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
```

Thread 1 | Thread 2

```
synchronized (L1) {
  synchronized (L2) {
    ...
  }
}
```

```
synchronized (L2) {
  synchronized (L1) {
    ...
  }
}
```

Example

```
// initialize locks and other state
```

Thread 1 | Thread 2

```
synchronized (L1) {
  synchronized (L2) {
    ...
  }
}
```

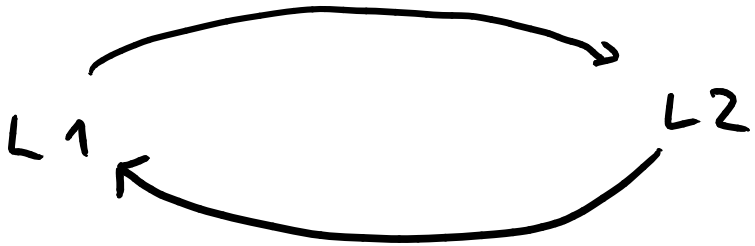
```
synchronized (L2) {
  synchronized (L1) {
    ...
  }
}
```

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

DeadlockFuzzer

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 (ℓ) ... acquire lock ℓ at location c
  - c: Release (ℓ) ... release lock ℓ at location c
  - c: o = 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 $\underbrace{(t, L, \ell, C)}$

- New tuple created whenever in 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

$(t_1, L_1, \ell_1, C_1), \ldots, (t_m, L_m, \ell_m, C_m)$ where

- $\ell_m \in L_1$ ← potential deadlock cycle
- threads $t_1, \ldots, t_m$ are distinct objects
- locks $\ell_1, \ldots, \ell_m$ are distinct objects
- $\ell_i \in L_{i+1}$ for all $i \in [1, m-1]$ ← each thread could potentially wait to acquire a lock held by the next thread
- $L_i \cap L_j = \emptyset$ for all $i, j \in [1, m]$ and $i \neq j$

Example

(t1, {}, L1, (line 1)),

(t1, {L1}, L2, (line 1, line 2))

(t2, {}, L2, (line 6)),

(t2, {L2}, L1, (line 6, line 7))

relevant tuples

Computing Lock Dependency Relation

- Initialize LockSet[t] and Context[t] is empty stacks

  locks held by thread t

  labels where currently held locks were acquired

- Initialize relation D to empty set
- 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]
- 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

+1                                  +2
sync (L1) {  1 ✓                    sync (L2) {  3 ✓
    sync(L2) {  ← pause 2               sync (L1) {  ← deadlock 4
        ⋮                                   ⋮
    }                                   }
}                                   }
```

Challenge: Identifying Objects

- **How to identifiy 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 $(c_1, ..., c_k)$ 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