

# Concurrency Errors

<http://d3s.mff.cuni.cz>

Department of  
Distributed and  
Dependable  
Systems



*Pavel Parízek*



FACULTY  
OF MATHEMATICS  
AND PHYSICS  
Charles University

# Basic taxonomy of concurrency bugs

- Data race condition (unsynchronized access)
- Deadlock caused by incorrectly nested locking
- Deadlock caused by missed notification (early)
- Atomicity violation (inconsistent data values)
- Ordering violation (method calls in two threads)
- Spurious wake-up (forgotten condition check)

# Data race condition

```
Producer.run() {  
    while (true) {  
        synchronized (buf) {  
            buf.add(...);  
        }  
        count++;  
    }  
}
```

```
Consumer.run() {  
    while (true) {  
        if (count > 0) {  
            synchronized (buf) {  
                ... = buf.get(0);  
            }  
        }  
        --count;  
    }  
}
```

```
public static List buf;  
  
main() {  
    (new Producer()).start();  
    (new Consumer()).start();  
}
```

# Deadlock caused by incorrectly nested locks

```
Producer.run() {  
    while (true) {  
        synchronized (coord) {  
            synchronized (buf) {  
                buf.add(...);  
            }  
            count++;  
        }  
    }  
}
```

```
Consumer.run() {  
    while (true) {  
        synchronized (buf) {  
            synchronized (coord) {  
                ... = buf.get(0);  
            }  
            --count;  
        }  
    }  
}
```

```
public static List buf;  
  
main() {  
    (new Producer()).start();  
    (new Consumer()).start();  
}
```

# Deadlock caused by missed notification

```
Subject.run() {  
    ...  
    synchronized (events) {  
        events.add(...);  
        events.notify();  
    }  
    ...  
}
```

```
Observer.run() {  
    ...  
    synchronized (events) {  
        events.wait();  
        ... = events.get(0);  
    }  
    ...  
}
```

```
public static List events = ...  
  
main() {  
    (new Subject()).start();  
    (new Observer()).start();  
}
```

# Atomicity violation

```
Reader.run() {  
    ...  
    synchronized (db) {  
        x = db.value1;  
    }  
    synchronized (db) {  
        y = db.value2;  
    }  
    ...  
}
```

```
Writer.run() {  
    ...  
    synchronized (db) {  
        db.value1 = 10;  
        db.value2 = 20;  
    }  
    ...  
}
```

```
Database db = ...
```

```
main() {  
    (new Reader(db)).start();  
    (new Writer(db)).start();  
}
```

# Ordering violation

```
Server.run() {  
    ...  
    startInit();  
    for (Worker w : workers) {  
        w.start();  
    }  
    finishInit();  
    ...  
}
```

```
Worker.run() {  
    while (true) {  
        waitForRequest();  
        openDatabase();  
        executeDBQuery();  
        processResults();  
        sendResponse();  
    }  
}
```

# Spurious wake-up

```
Producer.run() {  
    synchronized (buf) {  
        while (count >= MAX) {  
            buf.wait();  
        }  
        buf.add(...);  
        count++;  
        buf.notify();  
    }  
}
```

```
Consumer.run() {  
    synchronized (buf) {  
        if (count == 0) {  
            buf.wait();  
        }  
        ... = buf.get(0);  
        --count;  
        buf.notify();  
    }  
}
```

```
public static List buf;  
  
main() {  
    (new Producer()).start();  
    (new Consumer()).start();  
    (new Consumer()).start();  
}
```



# Detecting concurrency bugs

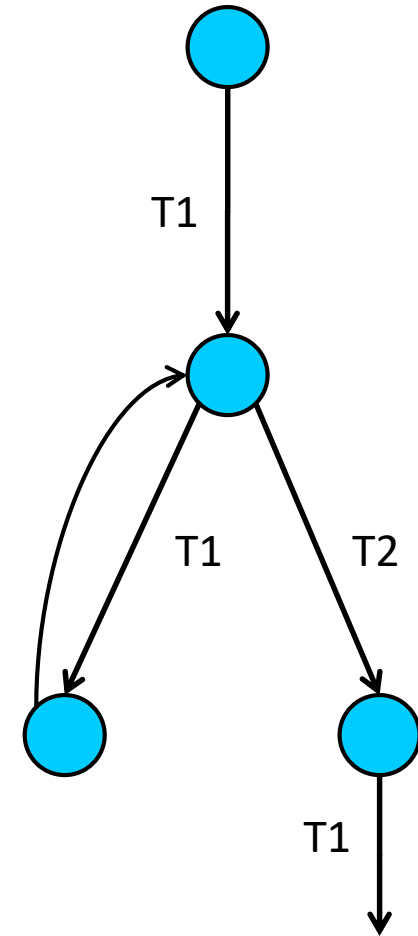


# Detecting concurrency bugs

- Basic approach
  - Exhaustive state space traversal with non-deterministic thread choices by a model checker (JPF)
- Selected variants of state space traversal
  - Using custom runtime to control thread scheduling and synchronization operations
  - Bounding the number of thread preemptions
  - Optimizations (e.g., preemption sealing)
- Other approaches
  - Computing the lock-set analysis
  - Happens-before relation (order)

# Exhaustive state space traversal with thread choices (JPF)

- Single root node
  - Initial program state
- Thread choices
- State matching
- Backtracking



# Using custom runtime

- Controls thread scheduler in the operating system
- Custom library for synchronization primitives
  - source code instrumentation, dynamic linking
- Tracking execution of statements accessing the global state (heap objects, locks)
  - source code instrumentation, dynamic monitoring

# Executing program with different schedules

- Restart program execution many times
  - Each time with a different thread interleaving
- Keep track of explored thread schedules
- Stateless traversal
  - no set of visited states, no state matching

# Bounded number of preemptions

- Motivation: errors triggered with few thread preemptions (2-5) and few threads (2)
  - General principle: *small scope hypothesis*
- Limit the number of thread preemptions
- Systematic exploration within the given bound
- Common alternative name: context bounding

# Bounded number of preemptions

- Motivation: errors triggered with few thread preemptions (2-5) and few threads (2)
  - General principle: *small scope hypothesis*
- Limit the number of thread preemptions
- Systematic exploration within the given bound
- Common alternative name: context bounding

# Bounded number of preemptions

- Method limitations
  - Ignores concurrency errors triggered by more context switches (preemptions)
  - Checks program behavior only for a single input
    - Remedy: **symbolic execution**
- Theoretical complexity: NP-complete



# Preemption sealing

- Disable thread choices in
  - System libraries (e.g., core and collections)
  - Already explored state space fragments
    - Method tested during previous runs of the checker
    - Code triggering already known concurrency bugs

# CHESS: Systematic Concurrency Testing

- Main features
  - Custom runtime with scheduler
  - Stateless traversal with fairness
  - Iterative context-bounding
- Supported platforms
  - C#, C/C++, Win32, .NET
  - Probably just 32-bit CPU
- Further information & source code
  - <https://www.microsoft.com/en-us/research/project/chess-find-and-reproduce-heisenbugs-in-concurrent-programs>

# COYOTE: Concurrency Unit Testing

- Main features
  - Unit tests written in C# running multiple threads
  - Exploration strategies over possible interleavings
  - Debugging: reproduces errors, visualizing traces
- Target platform
  - Recent .NET frameworks on Windows/Linux
- Further information and source code (binaries)
  - <https://www.microsoft.com/en-us/research/project/coyote/>
  - <https://microsoft.github.io/coyote/>

# Context bounding done another way

- Transforming concurrent programs to sequential programs
  - Approach: source-to-source translation

# Context bounding done another way

- Transforming concurrent programs to sequential programs
  - Approach: source-to-source translation
- Model checking the sequential program
- Thread preemption
  - non-deterministic data choice
  - jump to another code location
  - set up execution context (stack)
- Program state: cross-product of local variables of all threads and global variables

# Lock-set analysis

- Find the set of locks held at each access to a shared global variable
- Check whether accesses to shared variables follow a consistent locking discipline
- Two concurrent accesses to a global variable
  - Empty intersection of lock sets → data race
- Every access to a shared variable protected by the same lock
  - Thread using a different lock than before → data race

# Happens-before ordering (relation)

- Relationships between synchronization events
  - causal, temporal, execution flow
- Partial happens-before ordering
- Example 1: wait – notify
- Example 2: lock release – lock acquire
- Ordering between field accesses → no data race

# Defining correctness of concurrent programs





# Correctness conditions

- Example: LinkedList
  - Operations: add(o), get(i), remove(i), size()
- Data race freedom
- Serializability (atomicity)
  - No overlap between concurrent actions
- **Linearizability**

# Linearizability

- Concurrent history  $H$ 
  - Operation: invoke, result
  - Partial order:  $e_1 <_H e_2$  if  $\text{res}(e_1)$  precedes  $\text{inv}(e_2)$
- Linearizable concurrent history  $H$ 
  - Exists serial witness that respects partial order and every operation has the same result value as in  $H$
- Set of concurrent operations
  - Every possible concurrent history is linearizable with respect to a sequential specification

# Verifying linearizability

- Linearization points
  - Operations must appear to take their effect at some instant between the call and return
- State space traversal
  - Phase 1: find all possible sequential histories
  - Phase 2: explore concurrent histories
    - Identify corresponding serial witness for each
- More complicated algorithmic techniques

# Relaxed memory models



# Relaxed memory models

- Defines valid program transformations
  - System: compiler, virtual machine, hardware
- Motivation: optimizing performance
- Possible transformations
  - Reordering write accesses to a shared variable in a given thread
  - Delaying propagation of the new value of a global variable to other threads (shared memory)

# Relaxed memory models

- Sequential consistency
- Data race free models
- Case study: Java Memory Model
- Case study: C++11 Memory Model
  - Various extensions: C++14/17/20

# Sequential consistency

- Memory accesses execute one at a given time
- Total order of memory accesses (read, write)
- Reads observe the most recent written value
- Each thread must respect the program order
  - Order defined by the source code (developer)

# Java Memory Model

- Data race free programs behave correctly
  - Guaranteed sequentially consistent semantics
- Program with data races → up to the developer
  - Model provides only weak guarantees
- Memory barriers
  - Boundaries of `synchronized` blocks
  - Accessing `volatile` variables
- Defined formally using the happens-before ordering
  - Very complex (many rules): lot of research papers about it
- Used since J2SE 5.0



# Hardware memory models

- Total Store Order (TSO)
  - Delaying writes (stores) relative to subsequent reads (loads) on the same processor
  - CPU architecture: x86
- Partial Store Order (PSO)
  - Additionally, delaying stores relative to other stores (to different memory locations) on the same processor
- Partial Store Load Order (PSLO)
  - Additionally, permits reordering loads to execute before previous loads and stores on the same processor

# Relaxed memory models: verification support

- Java PathRelaxer
  - CHES: limited
  - COYOTE: not sure
- 
- Some tools for checking program behavior on hardware memory models (especially TSO)

# Data races

- Benign
  - Optimizing performance on multi-core CPUs
  - Exploiting properties of the memory model
  - Very hard to get the implementation right
  - Case study: `java.util.concurrent`
- Erroneous
  - Missing thread synchronization by a developer mistake
- Some people call for a “total ban” on data races

# ABA problem



# ABA problem

- Idea: same value but something changed
- Typical for lock-free data structures

# Further reading

- M. Musuvathi and S. Qadeer. **Iterative Context Bounding for Systematic Testing of Multithreaded Programs**. PLDI 2007
- M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. **Finding and Reproducing Heisenbugs in Concurrent Programs**. OSDI 2008
- P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal. **Industrial-Strength Controlled Concurrency Testing for C# Programs with COYOTE**. TACAS 2023
- S. Qadeer and D. Wu. **KISS: Keep it Simple and Sequential**. PLDI 2004
- N. Ghafari, A. Hu, and Z. Rakamaric. **Context-Bounded Translations for Concurrent Software: An Empirical Evaluation**. SPIN 2010
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**. ACM Transactions on Computer Systems, 15(4), 1997
- S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. **Line-Up: A Complete and Automatic Linearizability Checker**. PLDI 2010
- J. Manson, W. Pugh, and S.V. Adve. **The Java Memory Model**. POPL 2005