

# Verification of concurrent programs

16 November 2016

# Errors in concurrent programs

*Deadlock*

*Livelock* (loop without useful progress)

*Starvation*: inequitable resource access  
(threads that do not get access, though no deadlock overall)

*Race conditions*

in particular, data races

*Not observing atomicity*

simple source code statements (++) may not be atomic in  
machine code

variables covering several memory words (non-atomic writes)

## Synchronization primitives

Concurrent programs have synchronization primitives  
but how are they implemented ?

e.g. with hardware support: test\_and\_set instruction

```
// busy wait
// returns old value of lock
// sets it to 1 if it was 0
while (test_and_set(lock) == 1);
```

more general: compare-and-swap

```
int cmpxchg(int *x, int new, int old) { // atomic
    int current = *x;
    if (current == old) *x = new;
    return current; // change done iff it returns old
}
```

## Mutual exclusion: Peterson's algorithm

```
while (1) {  
    L1: flag[0] = true; // try  
    L2: turn = 1; // other's turn  
    L3: while (flag[1] && turn==1)  
        ; // wait  
    C0: flag[0] = false;  
}
```

```
while (1) {  
    R1: flag[1] = true; //try  
    R2: turn = 0; // other's turn  
    R3: while (flag[0] && turn==0)  
        ; // wait  
    C1: flag[1] = false;  
}
```

Designed for single-processor shared memory

Not safe in a multicore setting (will discuss)

## Data races

Happen when two threads access a variable, and  
at least one does a write access  
the threads are not explicitly synchronized

Analyzing race conditions is complicated by *reorderings within a thread* (through compiler optimizations)

init: $x = 0; y = 0;$	Possible outcomes $(r1, r2)$ :	$(0, 0)$
t1: $r1 = x;$	t2: $r2 = y;$	$(1, 0)$
$y = 2;$	$x = 1;$	$(0, 2)$

But by reordering in t1 and t2 we could obtain  $r1 = 1, r2 = 2$  !

This result does not match *sequential consistency*  
(that we are intuitively used to)

all memory accesses correspond to *total order* (linear), and  
order of accesses in any thread is *program order*

# Why are concurrent programs hard to verify?

Understanding concurrency problems is often hard

Difficult to exercise a certain execution sequence  
needs control over/changes to scheduler/external conditions

Error traces might be very rare (in certain complex scenarios)

Error conditions may be hard to reproduce (“Heisenbugs”)

Exhaustive exploration of all execution traces is infeasible  
quad (exponential in number of threads / their size)

## Error patterns in concurrent programs

[Farchi, Nir, Ur: Concurrent bug patterns and how to test them, 2003]

### *Ignoring non-atomicity*

`x = 0` || `x = 0x101`  $\Rightarrow$  `x == 1` possible!!

if the bytes are written separately (hi from 0, low from 0x101)

### *Two-step access*

even if accesses protected, object may change in between

```
lock(); idx = table.find(key); unlock();  
if (...) { lock(); table[idx] = newval; unlock(); }
```

### *Missing / wrong lock* (e.g. programmer unfamiliar with code)

```
t1: synchronized(o1) {n++;}    t2: n++; // not sync
```

or

```
t1: synchronized(o1) {n++;}    t2: synchronized(o2) {n++;}
```

## Error patterns in concurrent programs (cont.)

*Double-checked locking*: “optimizing” on-demand initialization

```
class Foo {
    private Helper helper = null;
    public Helper getHelper() { // to avoid some synchronization
        if (helper == null)    // already allocated? return
            synchronized(this) {
                if (helper == null) // second check is protected
                    helper = new Helper();
            }
        return helper; // other thread may see incomplete object
    }
}
```

Problem: compiler is free to reorder for optimization



## Error patterns in concurrent programs (cont.)

*Situations assumed impossible* (but which may happen):

sleep() wrongly used to *guarantee* a delay

*Lost Notify*: when executed *before* wait:

```
t1: synchronized(o) { o.wait(); }
```

```
|| t2: synchronized(o) { o.notifyAll(); }
```

*Unchecked Wait*: on resume, must check awaited condition  
(resume might have happened due to other causes)

*Deadlock scenarios*

code written assuming the critical section won't block

false, if (bad) code provided by someone else

“orphan” threads

if creator thread terminates with error  $\Rightarrow$  may lead to deadlock

## Java memory model

A concurrent language must have a memory model that is *intuitive*, and which does *not limit performance*, by restricting optimizations

Solution [JSR 133; Manson, Pugh, Adve, PLDI'05]:

define a class of *well-synchronized* programs (*data race free*) for which *sequential consistency* is ensured

+ minimal guarantees for rest of programs (even if incorrectly synchronized)

Principle

define a *happens-before* order [Lamport] between program actions: *transitive closure* of

- a) *ordering of synchronization actions* (b/w any unlock and lock on same monitor, and b/w write and read on a volatile variable)
- and b) *program order* (between execution threads)

## Volatile variables and synchronization

Reading a *volatile* variable:

last value written in synchronization order

Reading a non-volatile variable:

any value which is *not written later* according to *happens-before*  
and is not obsoleted by another write

# Volatile variables and synchronization

Reading a *volatile* variable:

last value written in synchronization order

Reading a non-volatile variable:

any value which is *not written later* according to *happens-before* and is not obsoleted by another write

Warning: *volatile* does NOT mean *atomic* !

Race condition =

conflicting accesses (r-w, w-w) not ordered by *happens-before*.

Well-synchronized program = does not have race conditions

## Unit testing solutions

Implicitly, JUnit observes thread that launched the test  
⇒ does not detect exceptions in threads launched later  
⇒ need frameworks with features adapted to concurrency

Various JUnit additions, e.g. ConcJUnit [Rice University]  
creates/observes a *group* of execution threads  
warns if other threads still running after main thread completes  
(should have been handled with a `join ...`)  
may insert arbitrary delays ⇒ generates other interleavings

RunnerScheduler (experimental API addition)

## Solutions for system-level testing

Idea: create variation in thread scheduling

ConTest [IBM Haifa]

instruments program (`sleep()`, `yield()`, etc.)

or simulates delays, message loss, etc.

⇒ random or guided variation in scheduling

measures coverage with respect to all possible  
schedules/interleavings

CHES [Microsoft Research]

captures calls to synchronization functions

*systematically* generates executions with new schedules

in increasing order of preemption count

can *reproduce* generated executions

## Detecting race conditions

Many proposed solutions. One widely used algorithm: Eraser [1997]

combines static and dynamic analysis

by analyzing *one* execution finds *potential* errors in others

keeps track of locks acquired by each thread

tries to derive which lock protects which shared object

```
init:    C(v) = all_locks;    // for each variable v
access:  C(v) = C(v) ∩ locks_held(t);    // on access by t
        if (C(v) = ∅) warning();    // unprotected access!
```

If extended, may distinguish read and write locks, tracking the state of each variable (virgin, exclusive, shared, shared-modified)

Conservative algorithm, may give false alarms for correct programs (which do not associate a variable with a unique lock throughout)

## High-level data races

[Artho, Havelund, Biere 2003]

Errors: when *granularity* of protected variables not same over time

```
void swap() {  
    int lx, ly;  
    synchronized(this) {  
        lx = this.x;  
        ly = this.y;  
    }  
    synchronized(this) {  
        this.x = ly;  
        this.y = lx;  
    }  
}  
  
void reset() {  
    synchronized(this) {  
        this.x = 0;  
    }  
    synchronized(this) {  
        this.y = 0;  
    }  
}
```

Member access synchronized, but swap and reset may interfere!

⇒ Need analysis not just for variables (what locks protect them?)  
but also starting from locks (what variable sets covered by each?)



# Java Pathfinder [NASA]: Model checking for concurrency

Completely explores program executions

- simulates nondeterminism through a custom virtual machine which allows choosing scheduling variants at each step and returning to unexplored ones (similar to backtracking)

Works at bytecode level; allows to check

- deadlocks

- exceptional conditions

- assertions in code

Limited to smaller programs (10 kloc): “state space explosion”

- size of stored states (number of program variables)

- number of possible traces (exponential in number of threads)

# What are developers doing in practice?

Lu, Park, Seo, Zhou: Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics, ASPLOS'08

Research Questions:

- what kinds of real bugs can be detected?

- are assumptions valid ? e.g. focus on single-variable access

- how helpful are tools in diagnosing and fixing ?

## Findings on Bugs [Lu et al.'08]

105 randomly selected real world concurrency bugs

74 non-deadlock bugs + 31 deadlock bugs

4 large open-source programs:

Apache, Mozilla, MySQL, OpenOffice

97% two patterns: atomicity or order violation

latter not well addressed by tools

97% two threads, circular wait

96% reproducible w/ partial order between 2 threads

92% order between  $\leq 4$  memory accesses

$\Rightarrow$  suggests *handling common cases is effective*

## Findings on Bugs [Lu et al.'08]

66% involved only one variable

22% caused by one thread acquiring resource held by itself

73% of non-deadlock bugs fixed *not* by adding locks

61% fix: prevent thread to acquire a lock; may cause other bugs

Transactional memory could avoid 39% of bugs

+ 42% more by addressing some concerns (I/O, atomic GC)

# Study in Software Maintenance Community

R. Xin et al., An Automation-assisted Empirical Study  
on Lock Usage for Concurrent Programs, ICSM 2013

4 programs: Aget, Apache httpd, MySQL, Pbzip2, up to 786Kloc

Issues to study:

- (language) characteristics of lock usage (function/lock counts)
- lock usage patterns
- lock usage evolution

## Findings [Xin et al., ICSM'13]

- ▶ 80% of the lock related functions acquire only one lock
- ▶ simple lock patterns account for 55% of all lock usage
- ▶ only 12 out of 527 detected patterns are conditional  
(more error-prone)
- ▶ only 0.65% of functions are lock related

## What do practitioners use?

Wojkicki & Strooper, A State-of-Practice Questionnaire on Verification and Validation for Concurrent Programs, PADTAD'06

35 survey respondents, Java development

Relevant defects: deadlock, interference (> 80%), starvation (50%)

Techniques: code inspection, junit test (> 80%)  
static analysis (50%, mostly FindBugs), code coverage,  
model checking (20%)

## How good are the tools used in practice?

Kester, Mwebesa, Bradbury (SCAM 2010):

How Good is Static Analysis at Finding Concurrency Bugs?

used 12 benchmarks from Java PathFinder and IBM ConTest

evaluated 3 tools: FindBugs, JLint, Chord

recall: 30-33 % of actual known bugs

precision: 100% (Chord), 78% (JLint), 31% (FindBugs)

Threat to validity: small-scale evaluation (13 bugs)



## Developer Study at Google

Sadowski & Yi. How Developers Use Data Race Detection Tools. SPLASH/PLATEAU'14

Two data race analysis mechanisms: `THREADSAFETY` and `TSAN`

`THREADSAFETY`: static, annotation-based, implemented in Clang led to 18 bug-fixing commits (1 month) in small section of code

`TSAN` (ThreadSanitizer) : dynamic identification of data races

- TSan v1 – Valgrind, 20-300x slowdown

- TSan v2 – LLVM, happens-before, 5-15x slowdown,

`TSAN` in 30 min. found Chrome bug hunted for 6 months

## Usage in Google development teams

Team A: `THREADSAFETY` for docs, nightly runs of `TSAN`  
find 1 race per 10 weeks

Team B: added annotations to all core libraries  
ensures annotation for all mutexes (automatically searched)

Team C: stable synch. code, no payoff for `THREADSAFETY`,  
not heard of `TSAN`

Team D: `THREADSAFETY` for tricky code, not heard of `TSAN`

## Google study findings

Reproducibility & low false positives are important

Team culture matters

Tradeoff: races vs. deadlocks (*crash is easy, inconsistency is hard*)

Manual inspection is implicit comparison point

Good docs important for building mental models

Limitations: slow speed and lack of coverage (TSAN),  
difficulty of annotation (THREADSAFETY)