Verification of concurrent programs

19 November 2015

Errors in concurrent programs

Deadlock

Livelock (loop without useful progress)

Starvation: inequitable resource access
(threads that do not get access, without deadlock overall)

Race conditions
in particular, data races

Not observing atomicity
simple source code statements (++) may not be atomic in machine code
or: writes to variables covering several memory words

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

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;`

t1: `r1 = x;` `y = 2;`

t2: `r2 = y;` `x = 1;`

Possible outcomes (r1, r2): (0, 0) (1, 0) (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*

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 the rest of programs (even if incorrectly synchronized)

Principle: define a *happens-before* ordering [Lamport] between program actions, which transitively combines
ordering of synchronization actions (b/w unlock and any lock on the same monitor, and between writing a volatile variable and reading it)
and *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

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
 (exponential in number of threads / their size)

Error patterns in concurrent programs

[after Farchi, Nir, Ur – IBM Haifa]

Ignoring non-atomicity

`x = 0` || `x = 0x101` ⇒ `x == 1` is possible!!

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

Two-step access

even if both accesses are 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++;`

or

t1: `synchronized(o1) { n++; }`   t1: `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() {  // tries to avoid some synchroniz
    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 ⇒ may lead to deadlock

Unit testing solutions

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

e.g.: ConcJUnit [Rice University]
creates/observers 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 $\Rightarrow$ generates other interleavings

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

CHESS [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 solutions have been proposed. One of the classic ones: 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: $C(v) = \mathit{all\_locks}$; // for each variable v
access: $C(v) = C(v) \cap \mathit{locks\_held}(t)$; // on access by t
    if $C(v) = \emptyset$ 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 lead to false alarms for correct programs
(which do not associate a variable with a unique lock throughout execution)

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;
  }
}
```

Access to members is synchronized, but `swap` and `reset` may interfere!

⇒ Analysis not just from point of view of 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 executions (exponential in number of threads)