ThreadSanitizer
Data race detection in practice

Konstantin Serebryany <kcc@google.com>

Sep 16, 2010

Data races are scary

A data race occurs when two or more threads concurrently access a shared memory location and at least one of the accesses is a write.

```
void Thread1() {
   x[123] = 1;
}
```

```
void Thread2() {
   x[456] = 2;
}
```

Data races are scary

A data race occurs when two or more threads concurrently access a shared memory location and at least one of the accesses is a write.

```
std::map<int,int> x;

void Thread1() {
   x[123] = 1;
}

void Thread2() {
   x[456] = 2;
}
```

Our goal: find races in Google code

Dynamic race detector

- Intercepts program events at run-time
  - Memory access: READ, WRITE
  - Synchronization: LOCK, UNLOCK, SIGNAL, WAIT
- Maintains global state
  - Locks, other synchronization events, threads
  - Memory allocation
- Maintains shadow state for each memory location (byte)
  - Remembers previous accesses
  - Reports race in appropriate state
- Two major approaches:
  - LockSet
  - Happens-Before

LockSet

```
void Thread1() {
  mu1.Lock();
  mu2.Lock();
  *X = 1;
  mu2.Unlock();
  mu1.Unlock(); ...
```

```
void Thread2() {
  mu1.Lock();
  mu3.Lock();
  *X = 2;
  mu3.Unlock();
  mu1.Unlock(); ...
```

- LockSet: a set of locks held during a memory access
  - Thread1: {`mu1`, `mu2`}
  - Thread2: {`mu1`, `mu3`}
- Common LockSet: intersection of LockSets
  - {`mu1`}

LockSet: false positives

```
void Thread1() {
  x->Update(); // LS={}
  mu.Lock();
  queue.push(x);
  mu.Unlock();
}
```

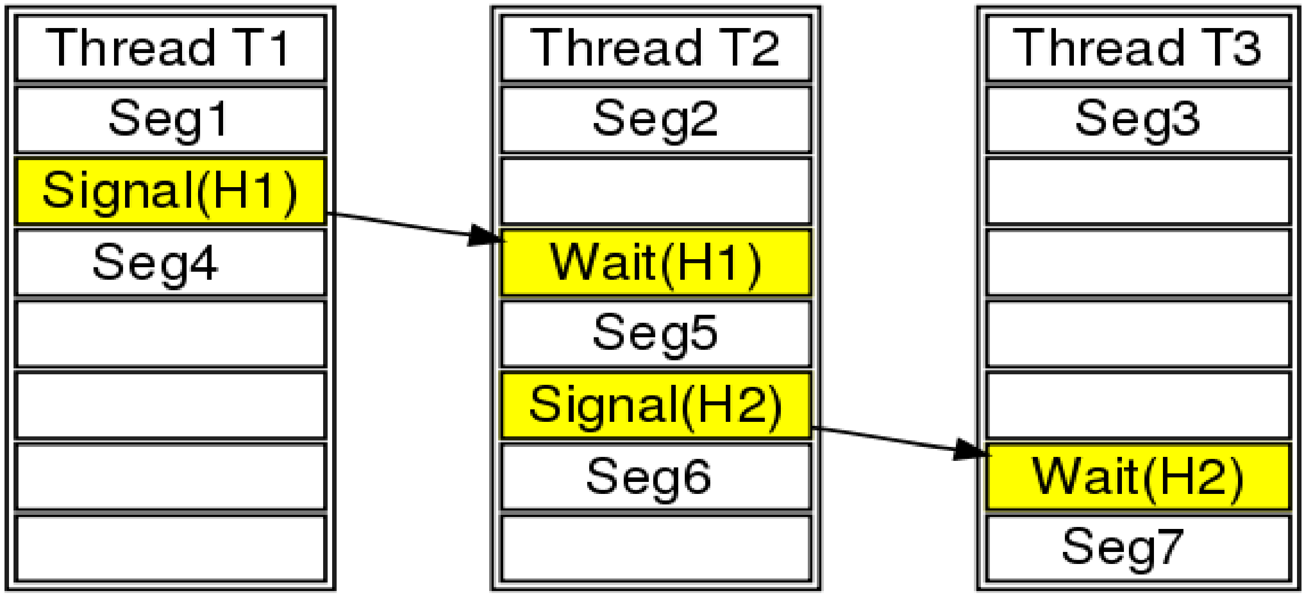
```
void Thread2() {


  Obj *y = NULL;
  mu.Lock();
  y = queue.pop_or_null();
  mu.Unlock();
  if (y) {
    y->UseMe(); // LS={}
  }
}
```

Google™

Happens-before

partial order on all events



Segment: a sequence of READ/WRITE events of one thread.
Signal(obj) ⟹ Wait(obj) is a happens-before arc

Seg1 $\prec$ Seg4 -- segments belong to the same thread.

Seg1 $\prec$ Seg5 -- due to Signal/Wait pair with a macthing object.

Seg1 $\prec$ Seg7 -- happens-before is transitive.

Seg3 $\not\prec$ Seg6 -- no ordering constraint.

Pure happens-before: misses races

```
void Thread1() {
  x = 1;
  mu.Lock();
  do_something1();
  mu.Unlock();
}
```

```
void Thread2() {
  do_something2();
  // do_something2()
  // may take
  // lots of time
  mu.Lock();
  do_something3();
  mu.Unlock();
  x = 2;
}
```

Google™

Happens-before vs LockSet

- Pure LockSet
  - Many false warnings
  - Does not miss races, fast
- Pure happens-before detectors:
  - Unlock ⇨ Lock is a happens-before arc
  - No false positives
    - unless you use lock-free synchronization
  - Less predictable, miss many races (30% - 50%)
- Hybrid (happens-before + LockSet):
  - Lock/Unlock don't create happens-before arcs
  - Have false positives (easy to annotate)
  - More predictable, find more races

Quiz: do we have a race?

```
static Mutex mu;
static bool flag = flag_init_value;
static int var;
void Thread1() {  // Runs in thread1.
   var = 1;  // << First access.
   mu.Lock();
   flag = true;
   mu.Unlock();
}
void Thread2() {  // Runs in thread2.
   bool f;
   do {
     mu.Lock();
     f = flag;
     mu.Unlock();
   } while(!f);
   var = 2; // << Second access.
}
```

Dynamic annotations

```
void Thread1() {
  x->Update(); // LS={}
  mu.Lock();
  queue.push(x);
  ANNOTATE_HAPPENS_BEFORE(x);
  mu.Unlock();
}
```

```
void Thread2() {
  Obj *y = NULL;
  mu.Lock();
  y = queue.pop_or_null();
  ANNOTATE_HAPPENS_AFTER(y);
  mu.Unlock();
  if (y) {
    y->UseMe(); // LS={}
  }
}
```

ThreadSanitizer: Algorithm

- Segment: a sequence of READ/WRITE events of one thread
  - All events in a segment have the same LockSet
- Segment Set: a set of segments none of which happen-before any other
- Shadow state:
  - Writer Segment Set: all recent writes
  - Reader Segment Set: all recent reads, unordered with or happened-after writes
- State machine: on each READ/WRITE event
  - update the Segment Sets
  - check accesses for race

ThreadSanitizer: Algorithm

HANDLE-READ-OR-WRITE-EVENT($IsWrite, Tid, ID$)

$(SS_{Wr}, SS_{Rd}) \leftarrow$ GET-PER-ID-STATE(ID)

$Seg \leftarrow$ GET-CURRENT-SEGMENT(Tid)

if $IsWrite$

then ▷ WRITE event: update SS_{Wr} and SS_{Rd}

$SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \wedge s \not\preceq Seg\}$

$SS_{Wr} \leftarrow \{s : s \in SS_{Wr} \wedge s \not\preceq Seg\} \cup \{Seg\}$

else ▷ READ event: update SS_{Rd}

$SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \wedge s \not\preceq Seg\} \cup \{Seg\}$

SET-PER-ID-STATE(ID, SS_{Wr}, SS_{Rd})

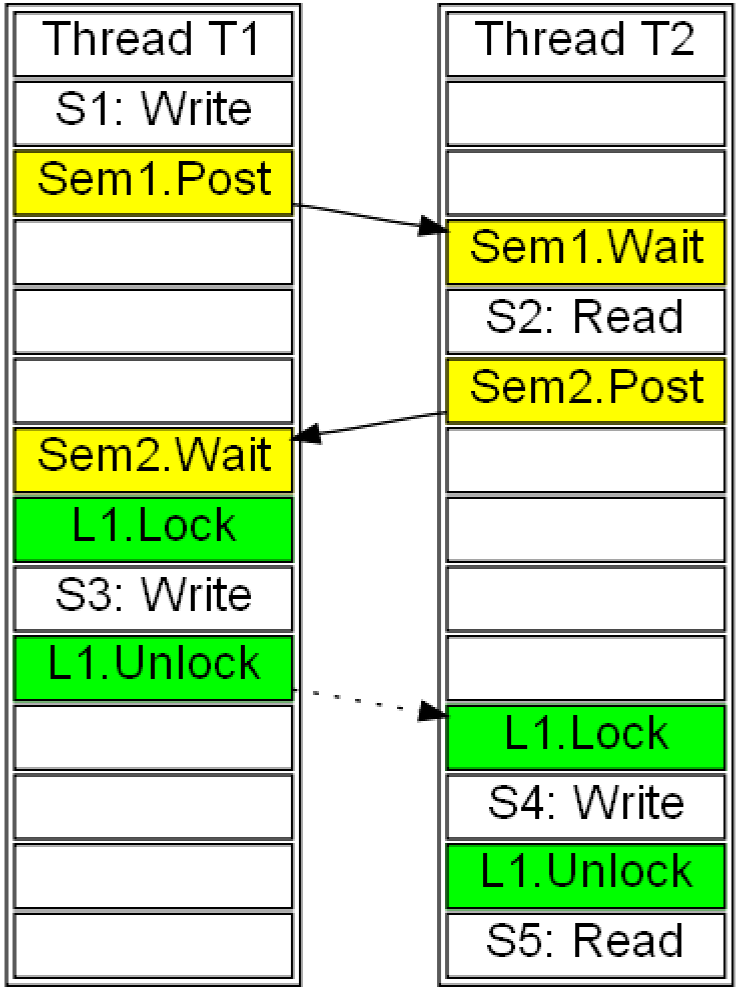
if IS-RACE(SS_{Wr}, SS_{Rd})

then REPORT-RACE($IsWrite, Tid, Seg, ID$)

Example

```
// Thread1
X = ...;
Sem1.Post();
Sem2.Wait();
L1.Lock();
X = ...;
L1.Unlock();
```

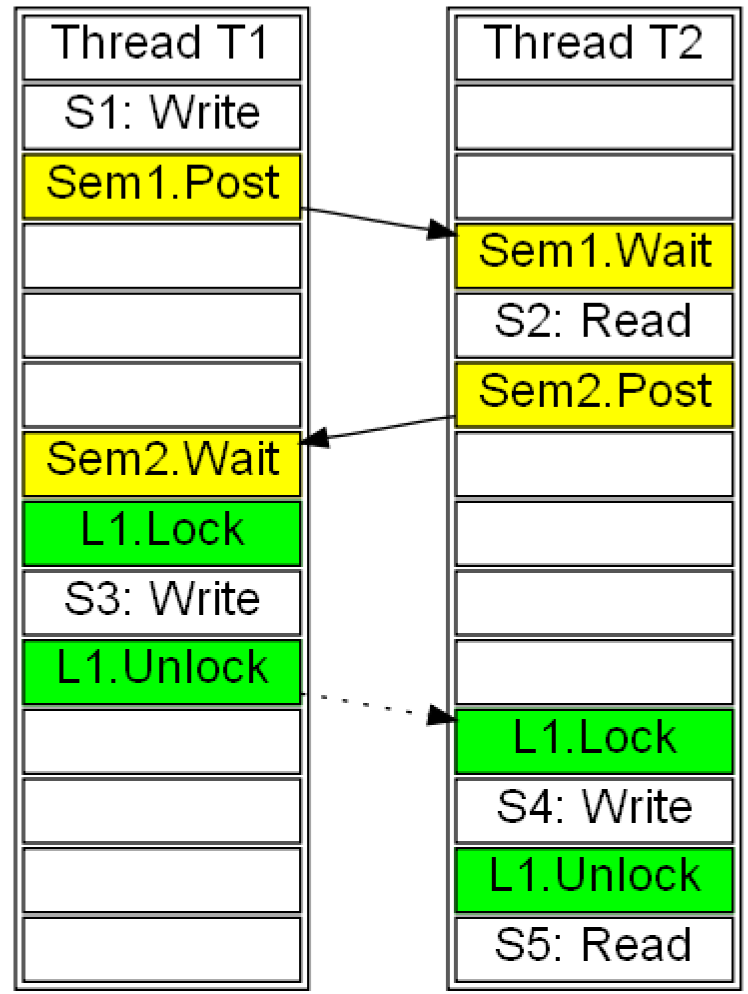
```
// Thread2
Sem1.Wait();
... = X;
Sem2.Post();
L1.Lock();
X = ...;
L1.Unlock();
... = X;
```



Example: shadow state

Hybrid	
Writer SS	**Reader SS**
S1	-
S1	S2
S3/L1	-
S3/L1, S4/L1	-
S3/L1, S4/L1	**S5 {Race!}**

Pure Happens-before	
Writer SS	**Reader SS**
S1	-
S1	S2
S3/L1	-
S4/L1	-
S4/L1	S5



Report example

```
WARNING: Possible data race during write of size 4 at 0x633AA0: {{{
  T2 (test-thread-2) (locks held: {L122}):
    #0 test301::Thread2() racecheck_unittest.cc:5956
    #1 MyThread::ThreadBody(MyThread*) thread_wrappers_pthread.h:320
    #2 ThreadSanitizerStartThread ts_valgrind_intercepts.c:387
 Concurrent write(s) happened at (OR AFTER) these points:
  T1 (test-thread-1) (locks held: {L121}):
    #0 test301::Thread1() racecheck_unittest.cc:5951
    #1 MyThread::ThreadBody(MyThread*) thread_wrappers_pthread.h:320
    #2 ThreadSanitizerStartThread ts_valgrind_intercepts.c:387
 Address 0x633AA0 is 0 bytes inside data symbol "_ZN7test3013varE"
 Locks involved in this report (reporting last lock sites): {L121, L122}
  L121 (0x633A10)
    #0 pthread_mutex_lock ts_valgrind_intercepts.c:602
    #1 Mutex::Lock() thread_wrappers_pthread.h:162
    #2 MutexLock::MutexLock(Mutex*) thread_wrappers_pthread.h:225
    #3 test301::Thread1() racecheck_unittest.cc:5950
    #4 MyThread::ThreadBody(MyThread*) thread_wrappers_pthread.h:320
    #5 ThreadSanitizerStartThread ts_valgrind_intercepts.c:387
  L122 (0x633A70)
    #0 pthread_mutex_lock ts_valgrind_intercepts.c:602
    #1 Mutex::Lock() thread_wrappers_pthread.h:162
    #2 MutexLock::MutexLock(Mutex*) thread_wrappers_pthread.h:225
    #3 test301::Thread2() racecheck_unittest.cc:5955
    #4 MyThread::ThreadBody(MyThread*) thread_wrappers_pthread.h:320
    #5 ThreadSanitizerStartThread ts_valgrind_intercepts.c:387
```

Conclusions

- ThreadSanitizer: dynamic detector of data races
  - C++, Linux & Mac (Valgrind), Windows (PIN)
  - Java (instrumentation with Java ASM, experimental)
- Has two modes of operation:
  - *conservative* (pure happens-before): few false reports, misses some races
  - *aggressive* (hybrid): more false positives, more real bugs
- Supports "Dynamic Annotations", a data race detector API:
  - describe custom (e.g. lock-less) synchronization
  - hide benign races
  - zero noise level even in the most aggressive mode
- Opensource

Conclusions (cont)

- Overhead is comparable to Valgrind/Memcheck
  - Slowdown: 5x-50x
  - Memory overhead: 3x-6x
- Detailed output
  - Contains all locks involved in a race and all stacks
- Runs regularly on thousands Google tests
  - Including Chromium and various server-side apps
- Found thousands of races
  - Several 'critical' (aka 'top crashers')
  - Dozens of potentially harmful
  - Tons of benign or test-only

Quiz: unsafe publication (race)

```
struct Foo {
  int a;
  Foo() { a = 42; }
};

static Foo *foo = NULL;

void Thread1() {  // Create foo.
  foo = new Foo();
}

void Thread2() {  // Consume foo.
  if (foo) {
     assert(foo->a == 42);
  }
}
```

Confirming races

- Important if using aggressive mode.
- Insert sleep (1ms-100ms) around suspected racy accesses.
- Wait for the second racy access to arrive.
- Two tools:
  - Manual (user instruments C++ code manually)
  - Automatic (Valgrind or PIN instrumentation)
- Can not confirm unsafe publication race.

Atomicity violations (high level races)

```
// If key exists, return m[key].
// Else set m[key]=val and return val.
// Runs from several threads.
int CheckMapAndInsertIfNeeded(int key, int val) {
  Map::iterator it;
  {
    ReaderLockScoped reader(&mu);
    it = m->find(key);
    if (it != m->end()) return it->first;
  }
  // <<<<< Another thread may change the map here.
  {
    WriterLockScoped writer(&mu);
    // Atomicity violation!
    ASSERT(m->find(key) == m->end());
    (*m)[key] = val;
    return val;
  }
}
```

Q&A

http://www.google.com/search?q=ThreadSanitizer