CS510 Advanced Topics in Concurrency

Jonathan Walpole

Portland State University

Threads Cannot Be Implemented as a Library

Reasoning About Programs

What are the valid outcomes for this program?

Is it valid for both r1 and r2 to contain 0 ?

```
Initialization:     x = y = 0;

 Thread 1               Thread 2

 x  = 1;                y  = 1;
 r1 = y;                r2 = x;
```

Sequential Consistency

If your intuition is based on sequential consistency, you may think that r1 = r2 = 0 is invalid

But sequential consistency does not hold on most (any?) modern architectures

- Compilers may reorder memory operations under the (invalid) assumption that the program is sequential!
- Hardware may reorder memory operations, as we will see in the next class
  - You are expected to use memory barriers if this hardware reordering would be problematic for your program!

The Problem

Languages such as C and C++ do not support concurrency

- C/C++ compilers do not implement it
- Threads are implemented in library routines (i.e. Pthreads)
- The compiler, unaware of threads, reorders operations as if it was dealing with a sequential program
- Some of this reordering is invalid for concurrent programs

But C programs with Pthreads work just fine

… don’t they?

The Pthreads Solution

Pthreads defines synchronization primitives

- Pthread_mutex_lock
- Pthread_mutex_unlock

Programmers must use them to prevent concurrent access to memory

- Define critical sections and protect them with mutex locks
- Do not allow “data races” in the program

Pthreads Synch. Primitives

Lock and unlock must be implemented carefully because they themselves contain data races!

- Use memory barriers to prevent hardware reordering
- Disable optimizations to prevent compiler reordering

This seems ok, doesn't it?

- Lock and unlock are implemented carefully
- The code between lock and unlock is executed sequentially due to mutual exclusion enforced by locks
- The code outside the critical section has no races

The Catch

The programmer needs to determine where there is a race in order to place the locking primitives

How can the programmer tell whether there is a data race?

- Need to reason about ordering of memory operations to determine if there is a race
- This requires the programming language to formally define a memory model
  - C / C++ do not! ... well, they didn't before now
  - Without a memory model, how can you tell if there is a race?

Example

Does this program contain a race?

- Is it possible for either x or y to be modified?
- Is x == y == 1 a possible outcome?

```
Initially: x = y = 0;


if (x == 1) ++y;
if (y == 1) ++x;
```

Valid Compiler Transformations

Assuming sequential consistency, there is no race

- But sequential consistency is a bad assumption!

Also, a compiler could legally transform

```
if (x == 1) ++y;
if (y == 1) ++x;
```

to

```
++y; if (x != 1) --y;
++x; if (y != 1) --x;
```

This transformation produces a race!
How can the compiler know not to do this?

The Programmer's Problem

Programmer needs to know the constraints on compiler or hardware memory reordering in order to determine whether there is a race

- Then needs to prevent the race by using the mutual exclusion primitives
- Some CPUs define a formal memory consistency model
- But the C programming language doesn't define a formal memory model, so its not possible to answer the first question with confidence in all cases!

The Compiler Developer's Problem

Compilers need to know about concurrency in order to know that its not OK to use certain sequential optimizations

- Code transformations that are valid for sequential programs but not concurrent ones
- Alternatively, the compiler must be conservative all the time, leading to performance degradation for sequential programs

Another Example

Rewriting adjacent data (bit fields)

```
struct { int a:17; int b:15 } x;
```

If thread 1 updates x.a and thread 2 updates x.b, is there a race?

Another Example

Rewriting adjacent data (bit fields)

```
struct { int a:17; int b:15 } x;
```

`x.a = 42` possibly implemented by the compiler as …

```
{
        tmp = x;              // Read both fields into a
                              // 32 bit temporary variable
        tmp &= ~0x1ffff;      // Mask off old a;
        tmp |= 42;            // Or in new value
        x = tmp;              // Overwrite all of x
}
```

The Programmer's Problem

An update to x.a also updates x.b by side effect!

- There appears to be no race in the program, but this transparent compiler optimization creates one
- Mutual exclusion primitives should have been used!

… but a separate lock for x.a and x.b will not do either!

If x.a and x.b are protected using separate locks a thread updating only x.a would need to also acquire the lock for x.b

- How is the programmer to know this?

The Compiler Developer's Problem

How can you know that this optimization (which is fine for a sequential program) is not ok?

- If your language has no concept of concurrency?

The granularity of memory operations (bit, byte, word etc) is architecture specific

- Programs that need to know this kind of detail are not portable!
- If the language defined a minimum granularity for atomic updates, what should it be?

Register Promotion

Register promotion can also introduce updates where there were none before.

Example: conditional locking for multithreading:

```
for (...) {
        ...
        if (mt) pthread_mutex_lock (...);
        x = ... x ...
        if (mt) pthread_mutex_unlock (...);
}
```

Register Promotion

Register promotion can also introduce updates where there were non before.

Example: conditional locking for multithreading:

Transformed to

Register Promotion

Register promotion can also introduce updates where there were non before.

Example: conditional locking for multithreading

```
r = x;
for (...) {
        ...
        if (mt) {
                x = r; pthread_mutex_lock (...); r = x;
        }
        r = ... r ...
        if (mt) {
                x = r; pthread_mutex_unlock (...); r = x;
        }
}
```

The Problem

The variable x is now accessed outside the critical section

The compiler caused this problem (i.e., broke the program)

- but how is the compiler supposed to know this is a mistake if it is unaware of concurrency and hence unaware of critical sections?

Bottom line: the memory abstraction is under-defined!

Other Problems

Sticking to the Pthreads rules prevents important performance optimizations

- not just in the compiler

It also makes programs based on non-blocking synchronization illegal!

- NBS programs contain data races, by definition!

Even with a formal memory model, how would we extend the Pthreads approach to cover NBS?

C++11 Memory Model

Compiler optimizations are not allowed to introduce data races!

- can be controlled via compilation flags

Atomic types and operations

- implemented via mutual exclusion
- or by lock-free algorithms
- or directly by underlying atomic instructions

Avoiding Data Races

Compiler optimizations must not introduce stores on code paths that don't have them

Packed data:

- each field/object is its own memory location
- compiler not allowed to write other memory locations
- packing small fields in to a word is no longer allowed
- use half-word or byte operations
- bit fields can still race ... could be implemented using CAS

Avoiding Data Races (cont.)

Ongoing analysis of existing optimizations to see if they generate races or not ...

Atomic Types

Operations on atomic types are indivisible

Memory coherence enforced per object

- each object has a single modification order
- atomic writes to the object are serialized

Choice of synchronization modes

- memory ordering semantics defined per operation

Ongoing work on lock-free implementations of atomic types

Synchronization Modes

Memory model *synchronization modes* give a choice of memory ordering semantics

- Sequentially Consistent (strong, the default)
- Acquire Release (weaker)
- Consume (even weaker)
- Relaxed (weakest)

Sequentially Consistent

```
-Thread 1-          -Thread 2-
y = 1               if (x.load() == 2)
x.store (2);           assert (y == 1)
```

Sequentially Consistent

```
-Thread 1-          -Thread 2-
y = 1               if (x.load() == 2)
x.store (2);           assert (y == 1)
```

The assert can not fail

Happens-before order exists between all operations

Sequentially Consistent

```
              a = 0
              y = 0
              b = 1
-Thread 1-                  -Thread 2-
x = a.load()                while (y.load() != b)
y.store (b)                    ;
while (a.load() == x)       a.store(1)
   ;
```

The final loop in thread 1 is not infinite!

Atomic operations are optimization barriers (like opaque functions)

Reordering can happen between them, but not across them!

Sequentially Consistent

```
-Thread 1-         -Thread 2-                        -Thread 3-
y.store (20);      if (x.load() == 10) {             if (y.load() == 10)
x.store (10);        assert (y.load() == 20)           assert (x.load() == 10)
                     y.store (10)
                   }
```

Sequentially Consistent

```
-Thread 1-        -Thread 2-                        -Thread 3-
y.store (20);     if (x.load() == 10) {             if (y.load() == 10)
x.store (10);       assert (y.load() == 20)           assert (x.load() == 10)
                    y.store (10)
                  }
```

Neither assert can fail

Relaxed

```
-Thread 1-
y.store (20, memory_order_relaxed)
x.store (10, memory_order_relaxed)

-Thread 2-
if (x.load (memory_order_relaxed) == 10)
  {
    assert (y.load(memory_order_relaxed) == 20) /* assert A */
    y.store (10, memory_order_relaxed)
  }

-Thread 3-
if (y.load (memory_order_relaxed) == 10)
  assert (x.load(memory_order_relaxed) == 10) /* assert B */
```

Relaxed

```
-Thread 1-
y.store (20, memory_order_relaxed)
x.store (10, memory_order_relaxed)

-Thread 2-
if (x.load (memory_order_relaxed) == 10)
  {
    assert (y.load(memory_order_relaxed) == 20) /* assert A */
    y.store (10, memory_order_relaxed)
  }

-Thread 3-
if (y.load (memory_order_relaxed) == 10)
  assert (x.load(memory_order_relaxed) == 10) /* assert B */
```

Either assert can fail !

No ordering is enforced (no happens before edges)

Only coherence order per variable is enforced

Relaxed

```
-Thread 1-
x.store (1, memory_order_relaxed)
x.store (2, memory_order_relaxed)

-Thread 2-
y = x.load (memory_order_relaxed)
z = x.load (memory_order_relaxed)
assert (y <= z)
```

Relaxed

```
-Thread 1-
x.store (1, memory_order_relaxed)
x.store (2, memory_order_relaxed)

-Thread 2-
y = x.load (memory_order_relaxed)
z = x.load (memory_order_relaxed)
assert (y <= z)
```

This assert can not fail (stores to same variable in same thread)

Once thread 2 has seen 2 in x, it can not see an earlier value.

Coherence order of x matches order of stores to x from thread 1

Acquire Release

```
-Thread 1-
y.store (20, memory_order_release);

-Thread 2-
x.store (10, memory_order_release);

-Thread 3-
assert (y.load (memory_order_acquire) == 20 && x.load (memory_order_acquire) == 0)

-Thread 4-
assert (y.load (memory_order_acquire) == 0 && x.load (memory_order_acquire) == 10)
```

Acquire Release

```
-Thread 1-
y.store (20, memory_order_release);

-Thread 2-
x.store (10, memory_order_release);

-Thread 3-
assert (y.load (memory_order_acquire) == 20 && x.load (memory_order_acquire) == 0)

-Thread 4-
assert (y.load (memory_order_acquire) == 0 && x.load (memory_order_acquire) == 10)
```

Like sequential consistency, but only for dependent variables, not between independent reads of independent writes

Both asserts can pass, because no ordering is implied between thread 1 and 2

Sequential consistency would require that if one passes the other must fail

Acquire Release

```
-Thread 1-
y = 20;
x.store (10, memory_order_release);

-Thread 2-
if (x.load(memory_order_acquire) == 10)
   assert (y == 20);
```

Acquire Release

```
-Thread 1-
y = 20;
x.store (10, memory_order_release);

-Thread 2-
if (x.load(memory_order_acquire) == 10)
   assert (y == 20);
```

Assert can not fail, because store to y happens before store to x, even though y is not atomic.

Consume

```
-Thread 1-
n = 1
m = 1
p.store (&n, memory_order_release)

-Thread 2-
t = p.load (memory_order_acquire);
assert( *t == 1 && m == 1 );

-Thread 3-
t = p.load (memory_order_consume);
assert( *t == 1 && m == 1 );
```

No happens-before ordering on non-dependent variables

Assert in thread 2 is true

Assert in thread 3 can fail

Consume

```
-Thread 1-
n = 1
m = 1
p.store (&n, memory_order_release)

-Thread 2-
t = p.load (memory_order_acquire);
assert( *t == 1 && m == 1 );

-Thread 3-
t = p.load (memory_order_consume);
assert( *t == 1 && m == 1 );
```

Review: Sequentially Consistent

```
-Thread 1-
y.store (20);
x.store (10);
```

```
-Thread 2-
if (x.load() == 10) {
  assert (y.load() == 20)
  y.store (10)
}
```

```
-Thread 3-
if (y.load() == 10)
  assert (x.load() == 10)
```

Review: Sequentially Consistent

```
-Thread 1-          -Thread 2-                       -Thread 3-
y.store (20);       if (x.load() == 10) {            if (y.load() == 10)
x.store (10);         assert (y.load() == 20)          assert (x.load() == 10)
                      y.store (10)
                    }
```

All threads see the same state

Both asserts are true

Review: Acquire Release

```
-Thread 1-        -Thread 2-                     -Thread 3-
y.store (20);     if (x.load() == 10) {          if (y.load() == 10)
x.store (10);       assert (y.load() == 20)        assert (x.load() == 10)
                    y.store (10)
                  }
```

Review: Acquire Release

```
-Thread 1-       -Thread 2-                       -Thread 3-
y.store (20);    if (x.load() == 10) {            if (y.load() == 10)
x.store (10);      assert (y.load() == 20)          assert (x.load() == 10)
                   y.store (10)
                 }
```

Only the two threads involved see the same state

Thread 2's assert is true

Thread 3's assert can fail, since thread 1 and 3 have not synchronized

Review: Relaxed

```
-Thread 1-          -Thread 2-                        -Thread 3-
y.store (20);       if (x.load() == 10) {             if (y.load() == 10)
x.store (10);         assert (y.load() == 20)           assert (x.load() == 10)
                      y.store (10)
                    }
```

Review: Relaxed

```
-Thread 1-          -Thread 2-                       -Thread 3-
y.store (20);       if (x.load() == 10) {            if (y.load() == 10)
x.store (10);         assert (y.load() == 20)          assert (x.load() == 10)
                      y.store (10)
                    }
```

Both asserts can fail