Threads cannot be Implemented as a Library

Author: Hans-J. Boehm

&&

The Java Memory Model

Authors: Jeremy Manson, William Pugh, Sarita Adve

Presenter: Chetan Bhole

Quick Recap

- **Memory Model**
  - Memory model will tell the programmer how the memory actions (reads and writes) in a program appear to execute
- **As a designer**

- Sequential Consistency Model
  - Memory actions (of the threads) execute in the same order they appear in the program

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

  - r2 == 2, r1==1 violates Sequential consistency
  - Problem: SC restricts optimizations
- Relaxed memory model
  - Reorder for performance improvement
  - so in the above example, stmt 2 can be performed before stmt 1 and stmt 4 can be performed before stmt 3 and will allow r2==2 and r1==1

- Earlier, lots of research has been done for memory models at the hardware interface
- But little research of MM in the programming language level
- MM in Programming language level is important because
  - They provide strong safety and security issues if language is type safe
  - Optimizations can be performed at a much higher level

What we look at here

- Relaxed Memory models for
  - C (language specs don't have multithreading specs) + library pthreads
  - Java which has a multithreading specification

Threads Cannot be Implemented as a library

C + pthreads

- Why a thread library almost works
  - Corrections on the thread library: based on previous painful experiences

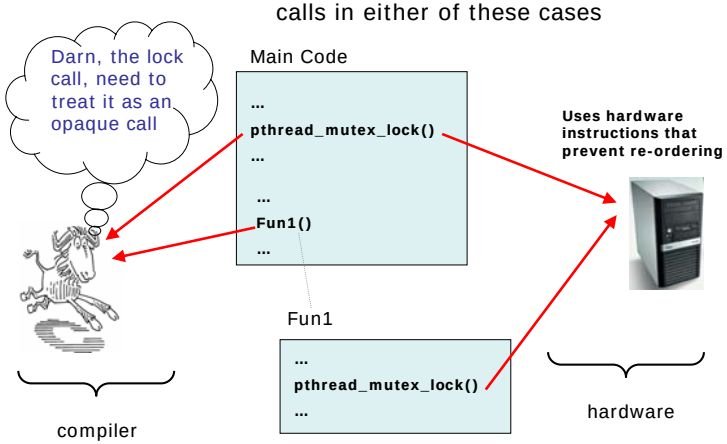
What we look at

- Concurrency achieved using pthreads
- C compiler: can produce incorrect code
- Simple case of less efficient parallel algorithm

Concurrency achieved using pthreads

- Use synchronization for mutual exclusion using pthread_mutex_lock, pthread_mutex_unlock etc
- C is not type-safe, so no guarantee of security from malicious code
- So leaves the data-race conditions of programs unspecified
- Change the compiler or hardware and some code that was working properly might start producing incorrect results

Can't reorder instructions around the calls in either of these cases



Problems

- Correctness Issues
  - Concurrent modification
  - Rewriting of adjacent data
  - Register promotion
- Performance case

Correctness Issues – I

Concurrent modification

- Define the memory model will allow to have language specifications and will allow user and compiler to know when a race will occur

- Race not possible (initial: x=y=0)

```
Thread 1   if (x == 1) ++y;
Thread 2   if (y == 1) ++x;
```

x and y always 0

- Converted by compiler for optimization puposes to a race one (initial: x=y=0)

```
Thread 1   ++y; if (x != 1) --y;
Thread 2   ++x; if (y != 1) --x;
```

can lead to x=1 and y=1

Correctness Issues – II

Rewriting of adjacent data

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

Assignment: `x.a = 42` implemented as:

Lock on x.a

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

- Language spec needs to define when adjacent data may be overwritten and provide restricting implicit writes to adjacent bit-fields

Correctness Issues – III

Register Promotion

Example

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

→

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

- Extra reads and writes when lock not held, code is broken.
- Lesson: compilers must be aware of threads and language specs must address thread specific semantic issues

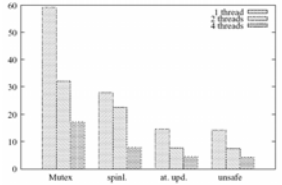
Performance

- Lock and unlock are expensive
- The atomic operations, memory barriers: 100 times the time taken compared to a register-register operation
- Hence synchronization performance can be critical and wise use of the hardware primitives and constrained shared variables necessary
- Lock-free and wait-free programming and simple atomic loads and stores may lead to performance benefits. Java has added this

Expensive Synchronization Example

```
for (my_prime = start;
     my_prime < 10000; ++my_prime)
  if (!get(my_prime)) {
    for (multiple = my_prime;
         multiple < 100000000;
         multiple += my_prime)
      if (!get(multiple)) set(multiple);
  }
```

- A[i] stores set(i) values
- On completion get(i) is false iff i is prime
- How about having N copies of the program with N threads?



Conclusion

Possible Solution:

Make changes to the C language specs and compilers

Java Memory Model

Goals of new Java memory model

- To allow programmers write more reliable multithreading code
  - Because testing to check errors might not work across platforms
  - anomalies occur rarely and non-repeatedly
- Allow implementers to create high performance JVMs

Goal of remaining talk:

Get a brief idea of the Java Memory Model

- Two important requirements of the JMM
  - Understand: correctly synchronized code
  - Guarantees of incorrect code
- Simpler model (Happens-Before Memory model)
- Causality + HB-MM = JMM
- Impact of the JMM model
- Thread Safe Lazy Initialization (if time permits)

Java Memory model: Introduction

Relaxed memory model

- guarantees Sequential Consistency to data-race-free programs
  - i.e. if your program is correctly synchronized
- Incorrect programs (data-race) : prohibited
  - to retain safety and security properties of java but trying to allow optimizations
  - The new model clearly defines boundaries of legal transformations

your program is correctly synchronized

iff all sequentially consistent executions of the program are free of data races.

- Data race: Accesses x and y form a data race if

thread 1 thread 2

at least one is a write

and $x \not\xrightarrow{hb} y$ i.e. is not Happens before Order

More definitions

- Happens before Order $x \xrightarrow{hb} y$ is the transitive closure of program order and synchronizes-with order
- Synchronizes-with order:
  - unlock(m) -> lock(m)
  - Write volatile(x) -> read volatile (x)
  - in synchronization order (i.e. total order over all synchronization actions – lock, unlock, read or write volatile variables)

Example:

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2 and r1 == 1 possible

- Correctly synchronized? No
- Solution: Declare x and y to be volatile

Incorrectly synchronized code

- Leaving out the semantics for this type of coding will affect the safety and security guarantees of Java
- And so need to provide clear and definitive semantics for incorrectly written code

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

y = 42

- Self justifying write speculation can create security violations

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

y =

y should not be assigned to this object

- Security violated say it was an object instead of 42, an object this thread was not supposed to have
- Not all such assignments are insecure
- The causality model will tell us which are secure and which are not

Happens-before memory model

- Simpler than the java memory model
- Requirements
  - Happens before order maintained
  - Actions t performs (given values of reads) in execution are same as when t generates in program order in isolation
  - Happens before consistency rule for values a non-volatile read can see
    - r -> w (then r will not see w)
    - w -> w' -> r (and r sees w, then w' cannot exist)
  - Synchronization order consistency rule for values a volatile read can see
    - w -> w' -> r (then r sees w')

Happens-before model example

Initially, x == 0, ready == false. ready is a volatile variable.

Thread 1	Thread 2
x = 1;	if (ready)
ready = true	r1 = x;

If r1 = x; executes, it will read 1.

Problems with the Happens-before model

- Correctly synchronized programs may not be satisfy SC

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

y = 42

Causality + Happens-Before = Java model

- Cause can't depend on Data
- cause eliminates data dependence

Initially, x = y = 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

Is r1 == r2 == r3 == 1 possible?

If it knew r1, r2 can take only {0,1} values

Initially, x = y = 0

Thread 1	Thread 2
2: r2 = 1;	4: r3 = y;
3: y = 1;	5: x = r3;
1: r1 = x;	

Causality

- Cause can't depend on control dependencies
- cause eliminates control dependence

Before compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

After compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: r2 = r1;	
3: if (true) ;	

r1 == r2 == r3 == 2 is sequentially consistent

problem arose because of performing the write earlier that it should have

Here the write was NOT going to take place in a SC

We don't want this behavior

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

But here, the write was going to take place in a SC

We want this behavior

Initially, x = y = 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

Is r1 == r2 == r3 == 1 possible?

Before compiler transformation

Initially, a = 0, b = 1

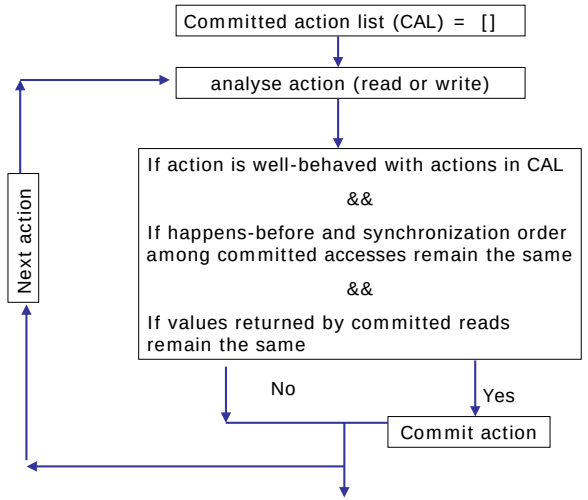
Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

Causality and the JMM

- Well behaved execution key for notion of causality
- Writes can be done early for well-behaved execution (ie consistent with SC execution)
- Well-behaved: a read that is not yet committed must return the value of a write that is ordered before it by happens-before

Causality and the JMM



Causality on examples

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

Before compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

Impact of the model

- Considerations for implementors
  - Control dependence
    - Should not reorder a write above a non-terminating loop
  - Semantics allow reordering
    - They show what kinds of reorderings are allowable
  - Other code transformations
    - Synchronization on thread local objects can be ignored
    - Volatile fields for thread local objects can be treated as normal fields
    - Redundant synchronization can be ignored

Impact of the model

- Considerations for Programmers
  - If the programmer has made sure the program is correctly synchronized, he doesn't need to worry about the re-orderings
  - The java specs do not guarantee pre-emptive multithreading or any kind of fairness (and rules are JVM specific)
  - Need to remember that some other requirement decisions were based on taste of Java builders

References

- The Java Memory Model by J. Manson et. al
- Threads cannot be implemented as a Library by H. Boehm
- The Java Memory Model, slides by J. Manson et. Al
- http://en.wikipedia.org/wiki/Double_checked_locking_pattern
- Images edited from:
  - http://www.uiowa.edu/~fyi/issues/issues2003_v41/04022004/photos/FYI%20400/BalanceRGB005.jpg
  - http://www.clipartof.com/images/clipart/xsmall2/12942_chubby_balding_man_drinking_soda_and_eating_a_chocolate_donut.jpg
  - http://www.gnu.org

Thread Safe Lazy Initialization
Double checking idiom

Original Double Checked Locking
// FIXME: THIS CODE IS BROKEN!

```
Volatile Helper helper;

Helper getHelper() {
    if (helper == null)
    synchronized(this) {
        if (helper == null)
            helper = new Helper();
    }
    return helper;
}
```

Locking in Java

- High level concurrency abstractions
  - java.util.concurrent
- Low level locking
  - **synchronized()** blocks
- Low level primitives
  - volatile variables, java.util.concurrent.atomic classes
  - allows for non-blocking synchronization